CMIS18 - HW3

Isabela Blucher

May 11, 2018

Introduction

When choosing a mesh, we want to go for a type that conforms to the boundaries of the real-world problem being dealt with. For the past two weeks we have been working with regular 2-dimensional grid meshes. While very useful for solving many PDEs and different types of problems, structured meshes have their limitations.

Due to those limitations, we introduce the triangle mesh. A triangle mesh is a type of polygon mesh comprised of triangles connected together to form a surface. These meshes are useful because triangles are easy to conform to smooth and curved surfaces.

Triangle mesh implementation and generators

The implementation of triangle meshes is based on face indexed arrays. Essentially, we deal with two arrays for the x and y coordinates and one matrix T to store the connectivities of the triangles in the grid. Each row of the matrix T represents one triangle, with each column being a vertex of said triangle. As for the vectors, the coordinates (x, y) are stored in the same index position on the two vectors. With this data structure sorted out, it is possible to implement and use triangle mesh generators.

DistMesh

One of the generators introduced this week is the `distmesh2d` function from the DistMesh generator. DistMesh takes as parameters a signed distance function. For the signed distance field generated from the image on Absalon the signed distance function was obtained by interpolating values on the grid. DistMesh also uses the Delaunay triangulation, but tries to optimize the grid node locations through a smoothing process. Figure 1 illustrates the result.

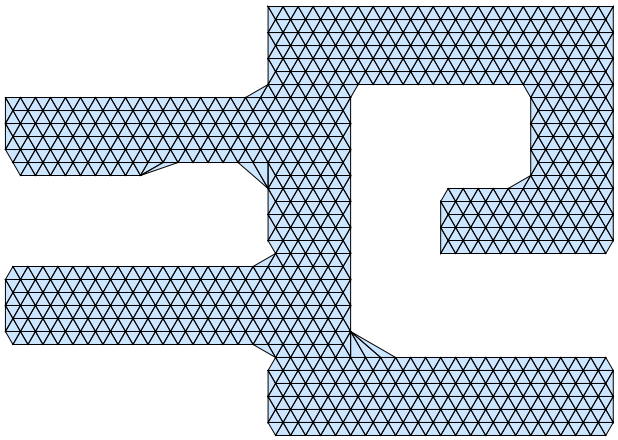
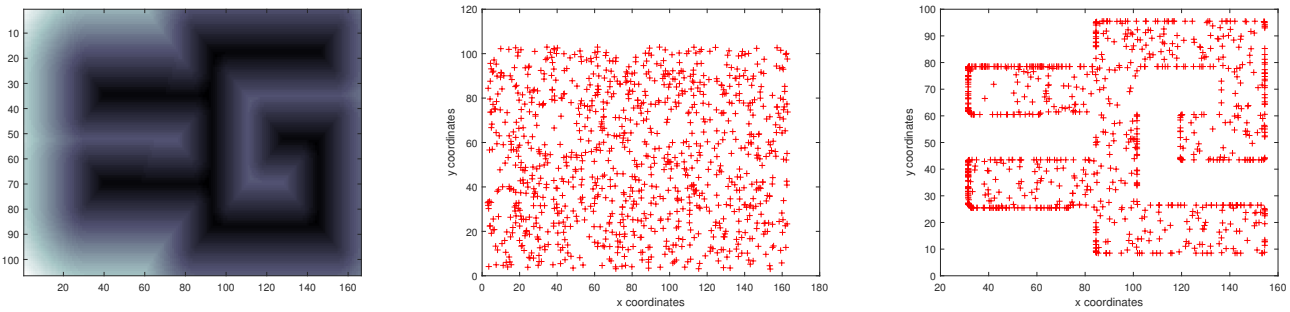


Figure 1: Mesh generated with `distmesh2D`

Delaunay triangulation

For the next two meshes, we use the image on Absalon to generate a signed distance field and use that as a starting point for the mesh generation. How this implementation works is that we first use the random particle generator and then project them to be on the boundary or inside the field. The plots in Figure 2 illustrate that process.



(a) Signed distance field (b) Randomly generated particles (c) Particles projected to field

Figure 2: Projecting particles on a signed distance field

Now that we have the particles and their coordinates, it is possible to use the Matlab built-in `delaunay` function, to generate a triangulation on x and y. By doing so we obtain Figure 3.

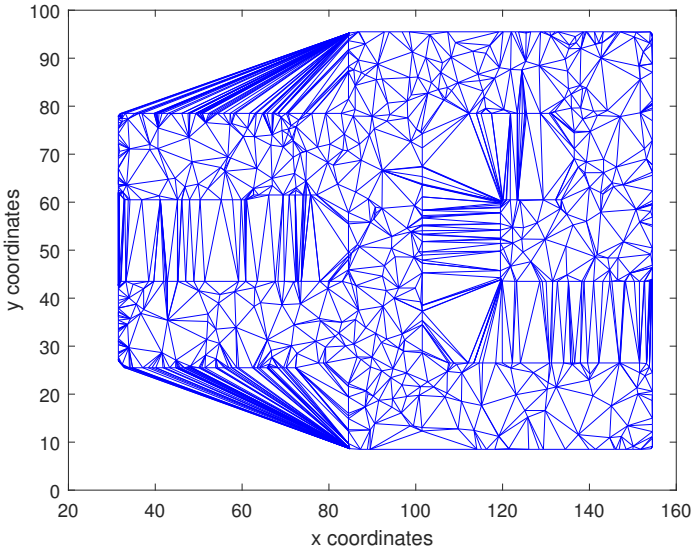


Figure 3: Delaunay triangulation on projected particles coordinates

As we can see here, we still don't have a good enough mesh because we have triangles that are partially outside the signed distance field ϕ. To correct this we generate 10 points inside each triangle and check if all of them are contained in the field. If yes, then we keep the triangle and if not, we remove it from the mesh. After this process we have the triangle mesh in Figure 4.

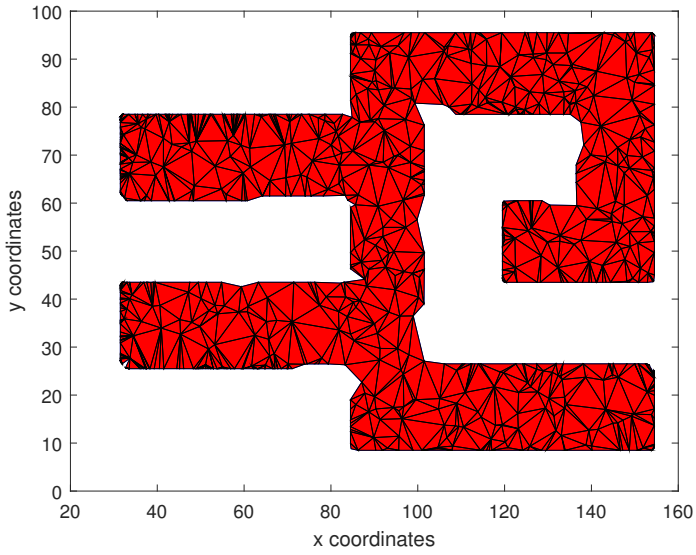


Figure 4: Delaunay triangulation on projected particles coordinates after removing triangles outside field

Marching triangles

During the study group work we implemented the marching triangles algorithm to generate a triangular mesh based on the slides. The essence of the algorithm is that we overlap our signed distance field and the triangular grid and check for each triangle how many of its vertices are inside the field. For each case we have a different outcome, but in general we "crop" the edges of the triangles so the vertices are now inside the field. The implementation's rule was that when finding a vertex outside the field, we'd cut the edge in half and make one or two new triangles, depending on how many vertices were outside of the field. Figure 5 shows the result of running the algorithm on the signed distance field used for the previous meshes.

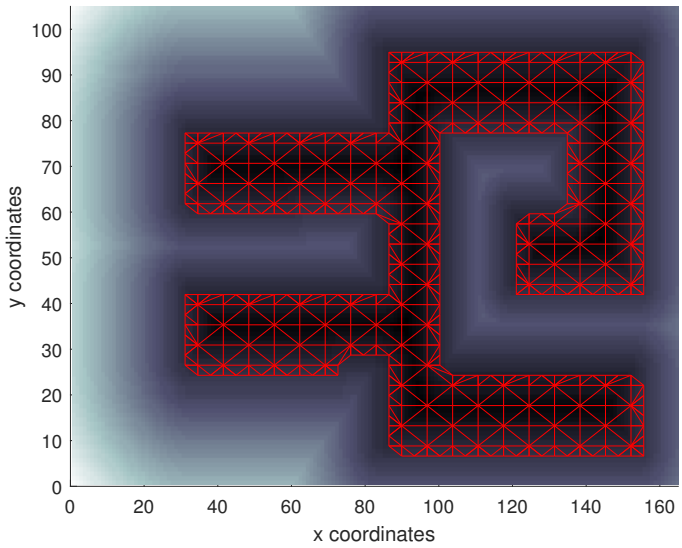


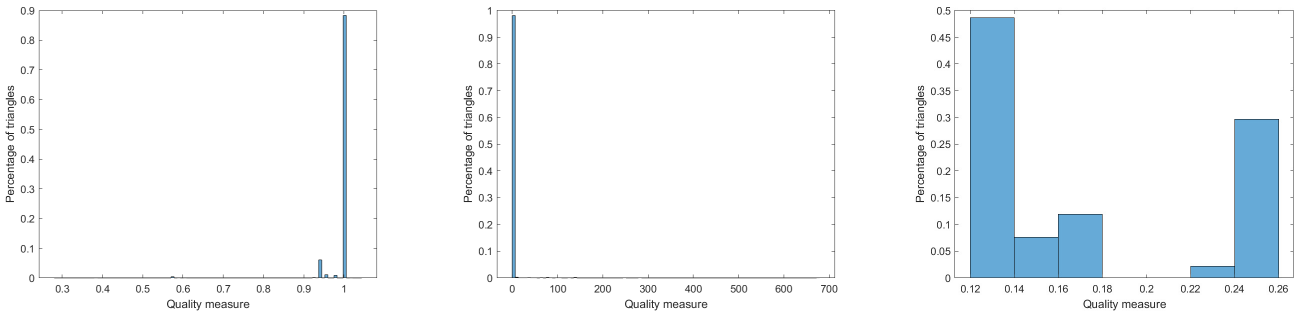
Figure 5: Mesh generated by the marching triangles algorithm on the given signed distance field

From visual inspection it is possible to see in the corners of the mesh the smaller triangles compared to the rest of them. This means that these triangles had vertices outside the field and thus, were rescaled to fit inside.

Quality measures

Having generated all of the meshes mentioned in the sections above, it is necessary to evaluate the quality of them. There are a lot of quality measures, here we'll evaluate two that focus on different properties of the mesh: size and shape. Focusing on size we evaluate how different the sizes of the triangles are, which means we could be penalizing a big range of triangle sizes. Focusing on shape we evaluate how different the triangles shapes are. Since we aren't looking at scale-invariant shapes, we don't penalize small angles as much, and thus can evaluate more detail-oriented meshes.

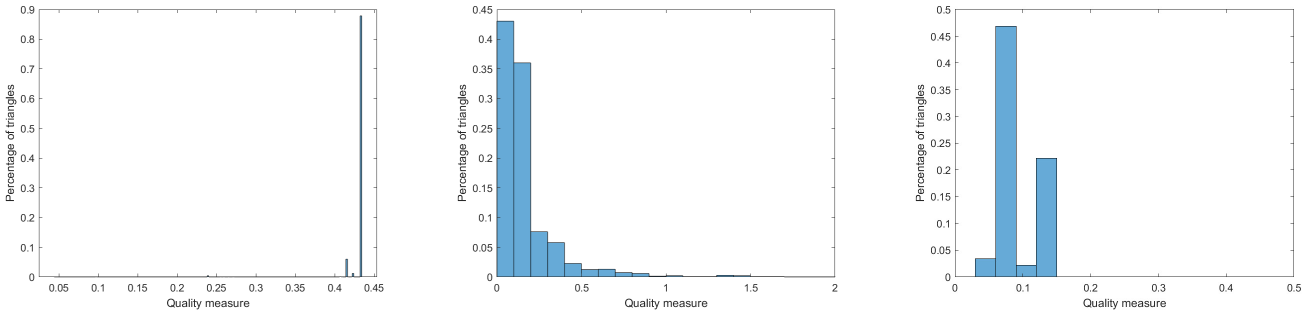
The size quality measure chosen is $QM = \frac{1}{max(l_1,l_2,l_3)}$, with l_1, l_2, l_3 being the sides of the triangle. The following plots illustrate the quality measure histograms for the three meshes previously generated. The histograms could not be scaled properly due to technical difficulties, but we can still see results from Figures 6 and 7.



(a) DistMesh (b) Delaunay (c) Marching Triangles

Figure 6: Size quality measure for previously generated meshes

The shape quality measure chosen is $QM = \frac{A}{l_1 \cdot l_2 \cdot l_3}$, with A being the area of the triangle. The following plots illustrate the quality measures histograms for the three meshes previously generated.



(a) DistMesh (b) Delaunay (c) Marching Triangles

Figure 7: Shape quality measure for previously generated meshes

From looking at the meshes in the previous sections we can see that the one that has been penalized the most with inconsistent measures is the Delaunay one, and we can see that the triangles in Figure 4 can be very different in shape and size.

Overall, the DistMesh generator seems to be the best out of the three, with consistent quality measures covering 90% or more of the triangles in the mesh. Even though the other graphs are badly scaled, we can see that the quality measures aren't very consistent and can vary a lot.

In conclusion, different mesh generators create meshes with differing properties (shape, size and scale of triangles). When choosing a mesh, we have to consider what would be the best application for the problem being dealt with. It is also important after having created the mesh to evaluate its quality, which implies that it is also important to consider the best measure for the chosen mesh.