A simple technique for unstructured mesh generation via adaptive finite elements

Tom Gustafsson*

November 17, 2020

Abstract

This work describes a concise algorithm for the generation of triangular meshes with the help of standard adaptive finite element methods. We demonstrate that a generic adaptive finite element solver can be repurposed into a triangular mesh generator if a robust mesh smoothing algorithm is applied between the mesh refinement steps. We present an implementation of the mesh generator and demonstrate the resulting meshes via several examples.

1 Introduction

Many numerical methods for partial differential equations (PDE’s), such as the finite element method (FEM) and the finite volume method (FVM), are based on splitting the domain of the solution into primitive shapes such as triangles or tetrahedra. The collection of the primitive shapes, i.e. the computational mesh, is used to define the discretisation, e.g., in the FEM, the shape functions are polynomial in each mesh element, and in the FVM, the discrete fluxes are defined over the edges of the cells.

This article describes a simple approach for the triangulation of two-dimensional polygonal domains. The process can be summarised as follows:

1. Find a constrained Delaunay triangulation (CDT) of the polygonal domain using the corner points as input vertices and the edges as constraints.

2. Solve the Laplace equation with the given triangulation and the FEM.

*Corresponding author. firstname.lastname@aalto.fi
3. Split the triangles with the largest error indicator using adaptive mesh refinement techniques.

4. Apply centroidal patch tessellation (CPT) smoothing to the resulting triangulation.

5. Go to step 2.

It is noteworthy that the steps 2, 3 and 5 correspond exactly to what is done in any implementation of the standard adaptive FEM; cf. Verfürth [22] who calls it the adaptive process.

The goal of this work is to demonstrate that if the mesh smoothing algorithm of step 4 is chosen properly, the adaptive process tends to produce reasonable meshes even if the initial mesh is of low quality. Thus, we demonstrate that the adaptive process—together with an implementation of the CDT and a robust mesh smoothing algorithm—can act as a simple triangular mesh generator.

2 Prior work

Two popular techniques for generating unstructured meshes are those based on the advancing front technique [14] or the Delaunay mesh refinement [8, 18, 20]. In addition, there exists several less known techniques such as quadtree meshing [24], bubble packing [21], and hybrid techniques combining some of the above [15].

Some existing techniques bear similarity to ours. For example, Bossen–Heckbert [4] start with a CDT and improve it by relocating the nodes. Instead of randomly picking nodes for relocation, we use a finite element error indicator that guides the refinement. Instead of doing local modifications, we split simultaneously all triangles that have their error indicators above a predefined threshold.

Persson–Strang [17] describe another technique based on iterative relocation of the nodes. An initial mesh is given by a structured background mesh which is then relaxed by interpreting the edges as a precompressed truss structure. The structure is forced inside a given domain by expressing the boundary using signed distance functions and interpreting the signed distance as an external load acting on the truss. In contrast to the present approach, the geometry description is implicit, i.e. the boundary is defined as the zero set of a user-given distance function.
We do not expect our technique to surpass the existing techniques in the quality of the resulting meshes or in the computational efficiency. However, the algorithm can be easier to understand for those with a background in the finite element method and, hence, it may be a viable candidate for supplementing adaptive finite element solvers with basic mesh generation capabilities.

3 Components of the mesh generator

The input to our mesh generator is a sequence of \( N \) corner points

\[ C = (C_1, C_2, \ldots, C_N), \quad C_j \in \mathbb{R}^2, \quad j = 1, \ldots, N, \]

that form a polygon when connected by the edges

\[(C_1, C_2), (C_2, C_3), \ldots, (C_{N-1}, C_N), (C_N, C_1).\]

We do not allow self-intersecting polygons although the algorithm can be generalised to polygons with holes. The corresponding polygonal domain is denoted by \( \Omega_C \subset \mathbb{R}^2 \).

3.1 Constrained Delaunay triangulation

A triangulation of \( \Omega_C \) is a collection of nonoverlapping nondegenerate triangles whose union is exactly \( \Omega_C \). Our initial triangulation \( \mathcal{T}_0 \) is a constrained Delaunay triangulation (CDT) of the input vertices \( C \) with the edges \((C_1, C_2), (C_2, C_3), \ldots, (C_{N-1}, C_N), (C_N, C_1)\) constrained to be a part of the resulting triangulation and the triangles outside the polygon removed; cf. Chew [7] for the exact definition of a CDT and an algorithm for its construction.

An example initial triangulation of a polygon with a spiral-shaped boundary is given in Figure 1. It is obvious that the CDT is not always a high quality computational mesh due to the presence of arbitrarily small angles. Thus, we seek to improve the initial triangulation by iteratively adding new triangles, and smoothing the mesh. Note that the remaining steps do not assume the use of CDT as an initial triangulation—any triangulation with the prescribed edges will suffice.
Figure 1: A spiral-shaped boundary approximated by linear segments and the corresponding CDT with the triangles outside of the polygon removed. An example from the documentation of the Triangle mesh generator [19].

3.2 Solving the Laplace equation

In order to decide on the placement of the new vertices and triangles, we solve the Laplace equation\(^1\) using the FEM and evaluate the corresponding a posteriori error estimator. The triangles that have the highest values of the error estimator are refined, i.e. split into smaller triangles.

The Laplace equation reads: find \( u : \Omega_C \to \mathbb{R} \) satisfying

\[
-\Delta u = 1 \quad \text{in } \Omega_C, \tag{1}
\]
\[
u = 0 \quad \text{on } \partial \Omega_C. \tag{2}
\]

The finite element method is used to numerically solve the weak formulation: find \( u \in V \) such that

\[
\int_{\Omega_C} \nabla u \cdot \nabla v \, dx = \int_{\Omega_C} v \, dx \quad \forall v \in V, \tag{3}
\]

where \( w \in V \) if \( w|_{\partial \Omega_C} = 0 \) and \( \int_{\Omega_C} (\nabla w)^2 \, dx < \infty. \)

We denote the \( k \)th triangulation of the domain \( \Omega_C \) by \( \mathcal{T}_k \), \( k = 0, 1, \ldots \), and use the piecewise linear polynomial space

\( V_h^k = \{ v \in V : v|_T \in P_1(T) \ \forall T \in \mathcal{T}_k \}, \)

where \( P_1(T) \) denotes the set of linear polynomials over \( T \). The finite element method corresponding to the \( k \)th iteration reads: find \( u_h^k \in V_h^k \) such that

\[
\int_{\Omega_C} \nabla u_h^k \cdot \nabla v_h \, dx = \int_{\Omega_C} v_h \, dx \quad \forall v_h \in V_h^k. \tag{4}
\]

\(^1\)The choice of the Laplace equation is motivated by the following heuristic observation: a quality mesh is often synonymous with a good mesh for the finite element solution of the Laplace equation.
The local a posteriori error estimator reads

\[ \eta_T(u_h^k) = \sqrt{h_T^2 A_T^2 + \frac{1}{2} h_T \int_{\partial T \setminus \partial \Omega_C} (\nabla u_h^k \cdot \mathbf{n})^2 \, ds}, \quad T \in T_k, \quad (5) \]

where \( A_T \) is the area of the triangle \( T \) and \( h_T \) is the length of its longest edge, \([w]_{\partial T \setminus \partial \Omega_C}\) denotes the jump in the values of \( w \) over \( \partial T \setminus \partial \Omega_C \), and \( \mathbf{n} \) is a unit normal vector to \( \partial T \). The error estimator \( \eta_T \) is evaluated for each triangle after solving (4). Finally, a triangle \( T \in T_k \) is marked for refinement if

\[ \eta_T > \theta \max_{T' \in T_k} \eta_{T'}, \quad (6) \]

where \( 0 < \theta < 1 \) is a parameter controlling the amount of elements to split during each iteration. [22]

### 3.3 Red-green-blue refinement

The triangles marked for refinement by the rule (6) are split into four. In order to keep the rest of the triangulation conformal, i.e. to not have nodes in the middle of an edge, the neighboring triangles are split into two or three by the so-called red-green-blue (RGB) refinement; cf. Carstensen [5]. Using RGB refinement to the example of Figure 1 is depicted in Figure 2.

![Figure 2: (Left.) The initial triangulation. (Right.) The resulting triangulation after a solve of (4) and an adaptive RGB refinement.](image)

### 3.4 Centroidal patch triangulation smoothing

We use a mesh smoothing approach introduced by Chen–Holst [6] who refer to the algorithm as centroidal patch triangulation (CPT) smoothing. The idea is to repeatedly move the interior vertices to the area-weighted averages of the barycentres of the surrounding triangles. The CPT smoothing is
combined with an edge flipping algorithm, also described in Chen–Holst [6], to improve the quality of the resulting triangulation. The mesh smoother is applied to the spiral-shaped domain example in Figure 3.

![Figure 3: (Left.) The adaptively refined triangulation. (Right.) The resulting triangulation after smoothing and edge flipping.](image)

4 The mesh generation algorithm

In previous sections we presented an overview of all the components of the mesh generation algorithm. The resulting mesh generator is now summarised in Algorithm 1. The total number of refinements $M$ is a constant to guarantee the termination of the algorithm. Nevertheless, in practice and in our implementation the refinement loop is terminated when a quality criterion is satisfied, e.g., when the average minimum angle of the triangles is above a predefined threshold. The entire mesh generation process for the spiral-shaped domain example is given in Figure 4.

**Algorithm 1** Pseudocode for the triangular mesh generator

**Precondition:** $C$ is a sequence of polygonal domain corner points

**Precondition:** $M$ is the total number of refinements

```plaintext
1: function ADAPT_MESH($C$)
2:     $T_0' \leftarrow CDT(C)$
3:     $T_0 \leftarrow T_0'$ with triangles outside of the boundary removed
4: for $k \leftarrow 1$ to $M$ do
5:     $T_k' \leftarrow RGB(T_{k-1}', \{T \in T_{k-1}\})$
6:     $T_k \leftarrow CPT(T_k')$
7: return $T_N$
```

6
Figure 4: The entire mesh generation process for the spiral-shaped domain example from left-to-right, top-to-bottom.
5 Implementation and examples

We created a prototype of the mesh generator in Python for computational experiments [9]. The implementation relies heavily on the scientific Python ecosystem [23]. It includes source code from pre-existing Python packages tri [3] (CDT implementation, ported from Python 2) and the older MIT-licensed versions of optimesh [2] (CPT smoothing) and meshplex [1] (edge flipping). Moreover, it dynamically imports modules from scikit-fem [11] (RGB refinement) and matplotlib [13] (visualization).

Some example meshes are given in Figure 5. By default our implementation uses the average triangle quality\(^2\) above 0.9 as a stopping criterion which can lead to individual slit triangles. This is visible especially in the last two examples that have small interior angles on the boundary.

\[\text{Figure 5: Some example meshes generated using our implementation of the proposed algorithm. The source code is available in [10].}\]

\(^2\)Triangle quality is defined as two times the ratio of the incircle and circumcircle radii.
6 An example application

Mayzel–Steinberg–Varshney [16] report on measurements using a microfluidic device which is analogous to electron transport in two-dimensional conductive materials such as graphene. The governing equation is the Stokes system

\[
\begin{aligned}
-\nabla \cdot (2\mu \varepsilon(\nabla u)) + \nabla p &= 0, \\
\nabla \cdot u &= 0,
\end{aligned}
\]

where the unknowns are the velocity \( u : \Omega \rightarrow \mathbb{R}^2 \) and the pressure \( p : \Omega \rightarrow \mathbb{R} \), the parameter \( \mu > 0 \) is given, and \( \varepsilon(\nabla u) = \frac{1}{2}(\nabla u + \nabla u^T) \) is the rate-of-strain tensor.

We extract the boundary \( \partial \Omega \) of the computational domain \( \Omega \subset \mathbb{R}^2 \) from a photograph of the experiment and input it to the mesh generator; the photograph of the experiment is given in Figure 6a and the resulting computational mesh in Figure 6b. In order to simplify the problem setup we choose the computational domain so that the left boundary corresponds to the vertical streamlines in the micro-particle image velocimetry; see Figure 6c. This allows using the Dirichlet boundary condition \( u_x = 0, \, u_y = -U, \, U > 0 \), on the left boundary.

We aim to reproduce the location of the swirl and, therefore, the exact parameter values \( U = \mu = 1 \) are irrelevant as they scale only the maximum values of the velocity and the pressure fields, i.e. the shape of the solution will not change. The numerical result, calculated and visualised with the help of the finite volume package FiPy [12] and the plotting library matplotlib [13], is given in Figure 6d.

7 Conclusions

We introduced an algorithm for the generation of triangular meshes for explicit polygonal domains based on the standard adaptive finite element method and centroidal patch triangulation smoothing. We presented a prototype implementation which demonstrates that many of the resulting meshes are reasonable and have an average triangle quality equal to or above 0.9. The majority of the required components are likely to exist in an implementation of the adaptive finite element method. Therefore, the algorithm can be a compelling candidate when an existing adaptive finite element solver is supplemented with basic mesh generation capabilities. Technically the approach extends to three dimensions although in practice the increase in
Figure 6: The experimental images on the left panel are cropped from the figures in Mayzel–Steinberg–Varshney [16]. The original images are licensed under Creative Commons Attribution 4.0 International License (see https://web.archive.org/web/20201113111448/https://creativecommons.org/licenses/by/4.0/).
computational effort can be significant and the quality of the resulting tetrahedralisations has not been investigated.

References

[1] meshplex 0.12.1. https://web.archive.org/web/20201023074645/https://pypi.org/project/meshplex/0.12.1/. Accessed: 2020-10-23.

[2] optimesh 0.6.1. https://web.archive.org/web/20201023080856/https://pypi.org/project/optimesh/0.6.1/. Accessed: 2020-10-23.

[3] tri 0.3. https://web.archive.org/web/20201023101325/https://pypi.org/project/tri/0.3/. Accessed: 2020-10-23.

[4] F. J. Bossen and P. S. Heckbert, A pliant method for anisotropic mesh generation, in 5th International Meshing Roundtable, 1996, pp. 63–74.

[5] C. Carstensen, An adaptive mesh-refining algorithm allowing for an $H^1$ stable $L^2$ projection onto Courant finite element spaces, Constructive Approximation, 20 (2004), pp. 549–564, https://doi.org/10.1007/s00365-003-0550-5.

[6] L. Chen and M. Holst, Efficient mesh optimization schemes based on Optimal Delaunay Triangulations, Computer Methods in Applied Mechanics and Engineering, 200 (2011), pp. 967–984, https://doi.org/10.1016/j.cma.2010.11.007.

[7] L. P. Chew, Constrained Delaunay triangulations, Proceedings of the third annual symposium on computational geometry - SCG ’87, (1987), https://doi.org/10.1145/41958.41981.

[8] L. P. Chew, Guaranteed-quality triangular meshes, (1989), https://doi.org/10.21236/ada210101.

[9] T. Gustafsson, kinnala/adaptmesh v0.1.3, (2020), https://doi.org/10.5281/zenodo.4172393.

[10] T. Gustafsson, kinnala/paper-meshgen v1, (2020), https://doi.org/10.5281/zenodo.4275656.
[11] T. Gustafsson and G. D. McBain, scikit-fem: A python package for finite element assembly, Journal of Open Source Software, 5 (2020), p. 2369, https://doi.org/10.21105/joss.02369.

[12] J. E. Guyer, D. Wheeler, and J. A. Warren, FiPy: Partial differential equations with Python, Computing in science & engineering, 11 (2009), pp. 6–15, https://doi.org/10.1109/MCSE.2009.52.

[13] J. D. Hunter, Matplotlib: A 2d graphics environment, Computing in science & engineering, 9 (2007), pp. 90–95, https://doi.org/10.1109/MCSE.2007.55.

[14] R. Löhner and P. Parikh, Generation of three-dimensional unstructured grids by the advancing-front method, International Journal for Numerical Methods in Fluids, 8 (1988), pp. 1135–1149, https://doi.org/10.1002/fld.1650081003.

[15] D. J. Mavriplis, An advancing front Delaunay triangulation algorithm designed for robustness, Journal of Computational Physics, 117 (1995), pp. 90–101, https://doi.org/10.1006/jcph.1995.1047.

[16] J. Mayzel, V. Steinberg, and A. Varshney, Stokes flow analogous to viscous electron current in graphene, Nature communications, 10 (2019), pp. 1–6, https://doi.org/10.1038/s41467-019-08916-5.

[17] P.-O. Persson and G. Strang, A simple mesh generator in MATLAB, SIAM review, 46 (2004), pp. 329–345, https://doi.org/10.1137/S0036144503429121.

[18] J. Ruppert, A Delaunay refinement algorithm for quality 2-dimensional mesh generation, Journal of Algorithms, 18 (1995), pp. 548–585, https://doi.org/10.1006/jagm.1995.1021.

[19] J. R. Shewchuk, Triangle: Engineering a 2d quality mesh generator and Delaunay triangulator, in Workshop on Applied Computational Geometry, Springer, 1996, pp. 203–222, https://doi.org/10.1007/ BFb0014497.

[20] J. R. Shewchuk, Delaunay refinement algorithms for triangular mesh generation, Computational Geometry, 22 (2002), pp. 21–74, https://doi.org/10.1016/s0925-7721(01)00047-5.
[21] K. Shimada and D. C. Gossard, *Bubble mesh*, Proceedings of the third ACM symposium on solid modeling and applications, (1995), https://doi.org/10.1145/218013.218095.

[22] R. Verfürth, *A Posteriori Error Estimation Techniques for Finite Element Methods*, Oxford University Press, 2013, https://doi.org/10.1093/acprof:oso/9780199679423.001.0001.

[23] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, et al., *SciPy 1.0: fundamental algorithms for scientific computing in Python*, Nature methods, 17 (2020), pp. 261–272, https://doi.org/10.1038/s41592-019-0686-2.

[24] M. Yerry and M. Shephard, *A modified quadtree approach to finite element mesh generation*, IEEE Computer Graphics and Applications, 3 (1983), pp. 39–46, https://doi.org/10.1109/mcg.1983.262997.