

CMIS Hand-in 3: Generating computational meshes

Nikolai Plambeck Nielsen

lpk331@alumni.ku.dk

Niels Bohr Institute, University of Copenhagen

1 GENERATING COMPUTATIONAL MESHES

In this hand-in we tackle the problem of generating and evaluating the quality of computational meshes for use with the finite element method. We do this in two ways: One is from scratch, where we generate a mesh from a source image by sampling random points, projecting them and jostling them about, whilst the other is by using the external tool `Triangle` created by J. R. Shewchuk.

2 GENERATING MESHES FROM SCRATCH: PROJECTING AND JOSTLING

The basic algorithm we use for generating meshes from scratch from a source geometry is as follows

- (1) Generate a set of uniformly distributed random vertices on the image
- (2) Push any vertices outside the geometry onto the nearest point on the geometry boundary
- (3) Perform a Delaunay triangulation and discard any triangles not inside the geometry
- (4) Jostle the vertices around to get a more even distribution inside the geometry
- (5) Repeat steps 2 through 4 until satisfied.

To generate the geometry from a source image, we first binarize the image according to some threshold of greyscale values (ie convert the image to greyscale, then all points with a greyscale value above the threshold are converted to white (1), and all others are converted to black (0)). Then we calculate the signed distance field ϕ for this binarized image, giving us the shape boundaries as the contours of $\phi = 0$.

The first step in the algorithm is very uninteresting: we just utilize `numpy`'s built in functions to generate a set of uniformly distributed points. In the second step we make use of the definition of the signed distance field. For each point, we interpolate the value of ϕ and $\nabla\phi$ from the values at the grid positions. To perform the interpolation I made use of the function `RectBivariateSpline` from the `Scipy` package, which generates a 3rd order bivariate spline from a regular mesh. This also allows one to calculate the first and second order derivatives at any interior point on the spline domain.

The interpolated values of ϕ are the euclidean distances to the nearest border of the geometry. The gradient has a magnitude of unity (provided the point is sufficiently far from an inflection point), and corresponds to minus the direction to the nearest boundary. As such, if one scales the gradient by the field, and subtracts this from the points position, the points are projected onto the border of the geometry:

$$\mathbf{r}_{\text{border},i} = \mathbf{r}_i - \phi(\mathbf{r}_i)\nabla\phi(\mathbf{r}_i). \quad (1)$$

The placement of the vertices after the projection step is not quite ideal though. On the first projection, a lot of vertices will now

be placed on the boundary (on average the proportion of border vertices is equal to the ratio between the geometry area and the total image area), and not necessarily evenly distributed. To fix this we need to jostle the vertices around a bit, in hopes of achieving a better distribution.

This is done by utilizing smart Laplacian smoothing, which is a quick way of simulating the position of a point attached to others by infinitely stiff springs. Smart Laplacian smoothing assumes that the equilibrium position of a point \mathbf{r} attached by springs to a set of neighbouring points \mathbf{r}_i will be the centre of mass of the points:

$$\mathbf{R} = \frac{1}{N} \sum_{i \in N(\mathbf{r})} \mathbf{r}_i \quad (2)$$

where N is the number of neighbouring points, and $N(\mathbf{r})$ is the set of points in the neighbourhood of \mathbf{r} . The updating is done in small steps, with 1 step per iteration of the algorithm. The updating is

$$\mathbf{r} \leftarrow \mathbf{r} - \tau(\mathbf{R} - \mathbf{r}), \quad 0 > \tau > 1 \quad (3)$$

The vertices in the “neighbourhood” of \mathbf{p} is chosen as all the vertices that \mathbf{p} shares a triangle with, after the Delaunay triangulation. The triangulation algorithm might create triangles which have some portion of their area outside of the geometry, which we do not want in our mesh.

To alleviate this, and generate a proper neighbourhood for \mathbf{p} we discard any triangle that fall outside the geometry by the following test:

- (1) Generate N points (by default $N = 10$) uniformly distributed within the triangle
- (2) Determine whether each point falls inside or outside of the geometry by interpolating the value of ϕ .
- (3) if **any** generated point falls outside the geometry, discard the triangle

To generate the points inside the triangle, we use the algorithm proposed in <http://www.cs.princeton.edu/~funk/tog02.pdf>, section 4.2, page 8.

The last step in the algorithm is to stop when “satisfied”. The algorithm deterministic, except for the first and third step. The first step is of little importance to convergence, but we hope that the stochasticity introduced by the third step is minimal compared to the rate of convergence for the algorithm, such that the vertices will eventually converge upon some equilibrium.

As such, the easiest way to terminate the algorithm is to just stop after some fixed number of iterations, which is the method we here employ.

Problems with this algorithm occurs, however. We see a “clumping” of vertices after a low number of iterations, leading to neighbourhoods of high vertex density scattered across the geometry (see figure 1 for an example).

To combat this I give artificial mass to the different vertices, when calculating the centre of mass. I employ a linear scale with

the nearest point (\mathbf{p} itself) having a mass of unity, and the farthest point having a mass of m_{\max} . The formula for this is

$$m(r) = \frac{m_{\max} - 1}{r_{\max} - r_{\min}} r + 1 \quad (4)$$

And the centre of mass formula becomes

$$\mathbf{R} = \frac{1}{M} \sum_{i \in N(\mathbf{r})} m_i \mathbf{r}_i \quad (5)$$

where M is the total mass of all vertices in the neighbourhood.

Creating a mesh with the same seed as in figure 2, but with $m_{\max} = 10$, which is still not ideal. In fact, it looks worse. A look at their quality measure histograms also reveals as much - the vast majority of triangles are in the first bin for both quality measures.

I don't know if I will have time for this next refinement, but my next attempt would be to separate the border of the geometry from the interior:

Use vertices generated by the contour function to generate vertices on the border. To control the density of the points we could use a 1D linear interpolation for points along the perimeter - create a linearly spaced vector of values between 0 and 1 (corresponding to the start and end of the perimeter) with the desired number of points, and linearly interpolate the x - and y -coordinates of these new points from the vertices generated by contour.

Then create the desired number of interior points via a simple Monte Carlo simulation: generate a point \mathbf{r} , if $\phi(\mathbf{r}) < 0$ we keep the point (ϕ again calculated with the bivariate spline from before).

In step 4 only update the vertices on the interior of the geometry, but let the neighbourhood include vertices on the border.

3 GENERATING MESHES WITH THE TRIANGLE PACKAGE

For externally generated meshes, we use the Triangle package, created by J. R. Shewchuk (<https://www.cs.cmu.edu/~quake/triangle.html>). To generate meshes we first have to specify the geometry. This we have automated (at least for simple binary) by generating a .poly (see the Triangle homepage for an explanation of the file format) file for Triangle to load.

The process works by generating the border of the geometry from the 0-contour of ϕ , and writing this to a file in the correct format, specifying vertex and segment numbers. To control the density of vertices on the edge of the geometry we perform a linear interpolation as described in the preceding section.

Then we use the command line interface of Triangle, with appropriate arguments to create a .node and .ele file. The first contains a list of vertices, and the second a list of triangles. This can then be imported into python for analysis.

4 MEASURING THE QUALITY OF MESHES

To measure the quality of the generated meshes we implement the calculation of two of the quality measures presented in Shewchuk, 2002. The quality measures are normalized in such a way that a value of 1 correspond to the perfect triangle: an equilateral, whilst a value of 0 corresponds to a degenerate triangle (a line).

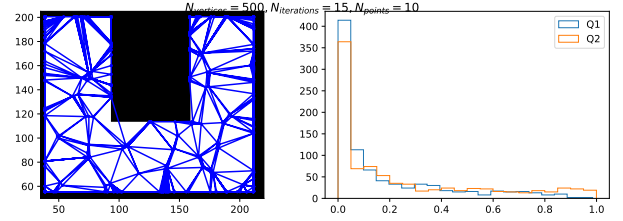


Figure 1: First attempt at generating a mesh. It went poorly

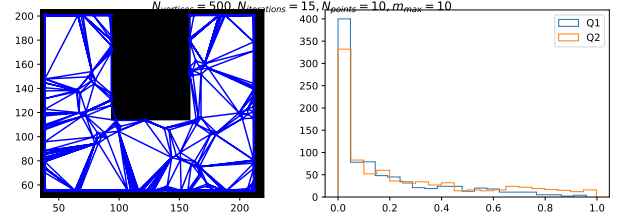


Figure 2: Second attempt at generating a mesh. Looks even worse

In particular we use the minimum angle (in radians) measure and the RMS aspect ratio measure:

$$Q_1 = \frac{3}{\pi} \arcsin\left(\frac{2A}{\ell_{\max}\ell_{\text{med}}}\right), \quad Q_2 = 4\sqrt{3} \frac{A}{\ell_1^2 + \ell_2^2 + \ell_3^2}. \quad (6)$$

These quality measures are calculated for each triangle in the mesh, and the results are analysed to measure the quality. If we have but one triangle with a quality measure value of 0 we discard the mesh as one of low quality, even if several triangles have a quality measure value of 1. In general we want a large spike of quality measure values (for both measures) as close to 1 as possible. To analyse this we plot the quality measures for each mesh in a histogram.

5 EXPERIMENTS

For this weeks experiments we compare the two methods of generating meshes from source images. We test with the two sample images provided in this week and the last, along with an image with several separate geometries.

First we look at the meshes generated from scratch of the sample image from last week. See figure 1 and 2:

Both meshes are terrible. The majority of triangles fall into the first bin of the quality measures. If we instead use the Triangle package we get the result seen in figure 3. If we instead use the sample image from this week, we get the results seen in figures 4 and 5. Again Triangle provides a much better mesh, especially compared to the ones created from scratch.

Lastly we look at an image with 3 separate geometries, with results in figures 6 and 7.

Note the drop in quality for the mesh generated from Triangle. This is due to the chosen parameters (100 points per contour). This fits fine for the larger geometries, but for the small rectangle, it is far too many, leading to a large number of triangles with far too shallow a minimum angle. Expanding the code to be able to

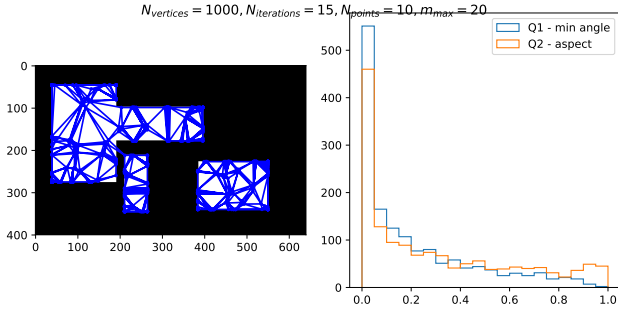


Figure 6: Mesh from scratch, for image with 3 separate geometries. 1000 vertices

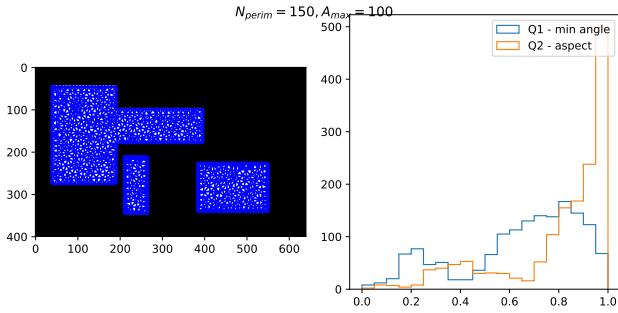


Figure 7: Mesh from Triangle, for image with 3 separate geometries. 100 points per contour, 100 max area

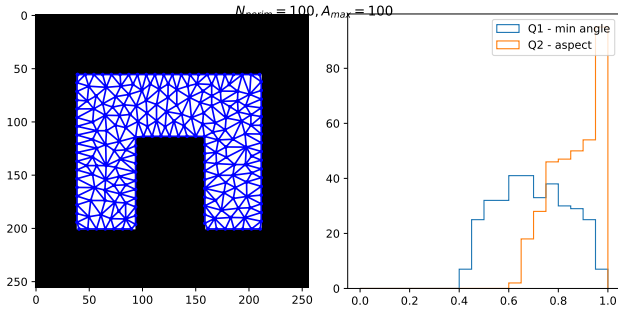


Figure 3: Creating a mesh from the sample image, there are 100 points along the border, and a max triangle area of 100 pixels

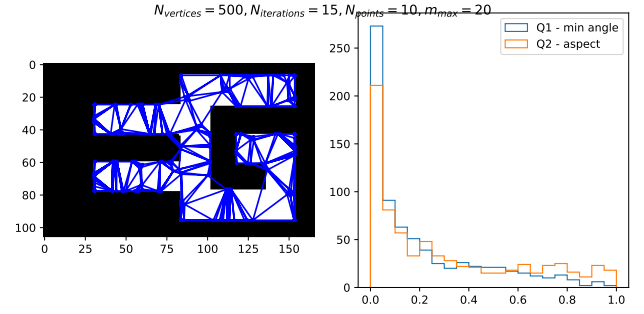


Figure 4: Creating a mesh with Triangle, from this weeks sample image. 200 vertices around the border, max area of 50.

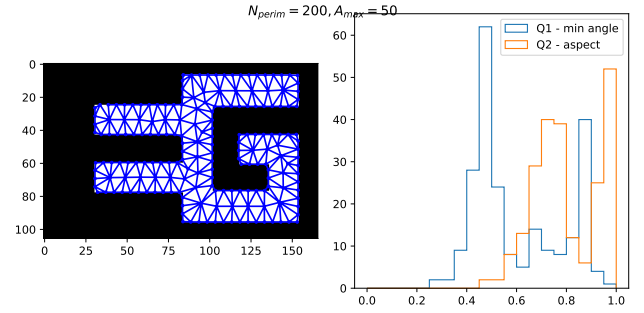


Figure 5: This weeks sample image, mesh created from scratch, 1000 vertices.

handle different numbers of vertices per contour would alleviate the problem.

In conclusion we see that with the current implemented algorithm, the meshes generated from scratch are no where near an acceptable level of quality, whilst the meshes generated from Triangle perform much better.

It is also worth noting the difference in distributions for the two quality measures. The minimum angle measure tends to have a much wider peak, at a lower value than that for the aspect ratio measure.