Struct Multidisc Optim (2012) 45:309–328
DOI 10.1007/s00158-011-0706-z

EDUCATIONAL ARTICLE

`PolyMesher`: a general-purpose mesh generator for polygonal elements written in Matlab

Cameron Talischi · Glaucio H. Paulino · Anderson Pereira · Ivan F. M. Menezes

Received: 9 November 2010 / Revised: 22 February 2011 / Accepted: 26 February 2011 / Published online: 8 January 2012

Abstract We present a simple and robust Matlab code for polygonal mesh generation that relies on an implicit description of the domain geometry. The mesh generator can provide, among other things, the input needed for finite element and optimization codes that use linear convex polygons. In topology optimization, polygonal discretizations have been shown not to be susceptible to numerical instabilities such as checkerboard patterns in contrast to lower order triangular and quadrilateral meshes. Also, the use of polygonal elements makes possible meshing of complicated geometries with a self-contained Matlab code. The main ingredients of the present mesh generator are the implicit description of the domain and the centroidal Voronoi diagrams used for its discretization. The signed distance function provides all the essential information about the domain geometry and offers great flexibility to construct a large class of domains via algebraic expressions. Examples are provided to illustrate the capabilities of the code, which is compact and has fewer than 135 lines.

Keywords Topology optimization · Polygonal elements · Centroidal Voronoi tessellations · Implicit geometries

Electronic supplementary material The online version of this article (doi:10.1007/s00158-011-0706-z) contains supplementary material, which is available to authorized users.

C. Talischi · G. H. Paulino (✉)
Department of Civil and Environmental Engineering, University of Illinois at Urbana-Champaign, 205 North Mathews Avenue, Newmark Laboratory, MC-250, Urbana, IL 61801, USA
e-mail: paulino@uiuc.edu

A. Pereira · I. F. M. Menezes
Tecgraf, Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Rua Marquês de São Vicente, 225, Gávea, 22453-900 Rio de Janeiro, RJ, Brazil

1 Introduction

Sharing and publication of educational software has become a tradition in the topology optimization community. For instance, in addition to the popular "99 line" code (Sigmund 2001) and its successor (Andreassen et al. 2011), Allaire and Pantz (2006) presented a structural optimization code based on FreeFem++. Liu et al. (2005) introduced a coupled level set method using the FEMLAB package and Challis (2010) presented a discrete level-set Matlab code very much in the spirit of the "99 line" code. More recently, Suresh (2010) developed a 199 line code for Pareto-optimal tracing with the aid of topological derivatives. In a series of papers, we aim to extend and complement the previous works by presenting a Matlab implementation of topology optimization that, among other things, features a general framework for finite element discretization and analysis. We also hope that the engineering community will make use of and even contribute to this computational framework.

Many engineering applications of topology optimization cannot be defined on a rectangular domain or solved on a structured square mesh. The description and discretization of the design domain geometry, specification of the boundary conditions for the governing state equation, and accurate computation of the design response may require the use of *unstructured meshes*. One main goal here is to provide the users a self-contained analysis tool in Matlab and show how the topology optimization code should be structured so as to separate the analysis routine from the particular formulation used. The latter is the subject of a companion paper (Talischi et al. 2011). We have elected to focus on polygonal discretizations in this educational effort for the following reasons: (1) The concept of Voronoi diagrams offers a simple way to discretize two-dimensional geometries with convex polygons. We will discuss a simple and robust Voronoi mesh

generation scheme that relies on an implicit description of the domain geometry; (2) Polygonal finite elements outperform linear triangles and quads in topology optimization as they are not susceptible to numerical instabilities such as checkerboard patterns (Langelaar 2007; Saxena 2008; Talischi et al. 2009, 2010); (3) The isoparametric formulation for polygonal finite elements can be viewed as extension of the common linear triangles and bilinear quads to all convex n-gons (Sukumar and Tabarraei 2004; Sukumar and Malsch 2006; Talischi et al. 2010). As a special case, these codes can generate and analyze structured triangular and quadrilateral meshes.

The main ingredients of our mesh generator are the implicit representation of the domain and the use of Centroidal Voronoi diagrams for its discretization. The signed distance function contains all the essential information about the meshing domain needed in our mesh algorithm. Inspired by the work of Persson and Strang (2004), we note that this implicit description provides great flexibility to construct a relatively large class of domains with algebraic expressions. A discretization of the domain is constructed from a Centroidal Voronoi Tessellation (CVT) that incorporates an approximation to its boundary. The approximation is obtained by including the set of reflections of the seeds (Bolander and Saito 1998; Yip et al. 2005). The Lloyd's method is used to establish a uniform (optimal) distribution of seeds and thus a high quality mesh (Talischi et al. 2010). We remark that CVTs have been previously used for generation and analysis of triangular discretizations (see, for example, Du and Gunzburger 2002; Du et al. 2003; Ju et al. 2006) and, in some cases, superconvergence of numerical solutions has been observed (Huang et al. 2008).

The remainder of this paper is organized as follows: in the next two sections, we review the main concepts and recall the properties of signed distance functions, Voronoi diagrams and CVTs before discussing the details of the proposed meshing algorithm in Section 4. We explain the Matlab implementation of this algorithm in Section 5 and present numerical examples in Section 6. Potential extensions and generalizations of the code are addressed in Section 7. An added educational aspect of the present work consists of demonstrating how intuitive and geometrical concepts can be expressed in the language of mathematics and ultimately linked with feasible computational algorithms.

2 Distance function and implicit representation

Let Ω be a subset of $\mathbb{R}^2$. The *signed distance function* associated with Ω is the mapping $d_\Omega : \mathbb{R}^2 \to \mathbb{R}$ defined by:

$$d_\Omega(\mathbf{x}) = s_\Omega(\mathbf{x}) \min_{\mathbf{y}\in\partial\Omega} \|\mathbf{x}-\mathbf{y}\| \tag{1}$$

where $\partial\Omega$ denotes the boundary of Ω, $\|\cdot\|$ is the standard Euclidean norm in $\mathbb{R}^2$ (so $\|\mathbf{x}-\mathbf{y}\|$ here is the distance between $\mathbf{x}$ and point $\mathbf{y}$ on the boundary of the domain), and the sign function is given by:

$$s_\Omega(\mathbf{x}) := \begin{cases} -1, & \mathbf{x}\in\Omega \\ +1, & \mathbf{x}\in\mathbb{R}^2\backslash\Omega \end{cases} \tag{2}$$

Thus, if $\mathbf{x}$ lies inside the domain Ω, $d_\Omega(\mathbf{x})$ is minus the distance of $\mathbf{x}$ to the closest boundary point. The following characterizations are immediate from this definition:

$$\begin{aligned} \overline{\Omega} &= \left\{\mathbf{x}\in\mathbb{R}^2 : d_\Omega(\mathbf{x}) \le 0\right\}, \\ \partial\Omega &= \left\{\mathbf{x}\in\mathbb{R}^2 : d_\Omega(\mathbf{x}) = 0\right\} \end{aligned} \tag{3}$$

In our meshing algorithm, we need to determine if a candidate point (seed) $\mathbf{x}$ lies in the interior of domain Ω. With an explicit representation of Ω, based on parametrization of its boundary, this may be difficult as it requires counting the number of times a ray connecting $\mathbf{x}$ and some exterior point intersects the boundary (Osher and Fedkiw 2003). Given the signed distance function, we recover this information by evaluating the sign of $d_\Omega(\mathbf{x})$ (see Fig. 1).

Other useful information about the domain geometry is provided by the signed distance function. Its gradient, ∇d_Ω, gives the *direction* to the nearest boundary point. If Ω has smooth boundary and $\mathbf{x}\in\partial\Omega$, then $\nabla d_\Omega(\mathbf{x})$ is the unit vector normal to the boundary. In general, for almost every point $\mathbf{x}\in\mathbb{R}^2$, we have:

$$\|\nabla d_\Omega(\mathbf{x})\| = 1 \tag{4}$$

It is possible that the distance function exhibits kinks even when $\partial\Omega$ is smooth. In particular, if $\mathbf{x}$ is equidistant to more than one point of $\partial\Omega$, then $\nabla d_\Omega(\mathbf{x})$ fails to exist. As illustrated in Fig. 2b, the variation of the distance function changes depending on which boundary point is approached. In such a case, numerical differentiation may result in $\|\nabla d_\Omega(\mathbf{x})\| \neq 1$.

In the proposed algorithm, we use the property of the gradient to find the reflection of $\mathbf{x}$ about the closest boundary point. Denoting the reflection by $R_\Omega(\mathbf{x})$, we have (cf. Fig. 2a):

$$R_\Omega(\mathbf{x}) = \mathbf{x} - 2d_\Omega(\mathbf{x})\nabla d_\Omega(\mathbf{x}) \tag{5}$$

Note that this expression is valid for both interior and exterior points, i.e., for any $\mathbf{x}\in\mathbb{R}^2$.

We can see from the discussion so far that when Ω is characterized by its signed distance function, a great deal

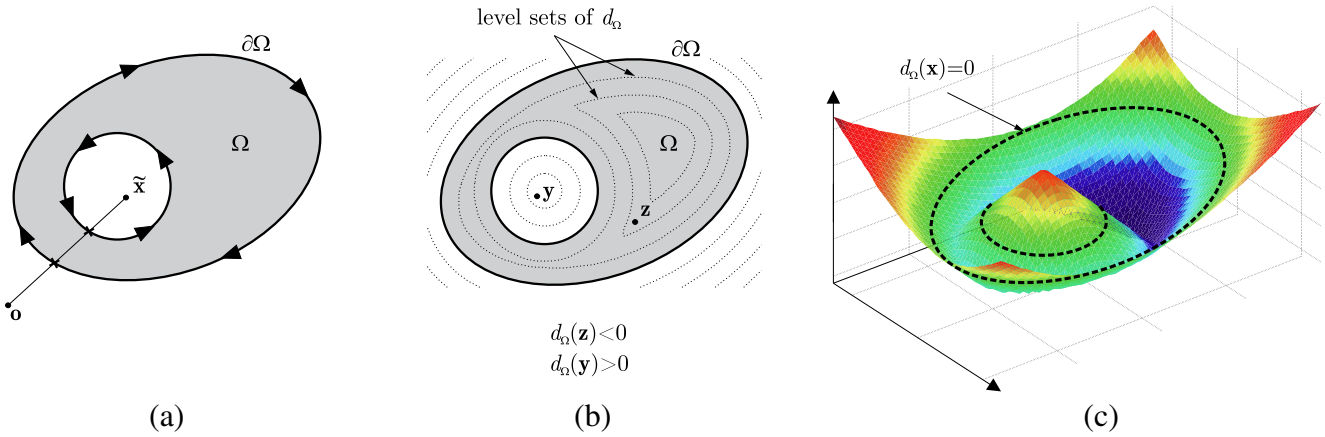


Fig. 1 **a** Explicit parametrization of domain boundary: the ray connecting point $\tilde{\mathbf{x}}$ to point **o**, known to lie outside the domain, intersects $\partial\Omega$ an even number of times, indicating $\tilde{\mathbf{x}} \notin \Omega$; **b** Implicit representation of the domain: the sign of the distance function $d_\Omega(\mathbf{x})$ determines if **x** lies inside the domain; **c** Surface plot of the signed distance function: note that $\partial\Omega$ is given by the zero level set of d_Ω

of useful information about Ω can be readily extracted. An essential task then is to construct d_Ω for a given domain Ω that we wish to discretize. For many simple geometries, the signed distance function can be readily identified. For example, if Ω is a circle of radius r centered at point $\mathbf{x}_o$, its distance function is given by:

$$d_\Omega(\mathbf{x}) = \|\mathbf{x} - \mathbf{x}_o\| - r \tag{6}$$

Moreover, set operations such as union, intersection, and complementation can be used to piece together and combine different geometries. Given domains Ω_1 and Ω_2, the expressions

$$\begin{aligned} d_{\Omega_1 \cup \Omega_2}(\mathbf{x}) &= \min\left(d_{\Omega_1}(\mathbf{x}), d_{\Omega_2}(\mathbf{x})\right) \\ d_{\Omega_1 \cap \Omega_2}(\mathbf{x}) &= \max\left(d_{\Omega_1}(\mathbf{x}), d_{\Omega_2}(\mathbf{x})\right) \\ d_{\mathbb{R}^2 \setminus \Omega_1}(\mathbf{x}) &= -d_{\Omega_1}(\mathbf{x}) \end{aligned} \tag{7}$$

capture the "sign" property of the distance function for the combined geometry, as illustrated in Fig. 3. However, we note that the "distance" property may not necessarily hold

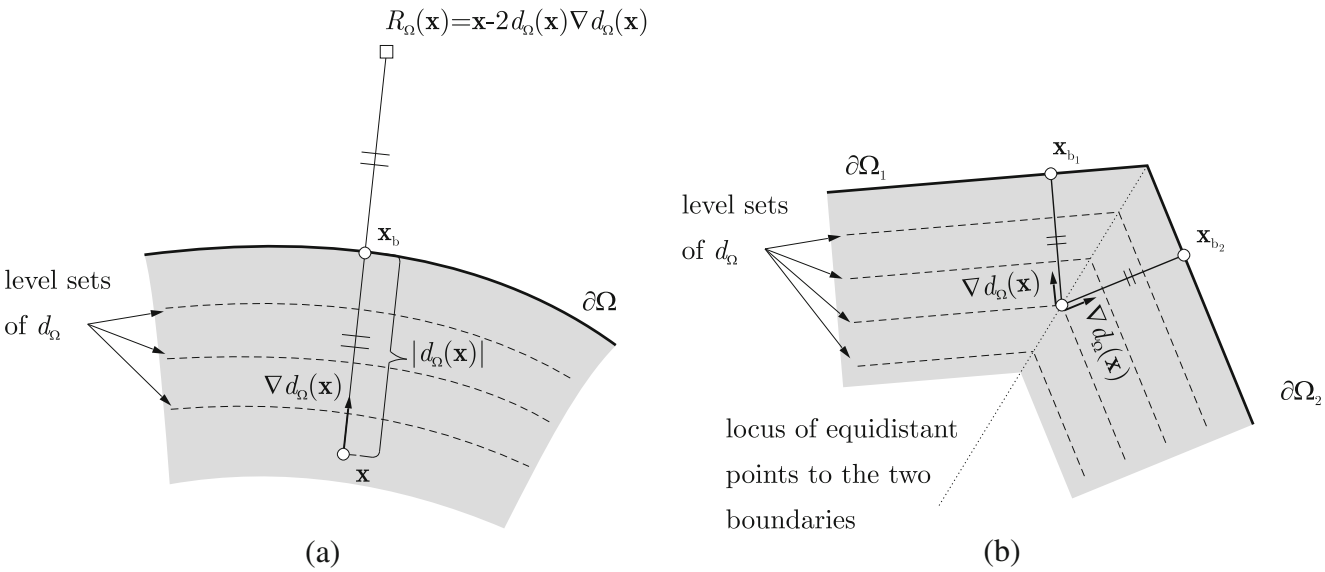


Fig. 2 **a** For $\mathbf{x} \in \mathbb{R}^2$, the direction to the closest boundary point, $\mathbf{x}_b$, is given by $\nabla d_\Omega(\mathbf{x})$, which can be used to compute the reflection $R_\Omega(\mathbf{x})$; **b** The distance function exhibit kinks at points that are equidistant to more than one boundary point. Here $\nabla d_\Omega(\mathbf{x})$ denotes the one-sided gradient at such a point **x**

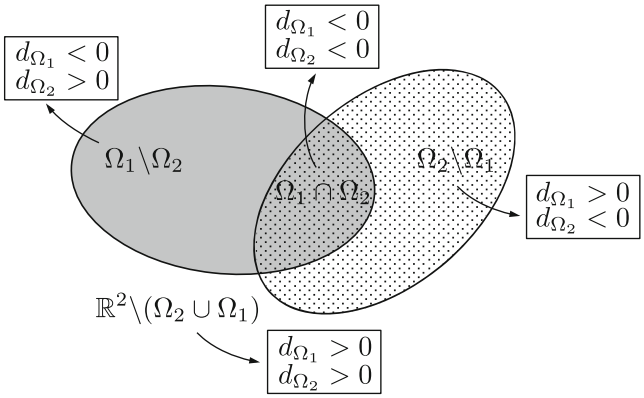


Fig. 3 Correspondence between set operations and the sign of distance functions

everywhere.[1] A reference commonly cited in conjunction with these equations is the work of Ricci (1973) but there the implicit functions do not carry the distance property. In our meshing algorithm, we require access to the distance functions of the constituent domains, in part, to address this issue.

Transformations such as rotation and translation can also be incorporated to obtain desired geometries. For example, if $\mathbf{T}_\theta$ is the matrix for rotation by angle θ about the origin, the signed distance function for rotated domain Ω_θ is given by:

$$d_{\Omega_\theta}(\mathbf{x}) = d_\Omega(\mathbf{T}_\theta^{-1}\mathbf{x}) \tag{8}$$

Also, a signed distance function can be obtained from the level sets of a given implicit function by solving a nonlinear system of equations (cf. Persson and Strang 2004) or more generally using marching algorithms (see, for example, (Sussman et al. 1998; Sethian 1999; Osher and Fedkiw 2003; Zhao 2004)). We will revisit the issue of computing distance functions in relation to our meshing algorithm and later through examples.

3 Voronoi diagrams, CVTs, Lloyd's algorithm

The concept of Voronoi diagrams plays a central role in our meshing algorithm. Given a set of n distinct points or *seeds* $\mathbf{P}$, the *Voronoi tessellation*[2] of the domain $\Delta \subset \mathbb{R}^2$ is defined by:

$$\mathcal{T}(\mathbf{P};\Delta) = \left\{V_\mathbf{y} \cap \Delta : \mathbf{y} \in \mathbf{P}\right\} \tag{9}$$

[1] For example, consider $\Omega_1 = \left\{(x_1,x_2) \in \mathbb{R}^2 : x_1 < 0\right\}$ and $\Omega_2 = \left\{(x_1,x_2) \in \mathbb{R}^2 : x_2 < 0\right\}$. The formula $d_{\Omega_1\cup\Omega_2}(\mathbf{x}) = \min\left(d_{\Omega_1}(\mathbf{x}), d_{\Omega_2}(\mathbf{x})\right)$ has incorrect distance "value" in the third quadrant, i.e., for $x_1 < 0$, $x_2 < 0$. In this region, the closest boundary point is the *new* corner $\mathbf{x} = (0,0)$ formed by the union operation.

[2] A *tessellation* or *tiling* of Δ is a collection of open sets S_i such that $\cup_i \overline{S}_i = \overline{\Delta}$ and $S_i \cap S_j = \emptyset$ if $i \neq j$.

where $V_\mathbf{y}$ is the *Voronoi cell* associated with point $\mathbf{y}$:

$$V_\mathbf{y} = \left\{\mathbf{x} \in \mathbb{R}^2 : \|\mathbf{x}-\mathbf{y}\| < \|\mathbf{x}-\mathbf{z}\|, \forall \mathbf{z} \in \mathbf{P}\backslash\{\mathbf{y}\}\right\} \tag{10}$$

Therefore $V_\mathbf{y}$ consists of points in the plane closer to $\mathbf{y}$ than any other point in $\mathbf{P}$. To simplify the notation for the meshing algorithm, we are defining the cells over the entire $\mathbb{R}^2$, while it is common in the literature to define $V_\mathbf{y} \cap \Delta$ to be the Voronoi cell (see Fig. 4).

The properties of Voronoi diagrams have been studied extensively and we refer the reader to review paper (Aurenhammer 1991) on the topic. One relevant property in two dimensions is that if a Voronoi cell is bounded, it is necessarily a convex polygon since it is formed by finite intersection of half-planes (each of which is a convex set). Hence, as we shall see in the next section, our meshing algorithm produces discretizations consisting only of *convex* polygons. This is pertinent to the isoparametric formulation for polygonal finite elements which requires convexity of all the elements in the mesh (Sukumar and Tabarraei 2004; Sukumar and Malsch 2006; Talischi et al. 2010).

The regularity of Voronoi diagrams is determined entirely by the distribution of the generating point set. A random or quasi-random set of generators may lead to a discretization not suitable for use in finite element analysis. Therefore, we restrict our attention to a special class of Voronoi tessellations that enjoy a higher level of regularity. A Voronoi tessellation $\mathcal{T}(\mathbf{P};\Delta)$ is *centroidal* if for every $\mathbf{y} \in \mathbf{P}$:

$$\mathbf{y} = \mathbf{y}_c \qquad \text{where} \qquad \boxed{\mathbf{y}_c := \frac{\int_{V_\mathbf{y}\cap\Delta} \mathbf{x}\mu(\mathbf{x})d\mathbf{x}}{\int_{V_\mathbf{y}\cap\Delta} \mu(\mathbf{x})d\mathbf{x}}} \tag{11}$$

and $\mu(\mathbf{x})$ is a given density function defined over Δ. Hence, in a Centroid Voronoi Tessellation (CVT), each generating point $\mathbf{y}$ coincides with the centroid $\mathbf{y}_c$ of the corresponding region (i.e., $V_\mathbf{y} \cap \Delta$).

An alternative variational characterization of a CVT is based on the deviation of each Voronoi region from its generating seed, measured by the following *energy* functional:

$$\mathcal{E}(\mathbf{P};\Delta) = \sum_{\mathbf{y}\in\mathbf{P}} \int_{V_\mathbf{y}(\mathbf{P})\cap\Delta} \mu(\mathbf{x}) \|\mathbf{x}-\mathbf{y}\|^2 d\mathbf{x} \tag{12}$$

Note that the energy depends on points in $\mathbf{P}$ not only through the appearance of $\mathbf{y}$ in the integral but also the Voronoi cells in domain of the integral. Critical points of $\mathcal{E}(\mathbf{P};\Delta)$ are point sets that generate CVTs since the gradient of energy functional with respect to a given $\mathbf{y} \in \mathbf{P}$ is given by (Du et al. 1999; Liu et al. 2009):

$$\nabla_\mathbf{y}\mathcal{E} = 2m_\mathbf{y}(\mathbf{y}-\mathbf{y}_c) \quad \text{where} \quad m_\mathbf{y} = \int_{V_\mathbf{y}\cap\Delta} \mu(\mathbf{x})d\mathbf{x} \tag{13}$$

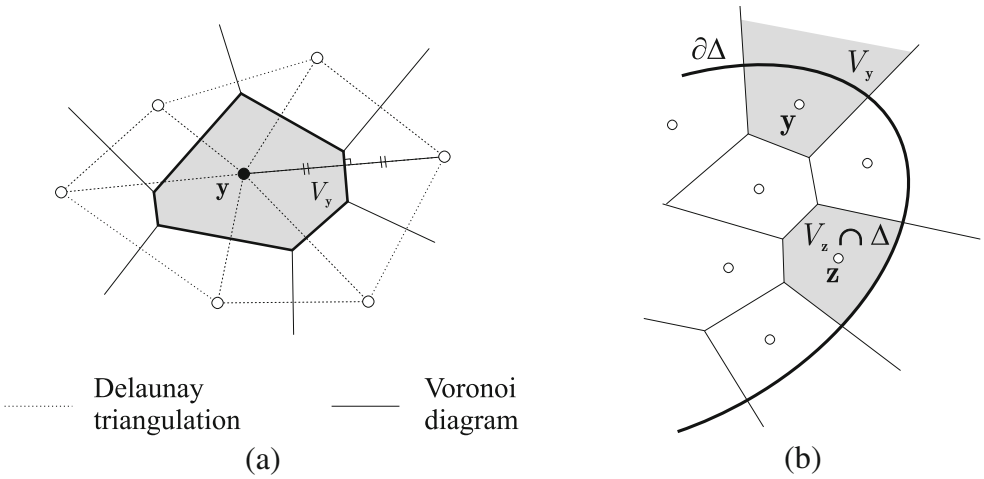


Fig. 4 **a** Voronoi diagram and its dual, the Delaunay triangulation; **b** Illustrating the difference between $V_{\mathbf{y}}$, defined in (10) as the Voronoi cell, and $V_{\mathbf{y}} \cap \Delta$, as the regions making up the Voronoi tessellation of Δ (cf. 9)

Clearly $\nabla_{\mathbf{y}}\mathcal{E} = \mathbf{0}$ when relation (11) holds. Moreover, CVTs can be further classified based on the minimization of the energy functional. The CVTs corresponding to saddle points of $\mathcal{E}$ are called *unstable* while local and global minimizers (for fixed number of seeds) of the energy functional are known as *stable* and *optimal* CVTs, respectively (Du and Wang 2005; Liu et al. 2009). The CVTs in the latter groups form a more compact tessellation of the domain and due

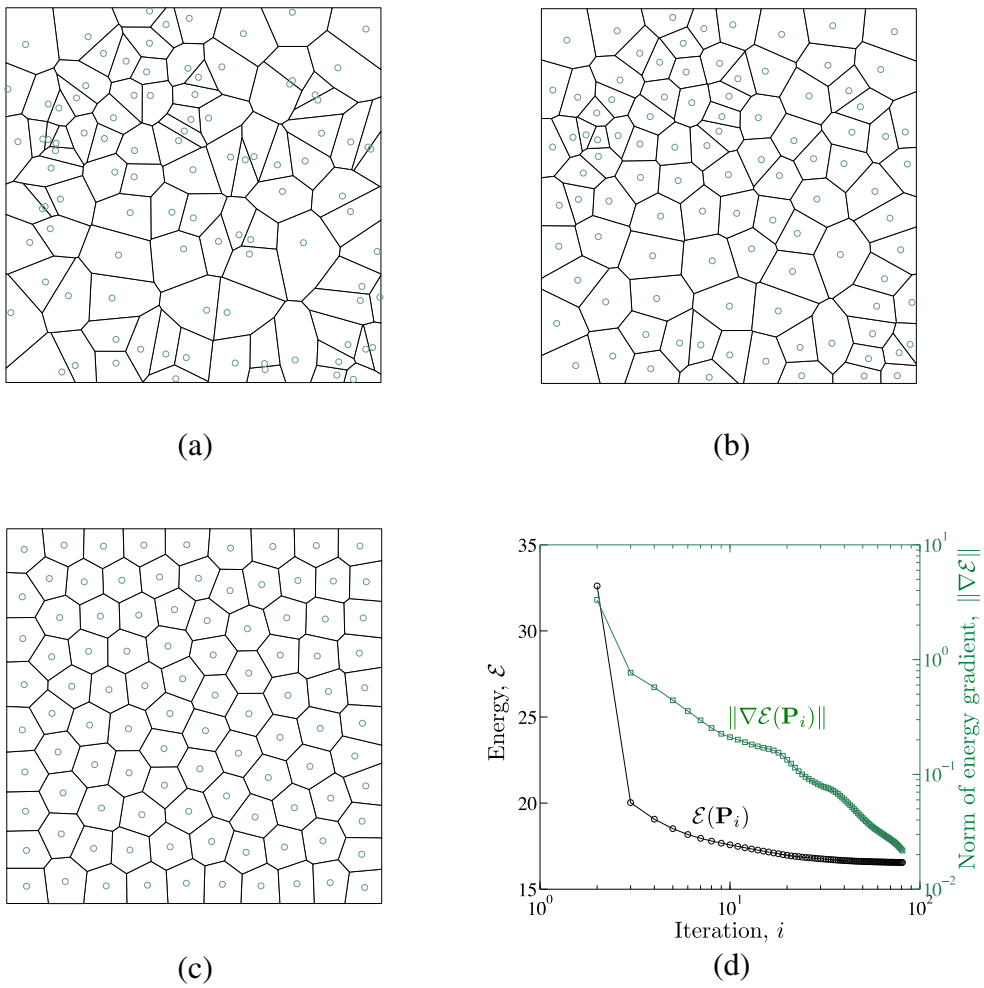


Fig. 5 **a** Random initial point set $\mathbf{P}_1$ and the corresponding Voronoi diagram; **b** First iteration of Lloyd's method: the Voronoi diagram generated by $\mathbf{P}_2 = \mathbf{L}(\mathbf{P}_1)$, i.e., the centroids of the Voronoi cells of $\mathbf{P}_1$; **c** Distribution of seeds and the diagram after 80 iterations (d) Monotonic convergence of the energy functional and decay in the norm of its gradient

to this property find many applications in areas other than mesh generation—see Du et al. (1999) for a survey on the topic.

A simple but powerful method for computing CVTs is the *Lloyd's algorithm*, which iteratively replaces the given generating seeds by the centroids of the corresponding Voronoi regions. Lloyd's algorithm can be thought of as a fixed point iteration for the mapping $\mathbf{L} = \left(\mathbf{L}_\mathbf{y}\right)_{\mathbf{y}\in\mathbf{P}}^{T} : \mathbb{R}^{n\times 2} \to \mathbb{R}^{n\times 2}$ where each component function is given by:

$$\mathbf{L}_\mathbf{y}(\mathbf{P}) = \frac{\int_{V_\mathbf{y}(\mathbf{P})\cap\Delta} \mathbf{x}\mu(\mathbf{x})d\mathbf{x}}{\int_{V_\mathbf{y}(\mathbf{P})\cap\Delta} \mu(\mathbf{x})d\mathbf{x}} \tag{14}$$

Therefore, $\mathbf{L}$ maps the point set $\mathbf{P}$ to the set of centroids of the Voronoi cells in $\mathcal{T}(\mathbf{P};\Delta)$. Given an initial point set $\mathbf{P}_1$, the Lloyd's method produces point set $\mathbf{P}_{k+1} = \mathbf{L}(\mathbf{P}_k)$ at the kth iteration. From the above relation, it is clear that a fixed point of this map, i.e., one that satisfies $\mathbf{P} = \mathbf{L}(\mathbf{P})$, forms a CVT. In Du et al. (2006), it is shown that the energy functional decreases in consecutive iterations of Lloyd's algorithm, that is,

$$\mathcal{E}(\mathbf{P}_{i+1};\Delta) \le \mathcal{E}(\mathbf{P}_i;\Delta) \tag{15}$$

which means that the Lloyd's algorithm can be viewed as a descent method for the energy functional. This property is illustrated in Fig. 5. As discussed later, Lloyd's algorithm is incorporated in our meshing algorithm to construct more uniform polygonal meshes.

4 Voronoi meshing

Before discussing the details of the proposed meshing algorithm, we illustrate the main ideas based on the concepts developed so far. As shown by Bolander et al. (Bolander and Saito 1998; Yip et al. 2005), a polygonal discretization can be obtained from the Voronoi diagram of a given set of seeds and their reflections.

4.1 Explanation of the approach

Assume $\Omega \subset \mathbb{R}^2$ is a bounded *convex* domain with smooth boundary and $\mathbf{P}$ is a given set of distinct seeds in Ω. To construct a polygonal discretization of Ω, we first reflect each point in $\mathbf{P}$ about the *closest* boundary point of Ω and denote the resulting set of points by $R_\Omega(\mathbf{P})$:

$$R_\Omega(\mathbf{P}) := \{R_\Omega(\mathbf{y}) : \mathbf{y} \in \mathbf{P}\} \tag{16}$$

Convexity of Ω ensures that all of the reflected points lie outside of Ω. We then construct the Voronoi diagram of the plane by including the original point set as well as its reflection. In other words, we compute $\mathcal{T}(\mathbf{P}\cup R_\Omega(\mathbf{P});\mathbb{R}^2)$. If the Voronoi cells of a point $\mathbf{y}$ and its reflection have a common edge, i.e., if $V_\mathbf{y}\cap V_{R_\Omega(\mathbf{y})} \neq \emptyset$, then this edge is tangent to $\partial\Omega$ at the $\mathbf{y}_b$ (see Fig. 6). Therefore, these edges form an approximation to the domain boundary and a reasonable discretization of Ω is given by the collection of Voronoi cells corresponding to the points in $\mathbf{P}$. For a given point set $\mathbf{P}$, such a discretization is uniquely defined and is denoted by $\mathcal{M}_\Omega(\mathbf{P})$. Thus, we have:

$$\mathcal{M}_\Omega(\mathbf{P}) = \left\{V_\mathbf{y} \in \mathcal{T}(\mathbf{P}\cup R_\Omega(\mathbf{P});\mathbb{R}^2) : \mathbf{y}\in\mathbf{P}\right\} \tag{17}$$

We further note that the convexity of Ω implies that the boundary edges lie on the exterior of the domain and so the discretization $\mathcal{M}_\Omega(\mathbf{P})$ covers Ω.

Clearly a better approximation is obtained if the points in $\mathbf{P}$ are distributed more "evenly" in Ω. In our algorithm, we will incorporate Lloyd's iterations to obtain a point set $\mathbf{P}$ that produces a CVT. Since optimal CVTs consist of Voronoi cells that are congruent to a basic cell (and thus are uniform in size) (Du and Wang 2005), it is expected $\cup_{V\in\mathcal{M}_\Omega(\mathbf{P})} V \approx \Omega$ especially for a large number of generating points. This gives a systematic and consistent approach for discretizing Ω under the given assumptions.

4.2 Algorithm

The basic ideas laid out in the previous section can be extended for discretization of more general domains, in particular those that are non-convex and have piecewise smooth boundaries (e.g. $\partial\Omega$ has corner points where there is a jump

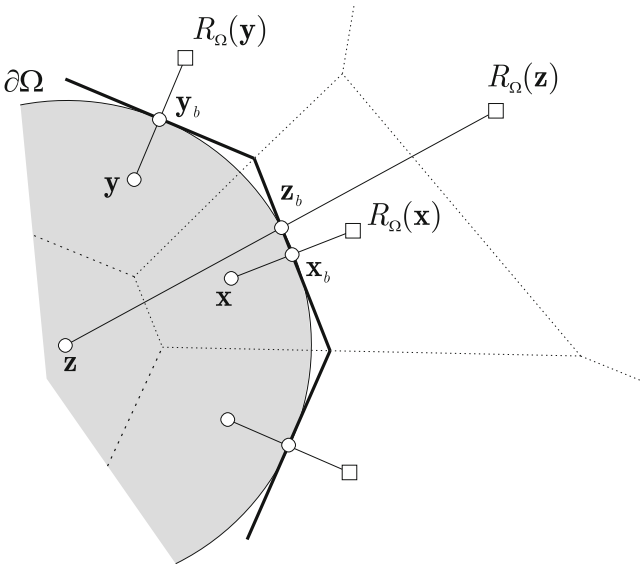


Fig. 6 Illustration of the meshing approach: the Voronoi edges shared between seeds and their reflection approximate the boundary of the domain. Note that the reflections of the interior seeds "far" from the boundary (e.g. point $\mathbf{z}$ in the figure) do not contribute to the final mesh

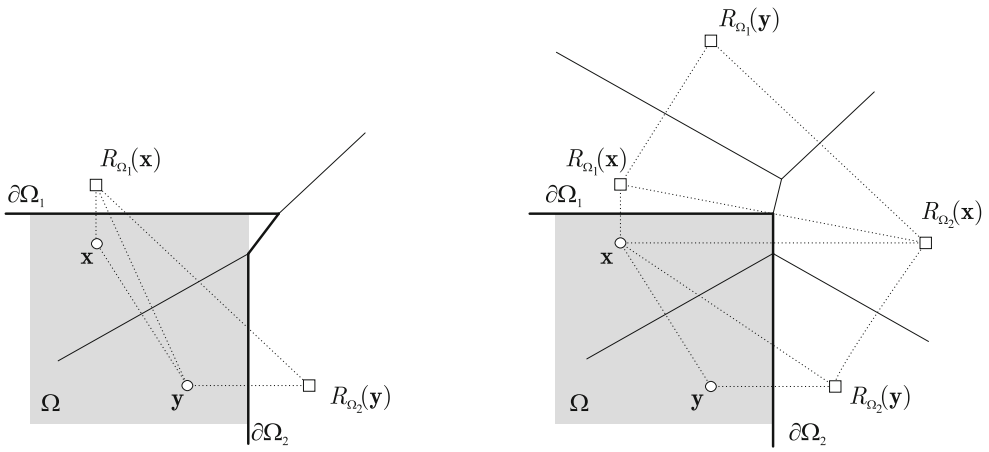


Fig. 7 To accurately capture a corner, nearby seeds need to be reflected about both boundary segments incident on that corner

in the normal vector). These features lead to a number of complications that require (minor) modifications of the previous approach. For example, reflecting a point about the nearest boundary point may not be sufficient to capture a nearby corner (see Fig. 7). We resolve this issue by reflecting the seeds about *both* boundary segments incident on the corner. Similarly, for non-convex domains, reflection of a seed far from the boundary may land inside the domain or interfere with the reflection of another seed (Fig. 8). We check the sign and value of the distance function to exclude such a scenario. Finally, as seen in Fig. 6, the reflection of most of the seeds in the interior of the domain has no effect on the approximation of the boundary. Thus, we add a condition to reflect only seeds that are in a band near the boundary. This significantly reduces the computational cost and improves the robustness of algorithm by alleviating the problem of interference, which is important in dealing with complex non-convex domains (Fig. 9). Based on these considerations, the following algorithm is proposed.

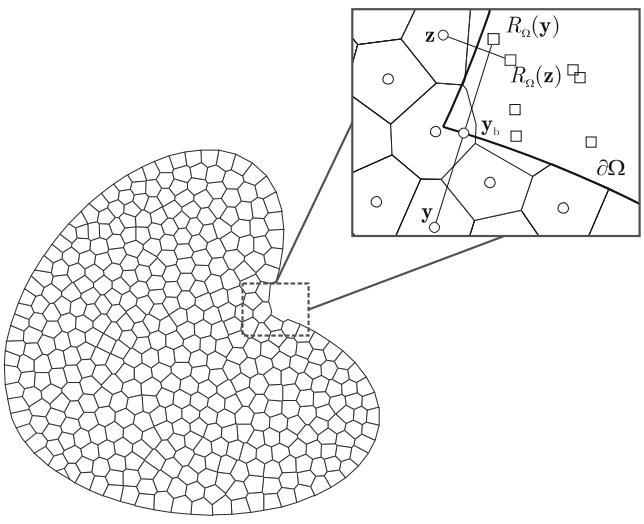


Fig. 8 In this non-convex domain, the reflection $R_\Omega(\mathbf{y})$ is closer to the boundary of the domain than the seed $\mathbf{y}$ itself, i.e., $|d_\Omega(R_\Omega(\mathbf{y}))| < |d_\Omega(\mathbf{y})|$. Not only this reflection does not contribute to the approximation of the boundary, it causes interference with seed $\mathbf{z}$

We consider domains Ω that are formed by finite union, intersection and/or difference of smooth but perhaps unbounded regions Ω_i, $i = 1, \ldots, m$. Using formulas in (7), the distance function associated with Ω can be written as a function of d_{Ω_i}:

$$d_\Omega(\mathbf{x}) = F(d_{\Omega_1}(\mathbf{x}), \ldots, d_{\Omega_m}(\mathbf{x})) \tag{18}$$

It is expected that Ω_i's and operations represented by F are defined in such a way that distance to the *every boundary segment of* Ω *can be found among the values of* d_{Ω_i}.

The first step in the algorithm is to generate an initial set of points $\mathbf{P}$. Algorithm 1 shows the basic steps for obtaining a random point set of n size. The implementation of random seed generation is simplified by specifying a bounding box $B = [x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$ that contains Ω. A random seed $\mathbf{y} \in B$ is accepted only if it lies inside the domain and this is determined by evaluating the sign of the $d_\Omega(\mathbf{y})$.

Algorithm 1 Initial random seed placement

input: B, n *%%$B \supset \Omega$ is the bounding box and n is the desired number of seeds*
  set $\mathbf{P} = \emptyset$
  while $|\mathbf{P}| < n$ **do**
    generate random point $\mathbf{y} \in B$
    if $d_\Omega(\mathbf{y}) < 0$ **then**
      $\mathbf{P} \leftarrow \mathbf{P} \cup \{\mathbf{y}\}$
    end if
  end while
output: $\mathbf{P}$

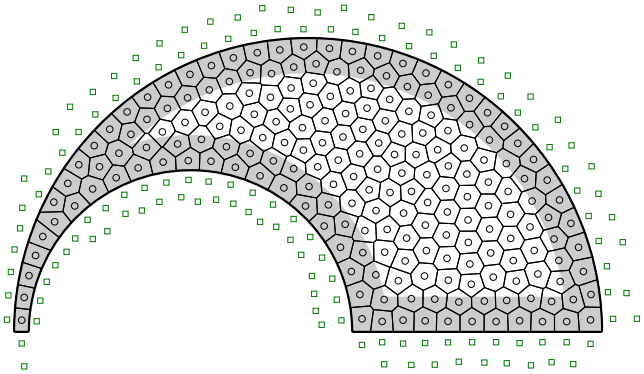


Fig. 9 In the algorithm, only the seeds in band of length $\alpha(n, \Omega)$ near the boundary (*shaded area* in the figure) are reflected

As discussed above, the set of reflections must be chosen carefully in order to deal with possible non-convex and non-smooth features of Ω. In (17), $R_\Omega(\mathbf{P})$ may not be sufficient for producing a good discretization $\mathcal{M}_\Omega$. The procedure for computing the new set of reflections, denoted by $R(\mathbf{P})$, is outlined in Algorithm 2. A seed $\mathbf{y} \in \mathbf{P}$ is reflected about boundary segment $\partial\Omega_i$ provided that:

$$\left|d_{\Omega_i}(\mathbf{y})\right| < \alpha(n, \Omega) \tag{19}$$

where $\alpha(n, \Omega)$ is a distance value proportional to the width of an element:

$$\alpha(n, \Omega) := c\left(\frac{|\Omega|}{n}\right)^{1/2} \tag{20}$$

We choose the constant of proportionality c to be greater than 1 so that α is larger than the average element width. Note that convex corners are captured as nearby seeds are reflected about both boundary segments incident on these corners.

The reflection $\overline{\mathbf{y}} = R_{\Omega_i}(\mathbf{y})$ is accepted if it lies outside the domain, i.e., $d_\Omega(\overline{\mathbf{y}}) > 0$. Moreover, the following criterion is added to avoid interference with the reflection of other seeds:

$$|d_\Omega(\overline{\mathbf{y}})| > \eta\left|d_{\Omega_i}(\mathbf{y})\right| \tag{21}$$

where $0 < \eta < 1$ is a specified parameter to adjust for numerical errors (round-off and numerical differentiation). Figure 8 illustrates the idea behind this criterion. In the case of a convex domain, $|d_\Omega(\overline{\mathbf{y}})| = \left|d_{\Omega_i}(\mathbf{y})\right|$ so no difficulty will occur. In general, however, the reflection may be closer to a boundary segment of Ω, other than the one that generated it (i.e., Ω_i), in which case the reflection will not help with the approximation of the boundary and may possibly interfere with the reflection of another seed. In particular, we expect an interference when $d_\Omega(\overline{\mathbf{y}}) < -d_{\Omega_i}(\mathbf{y})$. Hence we exclude this possibility by accepting only the reflections that satisfy the condition in (21).

Algorithm 2 Reflection

input: $\mathbf{P}, \alpha, \eta$
  $R(\mathbf{P}) = \emptyset$
  for each $\mathbf{y} \in \mathbf{P}$ **do**
    if $|d_\Omega(\mathbf{y})| < \alpha$ **then**
      for $i = 1$ to m **do** *%% m is the number of regions*
        if $\left|d_{\Omega_i}(\mathbf{y})\right| < \alpha$ **then**
          let $\overline{\mathbf{y}} = R_{\Omega_i}(\mathbf{y})$
          if $d_\Omega(\overline{\mathbf{y}}) > 0$ **and** $|d_\Omega(\overline{\mathbf{y}})| > \eta\left|d_{\Omega_i}(\mathbf{y})\right|$ **then**
            $R(\mathbf{P}) \leftarrow R(\mathbf{P}) \cup \{\overline{\mathbf{y}}\}$
          end if
        end if
      end for
    end if
  end for
output: $R(\mathbf{P})$

Once the set of reflections are determined, the Voronoi diagram of the $\mathbf{P} \cup R(\mathbf{P})$ is constructed. For each $\mathbf{y} \in \mathbf{P}$, we compute the centroid of Voronoi cell $V_\mathbf{y}$ to complete the first iteration of the meshing routine. Following the idea of Lloyd's algorithm, the set of centroids $\mathbf{P}_c$ will replace $\mathbf{P}$ in the next iteration until convergence is achieved. The convergence criterion is based on the magnitude of the gradient of the energy functional. In particular, the algorithm should terminate when $\|\nabla\mathcal{E}\| \approx 0$. Recalling (13), the norm of the gradient is given by:

$$\|\nabla\mathcal{E}\| := \left(\sum_{\mathbf{y}\in\mathbf{P}} \left\|\nabla_\mathbf{y}\mathcal{E}\right\|^2\right)^{1/2} = \left(\sum_{\mathbf{y}\in\mathbf{P}} m_\mathbf{y}^2 \|\mathbf{y} - \mathbf{y}_c\|^2\right)^{1/2} \tag{22}$$

We can see that this quantity is in fact a measure of the movement of the seeds in two consecutive iterations, weighted by the "size" of their Voronoi cells through terms $m_\mathbf{y}$. To identify the appropriate convergence tolerance, we note:

$$m_\mathbf{y} \propto \frac{1}{n}\int_\Omega \mu(\mathbf{x})d\mathbf{x}, \quad \|\mathbf{y} - \mathbf{y}_c\|^2 \propto \left|V_\mathbf{y}\right| \propto \frac{|\Omega|}{n} \tag{23}$$

and so the norm of the gradient scales as:

$$\begin{aligned}\|\nabla\mathcal{E}\| &\propto \left[n\left(\frac{1}{n}\int_\Omega \mu(\mathbf{x})d\mathbf{x}\right)^2 \frac{|\Omega|}{n}\right]^{1/2} \\ &= \frac{|\Omega|^{1/2}}{n}\int_\Omega \mu(\mathbf{x})d\mathbf{x}\end{aligned} \tag{24}$$

We define the following "non-dimensional" error parameter:

$$E_r := \frac{n\,\|\nabla\mathcal{E}\|}{|\Omega|^{1/2}\int_\Omega \mu(\mathbf{x})d\mathbf{x}} \tag{25}$$

Algorithm 3 Main function

input: n, M, ϵ_{tol} *%%number of seeds n, max. number of iterations M, tolerance* ϵ_{tol}
  generate $\mathbf{P}$, a random point set of size n
  $i \leftarrow 0$, $E_r \leftarrow 1$, $\mathbf{P}_c \leftarrow \mathbf{P}$ *%Initialization of variables*
  while $i \leq M$ **and** $E_r \geq \epsilon_{\text{tol}}$ **do**
    $\mathbf{P} \leftarrow \mathbf{P}_c$
    compute reflections $R(\mathbf{P})$
    construct diagram $\mathcal{T}(\mathbf{P} \cup R(\mathbf{P}); \mathbb{R}^2)$
    calculate $\mathbf{P}_c \leftarrow \{\mathbf{y}_c : \mathbf{y} \in \mathbf{P}\}$
    compute E_r using Equation (27)
    $i \leftarrow i + 1$
  end while
output: $\tilde{\mathcal{M}}_\Omega(\mathbf{P}) := \left\{V_\mathbf{y} \in \mathcal{T}(\mathbf{P} \cup R(\mathbf{P}); \mathbb{R}^2) : \mathbf{y} \in \mathbf{P}\right\}$

Thus convergence is established when:

$$E_r < \epsilon_{\text{tol}} \tag{26}$$

Here $0 < \epsilon_{\text{tol}} \ll 1$ is the specified convergence tolerance. The pseudo-code for the main routine is shown in Algorithm 3. Note that the mesh is still defined by expression (17) with $R_\Omega(\mathbf{P})$ replaced by $R(\mathbf{P})$.

4.3 Remarks on min/max formulas

We conclude this section with a remark regarding min/-max formulas in (7) and implicitly used in (18) and their influence on the behavior of the meshing algorithm. As mentioned before, these expressions may produce incorrect distance values in certain regions of the plane. A close inspection of the algorithm shows that the *magnitude* of d_Ω generated by (18) is used only in the evaluation of interference criterion (21). Elsewhere—in generating an initial point set, checking condition (19), or computing reflection of a seed—we either use the sign of d_Ω or the distance values of the constituent domains. The requirement on F and the structure of min/max formulas implicitly used in F guarantee that $|d_\Omega(\overline{\mathbf{y}})| = \left|d_{\Omega_j}(\overline{\mathbf{y}})\right|$ for some index j. Here d_Ω denotes the distance function generated by F, which may be different from the exact distance functions associated with Ω in some regions of the plane. In such a case, the violation of condition (21) implies

$$\left|d_{\Omega_j}(\overline{\mathbf{y}})\right| = |d_\Omega(\overline{\mathbf{y}})| \leq \eta \left|d_{\Omega_i}(\mathbf{y})\right| < \left|d_{\Omega_i}(\mathbf{y})\right|$$

for some $j \neq i$ and so the reflection $\overline{\mathbf{y}} = R_{\Omega_i}(\mathbf{y})$ is "correctly" rejected because it is expected to cause interference with approximation of Ω_j. In fact, in this algorithm, we do not need the exact distance function associated with Ω. The need for d_{Ω_i}'s is primarily due to the issue of corners and capturing non-convex features.

5 Matlab implementation

In this section, we explain the details of the Matlab code for polygonal mesh generator. We begin by describing the structure of the code, the input and output parameters and the user-defined `Domain` function that characterizes the meshing domain. Next we make some comments on the routines inside the meshing kernel.

5.1 Structure of the code: meshing kernel and input data

The kernel of the mesh generator is implemented in the `PolyMesher` function. The following variables are the user inputs to this function:

```
[Node,Element,Supp,Load,P]
   = PolyMesher(@Domain,NElem,MaxIter,P)
```

`Domain` This is a Matlab function defining the domain. As shown above, a handle to this function (e.g., `@MichellDomain`) is passed to the kernel, providing access to information about domain geometry (i.e., the sign distance function), the bounding box B (see Section 4.2), and the boundary conditions. The details of this function are discussed later in this section.

`NElem` This is an integer that represents the desired number of elements in the mesh. When an initial point set `P` is specified, `NElem` is replaced by the number of elements in `P`.

`MaxIter` This integer specifies the maximum number of Lloyd's iterations. By setting `MaxIter` to zero and providing an initial point set, the user can obtain/recover the Voronoi mesh produced by that point set.

`P` The user has the option of inputting an initial set of seeds through the array `P`, in which the kth row represents the coordinates of the kth seed. If no such input is passed to the code, an initial random point set is generated (see description of the function `PolyMshr_RndPtSet` below).

The `PolyMesher` function computes and returns the Voronoi discretization along with the final set of seeds and the boundary conditions arrays (`Supp` and `Load` as described below). The mesh is represented by the commonly used data structure in finite element community, namely a node list and an element connectivity matrix. The node list, `Node`, is a two-column array whose ith row represents the coordinates of the node with index number i. The connectivity `Element` is stored inside a Matlab cell of size `NElem`. The kth entry of the cell contains the indices of

Table 1 Various uses and commands for the `Domain` function

Use	Input	Output
Bounding box	`Domain('BdBox')`	Coordinates of bounding box `[xmin,xmax,ymin,ymax]`
Distance values	`Domain('Dist',P)`	$(m+1)\times n$ matrix of distance values for point set $\mathbf{P}$ consisting of n points
Boundary conditions	`Domain('BC',Node)`	Cell consisting of `Load` and `Supp` arrays

the nodes incident on the kth element in counter-clockwise order.[3]

All the domain-related information is included in `Domain` defined outside the kernel. This provides more flexibility for generating meshes for new domains and eliminates the need for the repeated modifications of the kernel. The domain function is used in the kernel for three distinct purposes: (1) retrieving the coordinates of the bounding box for the domain, (2) obtaining the distance of a given point set to the boundary segments, and (3) determining the boundary condition arrays for a given node list. In the sample domain functions, the input string `Demand` specifies the type of information requested. Table 1 summarizes the different input and output choices.

Within the `Domain` function, the user must define the distance function of the meshing domain. This function, called `DistFnc` in the sample domain functions, accepts the coordinates of a point set $\mathbf{P}$ and return a matrix of $(m+1)\times n$ distance values. The entry located at ith column and jth row represents the signed distance value $d_{\Omega_i}(\mathbf{P}_j)$, while the last column is the signed distance for the entire domain Ω. If `d` is a matrix of distance values, this last column can be quickly accessed by command `d(:,end)`.

The user also defines the function that returns the lists of nodal loads and supports given the node list for the mesh. The nodal support array `Supp` has three columns, the first holds the node number, the second and third columns give support conditions for that node in the x- and y-direction, respectively. Value of 0 means that the node is free, and value of 1 specifies a fixed node. The nodal load vector `Load` is structured in a similar way, except for the values in the second and third columns that represent the magnitude of the x- and y-components of the force.

5.2 Comments on the functions in the kernel

We now proceed to discuss some of the details of the implementation of the kernel and comment on its functions. We remark that in this implementation, it is assumed $\mu(\mathbf{x}) \equiv 1$ so $\mathbf{y}_c$ is the centroid of a polygon, $m_{\mathbf{y}} = |V_{\mathbf{y}}|$ and the expression for the error is simplified as:

$$E_r = \frac{n}{|\Omega|^{3/2}} \left(\sum_{\mathbf{y}\in\mathbf{P}} |V_{\mathbf{y}}|^2 \, \|\mathbf{y}-\mathbf{y}_c\|^2 \right)^{1/2} \tag{27}$$

`PolyMesher` The main function follows closely the pseudo-code in Algorithm 3. On line 7, we check to see if an initial point set is provided by the user. The variable `NElem` is updated on line 8 to make sure it is consistent with size of `P`. The iteration counter `It` and error value `Err` are initialized such that the while-loop is executed at least once. Upon the first execution, line 15 recovers the initial points in variable `P`. The area of the domain, needed for computing E_r and α, is initially set to the area of the bounding box on line 11. This value is updated when the centroids of the elements are computed.[4]

Since the output of the Matlab `voronoin` command on line 17 is the set of vertices and connectivity of the entire Voronoi diagram (i.e., with all the cells in $\mathcal{T}(\mathbf{P}\cup R(\mathbf{P});\mathbb{R}^2)$) and the discretization is composed only of the cells in $\mathcal{M}_\Omega$, we need to extract the nodes and connectivity of these cells. This is done in the `PolyMshr_ExtrNds` function (line 24). The mesh is updated once more in `PolyMshr_CllpsEdgs` to remove small edges that can appear in a CVT (line 25). Furthermore, since resulting node numbering and connectivity are random (as a consequence of random placement), the bandwidth of finite element stiffness matrix may be too large. This issue is addressed in the `PolyMshr_RsqsNds` function (line 26). The main function ends with obtaining the boundary condition arrays from the domain function (line 27) and plotting the final mesh (line 28).

`PolyMshr_RndPtSet` This function generates an initial random point set of size `NElem`. A candidate point set `Y` of size `NElem` is generated using the coordinates of the bounding box and `rand` function in Matlab (lines 33–34). Only seeds in `Y` that lie inside the domain are accepted. The variable `Ctr` counts the number of seeds accepted so far. The while-loop terminates when the desired number of seeds is reached.

[3] The cell structure allows for storing vectors of different size and is therefore suitable for connectivity of polygonal elements with different number of nodes.

[4] This small overhead can be removed after a few iterations once a good estimate value is obtained.

`PolyMshr_Rflct` The implementation of this function follows Algorithm 2 in Section 4.2. The gradient of the distance function is computed by means of numerical differentiation:

$$\begin{aligned}\mathbf{n} &:= \nabla d_{\Omega_i}(\mathbf{y}) \\ &\approx \epsilon_d^{-1}\big(d_{\Omega_i}(\mathbf{y}+(\epsilon_d,0)) - d_{\Omega_i}(\mathbf{y}), \\ &\qquad d_{\Omega_i}(\mathbf{y}+(0,\epsilon_d)) - d_{\Omega_i}(\mathbf{y})\big)\end{aligned} \tag{28}$$

Here ϵ_d is a small positive number, set to 10^{-8} by default on line 43. We compute the normal vector for the entire point set at once on lines 46 and 47, where `n1` and `n2` denote the first and second components of the normal, respectively. The logical array index `I`, computed on line 48, is then used to identify and reflect the seeds only about the nearby boundary segments, i.e., those within distance of α. In order to compute the reflections `R_P` at once, the point set `P` is extended, by repeating its columns, to two matrices of the same size as `n1` and `n2` on lines 49 and 50. Once the candidate set of reflections `R_P` of the boundary-adjacent seeds are obtained (lines 51–52), the two conditions for accepting the reflections are enforced on line 55 by means of another logical array index `J`.

`PolyMshr_CntrdPly` This function returns the areas and centroids of the first `NElem` cells in the mesh. Since the density is assumed to be constant, we can compute the area and centroid of each cell using the available formulae for polygons. The signed area for an ℓ-sided polygon is given by:

$$A = \frac{1}{2}\sum_{k=1}^{\ell}\left(v_x^{[k]}v_y^{[k+1]} - v_x^{[k+1]}v_y^{[k]}\right) \tag{29}$$

where $(v_x^{[k]}, v_y^{[k]})$ is the coordinates of the kth vertex of the polygon, $k = 1,\ldots,\ell$. In the above equation, we are defining $(\ell+1)$th vertex to be the same as the first. Similarly, The formula for computing the centroid is:

$$\begin{aligned}c_x &= \frac{1}{6A}\sum_{k=1}^{\ell}\left(v_x^{[k]} + v_x^{[k+1]}\right) \\ &\quad\times\left(v_x^{[k]}v_y^{[k+1]} - v_x^{[k+1]}v_y^{[k]}\right)\end{aligned} \tag{30}$$

$$\begin{aligned}c_y &= \frac{1}{6A}\sum_{k=1}^{\ell}\left(v_y^{[k]} + v_y^{[k+1]}\right) \\ &\quad\times\left(v_x^{[k]}v_y^{[k+1]} - v_x^{[k+1]}v_y^{[k]}\right)\end{aligned} \tag{31}$$

`PolyMshr_ExtrNds` The original discretization is passed to this function through variables `Element0` and `Node0`. The connectivity of the Voronoi cells that make up the mesh are stored in the first n arrays of the cell variable `Element0` as a result of passing the seeds `P` in the first block of the input to Matlab function `voronoin` on line 17. However, the vertices of these cells are not necessarily stored in the first block of rows of `Node0`. The extraction of the additional nodes requires modification of both the node list and the connectivity matrix. The function `PolyMshr_ExtrNds` first creates a one-dimensional array, `map`, containing the indices of the nodes that must remain in the mesh (the Matlab function `unique` is used to remove the appearance of duplicate nodes). The array `cNode`, needed for updating the node list and element connectivity matrix, is constructed on lines 69–70. By setting the entries of `cNode` that correspond to the nodes that must be removed to the maximum value in `map` (which is necessarily the index of a node that will remain in the node list) on line 70, we ensure that they are removed in `PolyMshr_RbldLists` function (see below for the description of this function).

`PolyMshr_CllpsEdgs` This function addresses the issue of appearance of small edges in the centroidal Voronoi

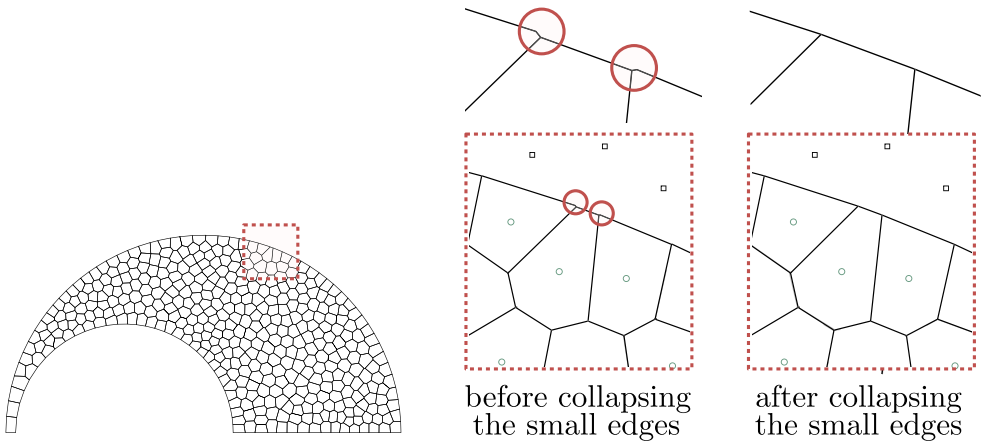


Fig. 10 Small edges can form in elements near a curved boundary since the generating seeds are not the same distance from that boundary. The issue can be addressed by a post-processing step of collapsing these edges onto single nodes

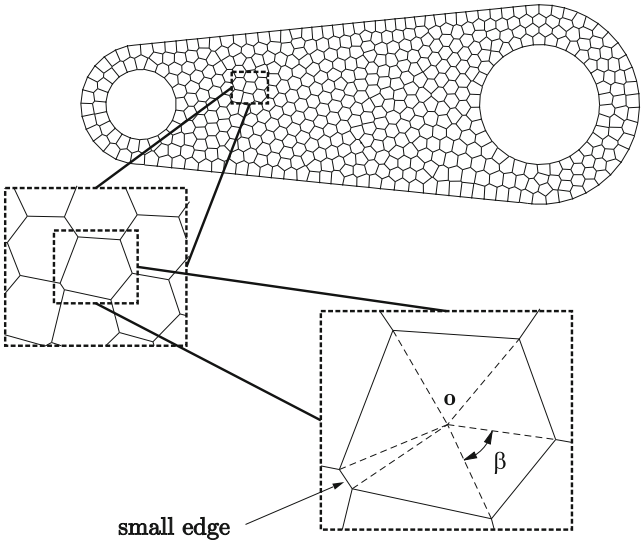


Fig. 11 Illustration of small interior edges in a CVT and definition of angle β in (32)

meshes, which can occur around the boundary of the domain or in the interior. In former case, such phenomenon is due to the use of numerical differentiation to compute the gradient of the distance function and unequal distance of seeds to curved boundaries, as illustrated in Fig. 10. A small interior edge can also form when four seeds are almost co-circular (see Fig. 11). A simple remedy, implemented in `PolyMshr_CllpsEdgs` function, is to collapse the edges that are small compared to the element size into a single node. This operation does not distort the mesh topology (elements remain to be convex polygons) and ensures a uniform quality of the mesh. An alternative penalization approach is discussed in (Sieger et al. 2010). In our implementation, we search all the edges in the mesh by looping over all the elements (for-loop in lines 76–84). Given an edge of an element, we compute the angle between the vectors connecting the center (the location with mean value of the coordinates of the nodes) of the element and the vertices forming the edge (lines 79–80). The edge will be collapsed into a single node when

$$\beta < \epsilon_a \left(\frac{2\pi}{\ell} \right) \tag{32}$$

Here ℓ is the number of vertices of the element and ϵ_a is a user-defined tolerance. To update the node list and element connectivity matrices, the set of edges that needs to be collapsed is stored in array `cEdge` (line 86). The input `cNode` is defined on line 89 such that the two nodes at the end of a tagged edge are replaced by a single node in `PolyMshr_RbldLists`.

`PolyMshr_RsqsNds` The Reverse Cuthill–McKee (RCM) algorithm (Cuthill and McKee 1969) is used to renumber the nodes once more in order to reduce the bandwidth and profile of the stiffness matrix for the mesh. Note that other resequencing algorithms may be used as well, e.g. (Paulino et al. 1994a, b). In the present function, the sparsity pattern of the corresponding stiffness matrix is first computed (lines 98–104). The assembly of this pseudo-stiffness matrix `K` is carried out by the use of Matlab's sparse assembly function (on line 105). The Matlab function `symrcm` returns the reverse Cuthill-McKee ordering, stored in variable `p`, which is then used to compute the `cNode` array passed to `PolyMshr_RbldLists` function. The reader is referred to the Matlab documentation on functions `sparse` and `symrcm` for further information of this implementation.

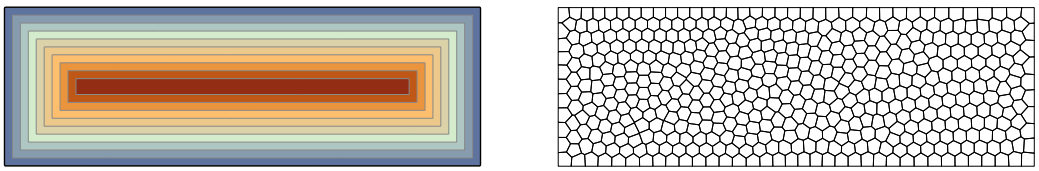
`PolyMshr_RbldLists` This auxiliary function is used in all three mesh modification functions (`PolyMshr_ExtrNds`, `PolyMshr_CllpsEdgs`, and `PolyMshr_RsqsNds`) for updating the node list and element connectivity according to the information contained in the input array `cNode`. This is an array of integer with the same length as the input node list `Node0`. If the ith entry of `cNode` is set to have value k, then after the update, the ith node, `Node0(i)`, is replaced by the coordinates of kth node, `Node0(k)`. For example, in order to collapse nodes `i` and j, we can set `cNode(i)=j` (see line 89 in `PolyMshr_CllpsEdgs` function). Similarly, if we wish to remove the node with index `i` from the node list (which is the case in `PolyMshr_ExtrNds` function), we set `cNode(i)` to be the highest index of the nodes that will remain in the mesh after the modification. On line 112, we use the `unique` function in Matlab to identify the duplicate nodes and the resulting index array `ix` allows us to extract the desired nodal coordinates from `Node0`. When reducing the node list size we need to make sure that the last mapped node is the maximum node of the input map `cNode` (line 113). The connectivity `Element` is updated on line 116 while lines 117–119 guarantee that final arrangement of nodes in the connectivity cell `Element` is ordered counter-clockwise. Note that the `voronoin` does not necessarily store the nodes in the connectivity cell in a counter-clockwise order requiring this correction.

`PolyMshr_PlotMsh` This function plots the mesh and boundary conditions. In order to use the Matlab `patch` function to plot the entire mesh at once, we create an element connectivity matrix `ElemMat` that is padded with `NaNs`. This matrix has `NElem` rows and

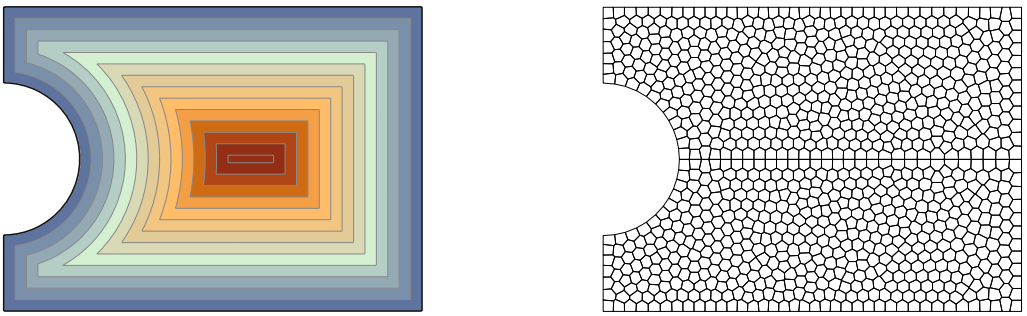
MaxNVer columns (as computed on line 125, MaxNVer is the maximum number of vertices of elements in the mesh). The location of support and loads are also marked provided that Supp and Load arrays are passed to the function.

6 Examples

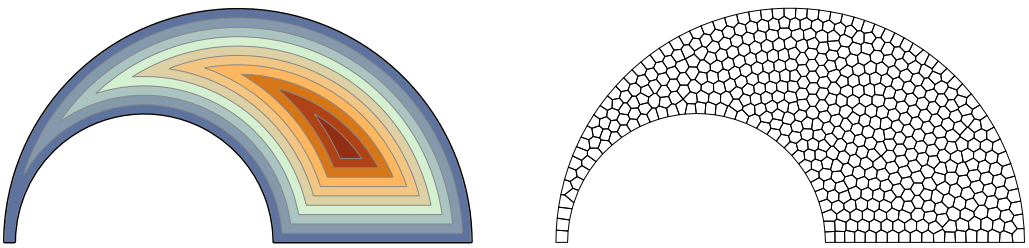
In this section, we will present several examples to illustrate the use of the code. In particular, we show how an appropriate distance function can be constructed using



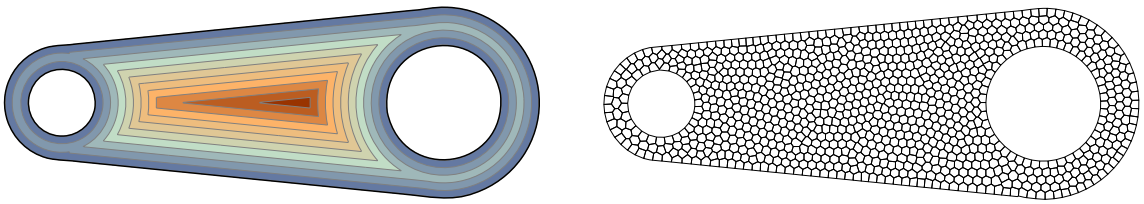
(a) Domain of the MBB beam



(b) Domain of the Michell cantilever problem



(c) Horn geometry



(d) Wrench geometry

Fig. 12 Distance functions (left) and sample meshes (right) for various domains. **a** Domain of the MBB beam; **b** Domain of the Michell cantilever problem; **c** Horn geometry; **d** Wrench geometry

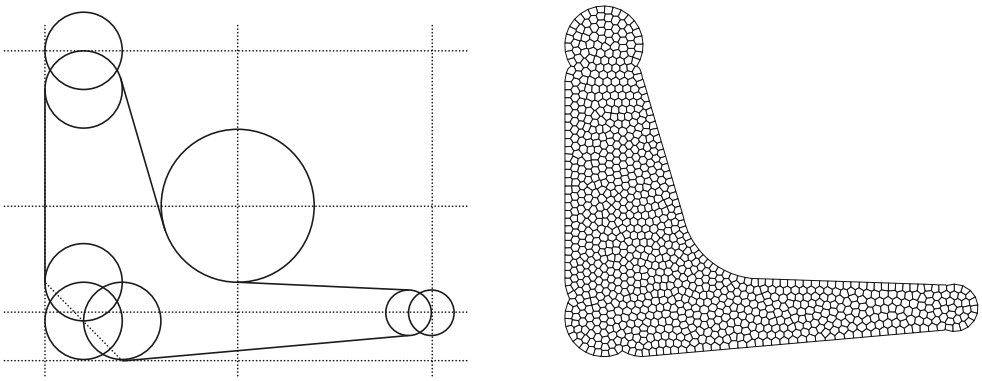


Fig. 13 Suspension triangle **a** geometry and **b** discretization

the library of distance functions. The `Domain` functions for these example geometries and the library of distance functions are provided as supplementary material. Figures 12 and 13 show the contour plot of the distance function and a sample mesh for each example. In all the cases, the default values of $c = 1.5$, $\epsilon_a = 0.1$ and $\epsilon_{\text{tol}} = 5 \times 10^{-3}$ were used.

6.1 MBB beam

The domain is a rectangular box with width of 3 and height of 1, and bottom left corner located at the origin. The distance function is simply given by:

```
d = dRectangle(p,0,3,0,1)
```

and the bounding box is the entire domain, i.e., `BdBox = [0 3 0 1]`. The boundary conditions for the MBB problem are specified by searching the nodes located along the left edge and bottom corner. The following command was used to generate the result in Fig. 12a:

```
[Node,Element,Supp,Load,P]
  = PolyMesher(@MbbDomain,200,100);
```

6.2 Michell cantilever

The second example is the extended domain for the Michell cantilever beam problem. The support is a circular hole and the a vertical load is applied at midspan of the free end. The distance function is constructed as follows:

```
d1 = dRectangle(p,0,5,-2,2)
d2 = dCircle(p,0,0,1)
d  = dDiff(d1,d2)
```

The mesh shown in Fig. 12b consists of 1000 elements, and was constructed to be symmetric about horizontal axis (located at the midspan). The initial point set was restricted

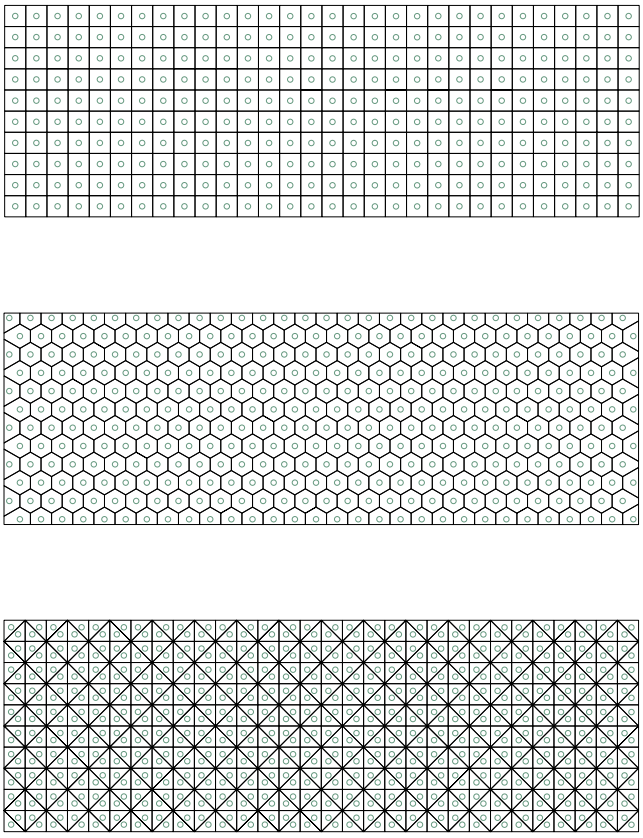


Fig. 14 Uniform discretizations of the MBB beam domain obtained from appropriate placements of the seeds

to the upper half of the domain by modifying line 36 as follows:

```
I = find(d(:,end)<0 & Y(:,2)>0);
```

Also during the iterations that point set was reflected about the x-axis to obtain the complete set of generating seeds. In particular, we replaced `P = Pc` on line 15 by the following line of code:

```
P = [Pc(1:NElem/2,:);[Pc(1:NElem/2,1),
  -Pc(1:NElem/2,2)]];
```

6.3 Horn

The third example has the geometry of a horn, which comes from the difference of two half-circles. The distance function is computed by:

```
d1 = dCircle(p,0,0,1)

d2 = dCircle(p,-0.4,0,0.55)

d3 = dLine(p,0,0,1,0)

d  = dIntersect(d3,dDiff(d1,d2));
```

The mesh in Fig. 12c consists of 500 elements.

6.4 Two more examples

The construction of the distance functions for the last two example involves several geometries and set operations. The first is an extended domain for the design of a socket wrench and the other is domain for a suspension triangle which is presented as an industrial application of topology optimization in Allaire and Jouve (2005). The constituent domains of the suspension triangle are shown in Fig. 13a. The meshes in Figs. 12d and 13b are both made up of 1,000 elements.

6.5 Uniform meshes

The proposed algorithm can also be used to generate certain uniform meshes (regular tessellations), as shown in Fig. 14 for the MBB domain. The user must specify the set of seed `P` and turn off the Lloyd's iterations by setting `MaxIter` to zero. The following set of seeds generates a uniform rectangular mesh with `nelx` elements in the x-direction and `nely` elements in the y-direction for the MBB beam problem:

```
dx = 3/nelx; dy = 1/nely;

[X,Y] = meshgrid(dx/2:dx:3,dy/2:dy:1);

P = [X(:) Y(:)];
```

7 Conclusions and Extensions

The present meshing paper was motivated by the desire to present a complete, self-contained, efficient and useful code in Matlab, including domain description and discretization algorithms. The code is based on the concept of Voronoi diagrams, which offers a simple and effective approach to discretize two-dimensional geometries with convex polygons.

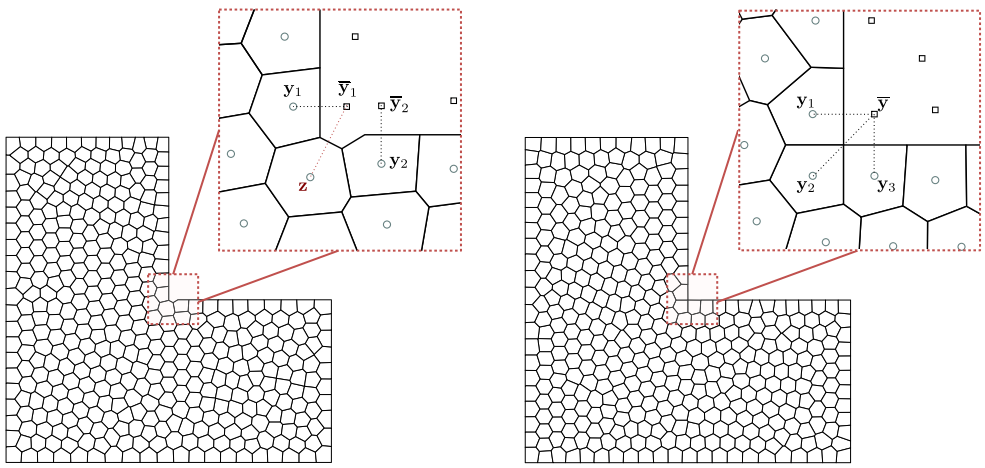
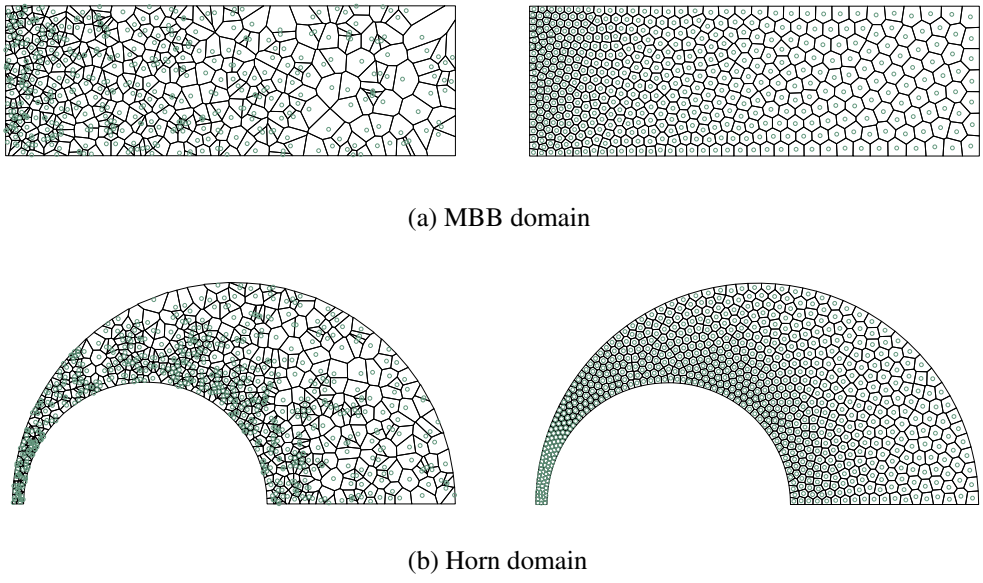


Fig. 15 In the left figure, the reflection of point $\mathbf{y}_1$ has interfered with the approximation of the horizontal boundary by seeds $\mathbf{z}$ and $\mathbf{y}_2$. In the right figure, seeds $\mathbf{y}_1$, $\mathbf{y}_2$ and $\mathbf{y}_3$ are fixed in such way that they all have $\bar{\mathbf{y}}$ as the reflection



(a) MBB domain

(b) Horn domain

Fig. 16 Sample graded meshes: the *left* figures show the initial distribution of seeds generated using a mesh size function and rejection method and the *right* figures show the final mesh after Lloyd's iterations. **a** MBB domain; **b** Horn domain

Its range of applications is broad, including optimization (shape, topology, etc), and other applications (Raghavan et al. 2004).

We conclude this paper by discussing a few extensions of our meshing algorithm, which were left out of the Matlab code for the sake of simplicity and clarity. The first is related to the approximation of certain non-convex corners, which may not be adequately captured with the reflection criteria described above. An example of a domain that exhibits such geometry is the L-shaped domain shown in Fig. 15. Since the seeds near the non-convex corner are placed independently (during the Lloyd's iterations), their reflections may interfere with each other. One possible solution is to place some seeds at an equal distance from the corner and fix them during the Lloyd's iteration. This process can be made systematic by choosing, as the fixed seeds, the reflections of appropriately places seed *outside* such a corner (e.g. point $\overline{\mathbf{y}}$ in Fig. 15).

The second extension is regarding the generation of non-uniform meshes. The Lloyd's algorithm with constant density function, $\mu(\mathbf{x})$, leads to a uniform distribution of seeds and subsequently a Voronoi discretization that is uniform in size over the entire domain. It is possible to generate non-uniform meshes with desirable gradation by selecting an appropriate density function, which biases the placement of points in certain regions (see the effects on varying density function in Du et al. (1999)). Note that, to generalize the code further in this direction, an integration scheme for convex polygons is needed to compute the centroids. Alternatively, we can choose an initial distribution of seeds such that regions that need refinement contain more seeds. Persson and Strang (2004) and Persson (2006) employed a rejection method that relies on defining a mesh size function $h(\mathbf{x})$ over the domain. Figure 16 shows examples of graded meshes that can be generated by the algorithm presented in this work in conjunction with the rejection method. Note that CVT iterations naturally lead to a smooth gradation in the mesh. Although not explored, it is also possible to combine the two approaches and specify suitable and related density and mesh size functions.[5]

We hope that the engineering community will make use of the present mesh generator in ways that we cannot anticipate. Because the source code is provided and explained in detail, the realm of applications of this mesh generator is boundless.

Acknowledgments The first and second authors acknowledge the support by the Department of Energy Computational Science Graduate Fellowship Program of the Office of Science and National Nuclear Security Administration in the Department of Energy under contract DE-FG02-97ER25308. The third and last authors acknowledge the financial support by Tecgraf (Group of Technology in Computer Graphics), PUC-Rio, Rio de Janeiro, Brazil. The authors also acknowledge the insightful comments of the two anonymous reviewers which contributed to improving the manuscript further.

[5] Since the gradation in mesh is often dictated by the geometry of the domain, it is natural that both μ and h be defined based on the distance function d_Ω.

Appendix A: PolyMesher

```
%------------------------------ PolyMesher ------------------------------%
% Ref: C Talischi, GH Paulino, A Pereira, IFM Menezes, "PolyMesher: A     %
%      general-purpose mesh generator for polygonal elements written in   %
%      Matlab," Struct Multidisc Optim, DOI 10.1007/s00158-011-0706-z     %
%------------------------------------------------------------------------%
function [Node,Element,Supp,Load,P] = PolyMesher(Domain,NElem,MaxIter,P)
if ~exist('P','var'), P=PolyMshr_RndPtSet(NElem,Domain); end
NElem = size(P,1);
Tol=5e-3; It=0; Err=1; c=1.5;
BdBox = Domain('BdBox');
Area = (BdBox(2)-BdBox(1))*(BdBox(4)-BdBox(3));
Pc = P; figure;
while(It≤MaxIter && Err>Tol)
  Alpha = c*sqrt(Area/NElem);
  P = Pc;                                          %Lloyd's update
  R_P = PolyMshr_Rflct(P,NElem,Domain,Alpha);      %Generate the reflections
  [Node,Element] = voronoin([P;R_P]);              %Construct Voronoi diagram
  [Pc,A] = PolyMshr_CntrdPly(Element,Node,NElem);
  Area = sum(abs(A));
  Err = sqrt(sum((A.^2).*sum((Pc-P).*(Pc-P),2)))*NElem/Area^1.5;
  fprintf('It: %3d   Error: %1.3e\n',It,Err); It=It+1;
  if NElem≤2000, PolyMshr_PlotMsh(Node,Element,NElem); end;
end
[Node,Element] = PolyMshr_ExtrNds(NElem,Node,Element);  %Extract node list
[Node,Element] = PolyMshr_CllpsEdgs(Node,Element,0.1);  %Remove small edges
[Node,Element] = PolyMshr_RsqsNds(Node,Element);        %Reoder Nodes
BC=Domain('BC',Node); Supp=BC{1}; Load=BC{2};           %Recover BC arrays
PolyMshr_PlotMsh(Node,Element,NElem,Supp,Load);         %Plot mesh and BCs
%------------------------------------------- GENERATE RANDOM POINTSET
function P = PolyMshr_RndPtSet(NElem,Domain)
P=zeros(NElem,2); BdBox=Domain('BdBox'); Ctr=0;
while Ctr<NElem
  Y(:,1) = (BdBox(2)-BdBox(1))*rand(NElem,1)+BdBox(1);
  Y(:,2) = (BdBox(4)-BdBox(3))*rand(NElem,1)+BdBox(3);
  d = Domain('Dist',Y);
  I = find(d(:,end)<0);                  %Index of seeds inside the domain
  NumAdded = min(NElem-Ctr,length(I));  %Number of seeds that can be added
  P(Ctr+1:Ctr+NumAdded,:) = Y(I(1:NumAdded),:);
  Ctr = Ctr+NumAdded;
end
%------------------------------------------------------- REFLECT POINTSET
function R_P = PolyMshr_Rflct(P,NElem,Domain,Alpha)
eps=1e-8; eta=0.9;
d = Domain('Dist',P);
NBdrySegs = size(d,2)-1;          %Number of constituent bdry segments
n1 = (Domain('Dist',P+repmat([eps,0],NElem,1))-d)/eps;
n2 = (Domain('Dist',P+repmat([0,eps],NElem,1))-d)/eps;
I = abs(d(:,1:NBdrySegs))<Alpha;  %Logical index of seeds near the bdry
P1 = repmat(P(:,1),1,NBdrySegs);  %[NElem x NBdrySegs] extension of P(:,1)
P2 = repmat(P(:,2),1,NBdrySegs);  %[NElem x NBdrySegs] extension of P(:,2)
R_P(:,1) = P1(I)-2*n1(I).*d(I);
R_P(:,2) = P2(I)-2*n2(I).*d(I);
d_R_P = Domain('Dist',R_P);
J = abs(d_R_P(:,end))≥eta*abs(d(I)) & d_R_P(:,end)>0;
R_P = R_P(J,:); R_P = unique(R_P,'rows');
%---------------------------------------------- COMPUTE CENTROID OF POLYGON
function [Pc,A] = PolyMshr_CntrdPly(Element,Node,NElem)
Pc=zeros(NElem,2); A=zeros(NElem,1);
for el = 1:NElem
  vx=Node(Element{el},1); vy=Node(Element{el},2); nv=length(Element{el});
  vxS=vx([2:nv 1]); vyS=vy([2:nv 1]); %Shifted vertices
  temp = vx.*vyS - vy.*vxS;
  A(el) = 0.5*sum(temp);
  Pc(el,:) = 1/(6*A(el,1))*[sum((vx+vxS).*temp),sum((vy+vyS).*temp)];
end
%------------------------------------------------------- EXTRACT MESH NODES
function [Node,Element] = PolyMshr_ExtrNds(NElem,Node0,Element0)
map = unique([Element0{1:NElem}]);
cNode = 1:size(Node0,1);
cNode(setdiff(cNode,map)) = max(map);
[Node,Element] = PolyMshr_RbldLists(Node0,Element0(1:NElem),cNode);
%---------------------------------------------------- COLLAPSE SMALL EDGES
```

```
function [Node0,Element0] = PolyMshr_CllpsEdgs(Node0,Element0,Tol)
while(true)
  cEdge = [];
  for el=1:size(Element0,1)
    if size(Element0{el},2)<4, continue; end;  %Cannot collapse triangles
    vx=Node0(Element0{el},1); vy=Node0(Element0{el},2); nv=length(vx);
    beta = atan2(vy-sum(vy)/nv, vx-sum(vx)/nv);
    beta = mod(beta([2:end 1]) -beta,2*pi);
    betaIdeal = 2*pi/size(Element0{el},2);
    Edge = [Element0{el}',Element0{el}([2:end 1])'];
    cEdge = [cEdge; Edge(beta<Tol*betaIdeal,:)];
  end
  if (size(cEdge,1)==0), break; end
  cEdge = unique(sort(cEdge,2),'rows');
  cNode = 1:size(Node0,1);
  for i=1:size(cEdge,1)
    cNode(cEdge(i,2)) = cNode(cEdge(i,1));
  end
  [Node0,Element0] = PolyMshr_RbldLists(Node0,Element0,cNode);
end
%-------------------------------------------------------------- RESEQUENSE NODES
function [Node,Element] = PolyMshr_RsqsNds(Node0,Element0)
NNode0=size(Node0,1); NElem0=size(Element0,1);
ElemLnght=cellfun(@length,Element0); nn=sum(ElemLnght.^2);
i=zeros(nn,1); j=zeros(nn,1); s=zeros(nn,1); index=0;
for el = 1:NElem0
  eNode=Element0{el}; ElemSet=index+1:index+ElemLnght(el)^2;
  i(ElemSet) = kron(eNode,ones(ElemLnght(el),1))';
  j(ElemSet) = kron(eNode,ones(1,ElemLnght(el)))';
  s(ElemSet) = 1;
  index = index+ElemLnght(el)^2;
end
K = sparse(i,j,s,NNode0, NNode0);
p = symrcm(K);
cNode(p(1:NNode0))=1:NNode0;
[Node,Element] = PolyMshr_RbldLists(Node0,Element0,cNode);
%-------------------------------------------------------------- REBUILD LISTS
function [Node,Element] = PolyMshr_RbldLists(Node0,Element0,cNode)
Element = cell(size(Element0,1),1);
[foo,ix,jx] = unique(cNode);
if size(Node0,1)>length(ix), ix(end)=max(cNode); end;
Node = Node0(ix,:);
for el=1:size(Element0,1)
  Element{el} = unique(jx(Element0{el}));
  vx=Node(Element{el},1); vy=Node(Element{el},2); nv=length(vx);
  [foo,iix] = sort(atan2(vy-sum(vy)/nv,vx-sum(vx)/nv));
  Element{el} = Element{el}(iix);
end
%---------------------------------------------------------------- PLOT MESH
function PolyMshr_PlotMsh(Node,Element,NElem,Supp,Load)
clf; axis equal; axis off; hold on;
Element = Element(1:NElem)';                %Only plot the first block
MaxNVer = max(cellfun(@numel,Element));     %Max. num. of vertices in mesh
PadWNaN = @(E) [E NaN(1,MaxNVer-numel(E))]; %Pad cells with NaN
ElemMat = cellfun(PadWNaN,Element,'UniformOutput',false);
ElemMat = vertcat(ElemMat{:});              %Create padded element matrix
patch('Faces',ElemMat,'Vertices',Node,'FaceColor','w'); pause(1e-6)
if exist('Supp','var')&&~isempty(Supp)&&~isempty(Load)%Plot BC if specified
  plot(Node(Supp(:,1),1),Node(Supp(:,1),2),'b>','MarkerSize',8);
  plot(Node(Load(:,1),1),Node(Load(:,1),2),'m^','MarkerSize',8); hold off;
end
%-------------------------------------------------------------------------%
```

Appendix B: Library of distance functions

```
%--------------------------------------------------------------------------%
function d = dRectangle(P,x1,x2,y1,y2)
d = [x1-P(:,1), P(:,1)-x2, y1-P(:,2), P(:,2)-y2];
d = [d,max(d,[],2)];
%--------------------------------------------------------------------------%
function d = dCircle(P,xc,yc,r)
d=sqrt((P(:,1)-xc).^2+(P(:,2)-yc).^2)-r;
d=[d,d];
%--------------------------------------------------------------------------%
function d = dLine(P,x1,y1,x2,y2)
% By convention, a point located at the left hand side of the line
% is inside the region and it is assigned a negative distance value.
a=[x2-x1,y2-y1]; a=a/norm(a);
b=[P(:,1)-x1,P(:,2)-y1];
d=b(:,1)*a(2)-b(:,2)*a(1);
d=[d,d];
%--------------------------------------------------------------------------%
function d = dUnion(d1,d2) % min(d1,d2)
d=[d1(:,1:(end-1)),d2(:,1:(end-1))];
d=[d,min(d1(:,end),d2(:,end))];
%--------------------------------------------------------------------------%
function d = dIntersect(d1,d2) % max(d1,d2)
d=[d1(:,1:(end-1)),d2(:,1:(end-1))];
d=[d,max(d1(:,end),d2(:,end))];
%--------------------------------------------------------------------------%
function d = dDiff(d1,d2) % max(d1,-d2)
d=[d1(:,1:(end-1)),d2(:,1:(end-1))];
d=[d,max(d1(:,end),-d2(:,end))];
%--------------------------------------------------------------------------%
```

References

Allaire G, Jouve F (2005) A level-set method for vibration and multiple loads structural optimization. Comput Methods Appl Mech Eng 194(30–33):3269–3290. doi:10.1016/j.cma.2004.12.018

Allaire G, Pantz O (2006) Structural optimization with FreeFem++. Struct Multidisc Optim 32(3):173–181. doi:10.1007/s00158-006-0017-y

Andreassen E, Clausen A, Schevenels M, Lazarov B, Sigmund O (2011) Efficient topology optimization in MATLAB using 88 lines of code. Struct Multidisc Optim 43(1):1–16. doi:10.1007/s00158-010-0594-7

Aurenhammer F (1991) Voronoi diagrams—a survey of a fundamental geometric data structure. ACM Comput Surv 23(3):345–405

Bolander JE, Saito S (1998) Fracture analyses using spring networks with random geometry. Eng Fract Mech 61(5–6):569–591. doi:10.1016/S0013-7944(98)00069-1

Challis VJ (2010) A discrete level-set topology optimization code written in matlab. Struct Multidisc Optim 41(3):453–464. doi:10.1007/s00158-009-0430-0

Cuthill E, McKee J (1969) Reducing the bandwidth of sparse symmetric matrices. In: Proceedings of the 24th national conference. ACM Press, New York, NY, pp 157–172. doi:10.1145/800195.805928

Du Q, Gunzburger M (2002) Grid generation and optimization based on centroidal Voronoi tessellations. Appl Math Comput 133(2–3):591–607. doi:10.1016/S0096-3003(01)00260-0

Du Q, Wang DS (2005) The optimal centroidal Voronoi tessellations and the Gersho's conjecture in the three-dimensional space. Comput Math Appl 49(9–10):1355–1373

Du Q, Faber V, Gunzburger M (1999) Centroidal Voronoi tessellations: applications and algorithms. Siam Rev 41(4):637–676

Du Q, Gunzburger M, Ju L (2003) Constrained centroidal Voronoi tessellations for surfaces. Siam J Sci Comput 24(5):1488–1506

Du Q, Emelianenko M, Ju LL (2006) Convergence of the Lloyd algorithm for computing centroidal Voronoi tessellations. Siam J Numer Anal 44(1):102–119. doi:10.1137/040617364

Huang Y, Qin H, Wang D (2008) Centroidal Voronoi tessellation-based finite element superconvergence. Int J Numer Methods Eng 76(12):1819–1839. doi:10.1002/nme.2374

Ju L, Gunzburger M, Zhao W (2006) Adaptive finite element methods for elliptic PDEs based on conforming centroidal Voronoi-Delaunay triangulations. Siam J Sci Comput 28(6):2023–2053

Langelaar M (2007) The use of convex uniform honeycomb tessellations in structural topology optimization. In: 7th world congress on structural and multidisciplinary optimization, Seoul, South Korea, 21–25 May 2007

Liu Y, Wang W, Levy B, Sun F, Yan D, Lu L, Yang C (2009) On centroidal Voronoi tesselleations–energy smoothness and fast computation. ACM Trans Graph 28(4):101:1–17

Liu Z, Korvink JG, Huang R (2005) Structure topology optimization: fully coupled level set method via FEMLAB. Struct Multidisc Optim 29:407–417

Osher S, Fedkiw R (2003) Level set methods and dynamic implicit surfaces. Applied mathematical sciences, vol 153. Springer-Verlag, New York

Paulino GH, Menezes IFM, Gattass M, Mukherjee S (1994a) Node and element resequencing using the laplacian of a finite element graph: part I—general concepts and algorithm. Int J Numer Methods Eng 37(9):1511–1530. doi:10.1002/nme.1620370907

Paulino GH, Menezes IFM, Gattass M, Mukherjee S (1994b) Node and element resequencing using the laplacian of a finite element graph: part II—implementation and numerical results. Int J Numer Methods Eng 37(9):1531–1555. doi:10.1002/nme.1620370908

Persson P (2006) Mesh size functions for implicit geometries and PDE-based gradient limiting. Eng Comput 22(2):95–109

Persson P, Strang G (2004) A simple mesh generator in MATLAB. Siam Rev 46(2):329–345

Raghavan P, Li S, Ghosh S (2004) Two scale response and damage modeling of composite materials. Finite Elem Anal Des 40(12):1619–1640

Ricci A (1973) A constructive geometry for computer graphics. Comput J 16(2):157–160

Saxena A (2008) A material-mask overlay strategy for continuum topology optimization of compliant mechanisms using honeycomb discretization. J Mech Des 130(8):082304. doi:10.1115/1.2936891

Sethian JA (1999) Fast marching methods. Siam Rev 41(2):199–235

Sieger D, Alliez P, Botsch M (2010) Optimizing Voronoi diagrams for polygonal finite element computations. In: Proceedings of the 19th international meshing roundtable

Sigmund O (2001) A 99 line topology optimization code written in Matlab. Struct Multidisc Optim 21(2):120–127

Sukumar N, Malsch EA (2006) Recent advances in the construction of polygonal finite element interpolants. Arch Comput Methods Eng 13(1):129–163

Sukumar N, Tabarraei A (2004) Conforming polygonal finite elements. Int J Numer Methods Eng 61(12):2045–2066. doi:10.1002/nme.1141

Suresh K (2010) A 199-line matlab code for Pareto-optimal tracing in topology optimization. Struct Multidisc Optim 42(5):665–679. doi:10.1007/s00158-010-0534-6

Sussman M, Fatemi E, Smereka P, Osher S (1998) An improved level set method for incompressible two-phase flows. Comput Fluids 27(5–6):663–680

Talischi C, Paulino GH, Le CH (2009) Honeycomb Wachspress finite elements for structural topology optimization. Struct Multidisc Optim 37(6):569–583. doi:10.1007/s00158-008-0261-4

Talischi C, Paulino GH, Pereira A, Menezes IFM (2010) Polygonal finite elements for topology optimization: a unifying paradigm. Int J Numer Methods Eng 82(6):671–698. doi:10.1002/nme.2763

Talischi C, Paulino GH, Pereira A, Menezes IFM (2011) PolyTop: a Matlab implementation of a general topology optimization framework using unstructured polygonal finite element meshes. doi:10.1007/s00158-011-0696-x

Yip M, Mohle J, Bolander JE (2005) Automated modeling of three-dimensional structural components using irregular lattices. Comput-aided Civil Infrastruct Eng 20(6):393–407. doi:10.1111/j.1467-8667.2005.00407.x

Zhao H (2004) A fast sweeping method for Eikonal equations. Math Comput 74(250):603–627