Adaptive Mesh Generation for Two-Dimensional Simulation of Polygonal Particles in Fluid

Shi Han NG and Hans-Georg MATUTTIS

Department of Mechanical Engineering and Intelligent Systems, The University of Electro-Communications, chofu, Tokyo

Phenomena which concern granular particles and porous media such as landslides and ripples in the sand of windy deserts are strongly connected with the fluid around the granular particles. To analyze these phenomena, a simulation method is necessary which yields both macroscopic and mesoscopic physical quantities of the system. For the combination of the discrete element method for the particles with a finite element method for the fluid of our research group we developed an automatic mesh generator based on a constrained Delaunay triangulation. The meshes in the resulting unstructured grid are equilibrated towards equilateral triangles for improved accuracy. The relaxation is performed by an integrator of zeroth order.

1. INTRODUCTION

To simulate two-phase flow problems of fluid-solid interaction (porous media, suspensions, pneumatic transport etc.) on the "mesoscopic level" with granular particles in fluids, efficient automatic grid generation algorithms are needed. We use a finite element method for the flow simulation,\(^1\) while the granular phase is implemented via a discrete element method with polygonal particles so that the boundaries of the granular particles form the boundaries of the fluid. The triangles of the mesh should be so that we can obtain the highest precision from the finite element method. In this article, we show how to generate a mesh by a constrained Delaunay triangulation and optimize the mesh for better accuracy with a newly developed relaxation algorithm.

2. VORONOI DIAGRAM AND DELAUNAY TRIANGULATION

2.1. AUTOMATIC MESH GENERATION

The first step in the application of the finite element method (FEM) is the mesh generation. A fully automatic mesh generator should be capable of generating valid grids (in the FEM-sense: space-filling and without obtuse angles.) over arbitrary domains only with the information of the specified geometric boundary of the domain and the required distribution of the element size.\(^2, p. 265\) Basically, there are structured meshes (all points in the mesh have the same connectivity) and unstructured meshes. For space-filling tessellations of meshes for the fluid simulation with polygonal particles inside, unstructured triangular meshes are preferable. One of the benefits of incorporating the unstructured meshes in the code is that we will have the maximal geometric flexibility for the simulation of fluids with free surfaces we plan for the future. We will base our approach to unstructured mesh generation on the Delaunay triangulation method.
2.2. VORONOI DIAGRAM

Let \( P = \{p_i, i = 1, 2, \ldots, N\} \) be a set of distinct points in the two-dimensional Euclidean plane \( \mathbb{R}^2 \). In the following, we will refer to them as “forming points” in accordance with Zienkiewicz et al.\(^2\) p.304]. We can define the Voronoi region \( V(p_i) \) of a point \( p_i \) as the set of points \( x \in \mathbb{R}^2 \) closer to \( p_i \) than to any other forming points in \( P \).

\[
V(p_i) = \{x \in \mathbb{R}^2 : \|x - p_i\| \leq \|x - p_j\|, \text{ for all } j \neq i\} \quad (1)
\]

Points that belong to more than one region form the edges of the Voronoi region \( V(p_i) \). The edge of the Voronoi region is perpendicular to the edge of the Delaunay triangle which joins forming points \( p_i \) and \( p_j \) when \( V(p_i) \) and \( V(p_j) \) are contiguous. The Voronoi region is a convex polygonal region and the union of the regions of the point set is called the Voronoi diagram.\(^2\) p.304]

2.3. DELAUNAY TRIANGULATION

The dual graph of a Voronoi diagram is called its Delaunay triangulation. It is constructed by connecting the forming points of neighboring Voronoi regions which share a common edge with straight lines. Fig.1 illustrates the Delaunay triangulation (solid line) and its corresponding Voronoi diagram (dashed line). Delaunay triangulation and Voronoi diagram should have the properties listed below:\(^2\) p.304]

1. To avoid degeneracies, a Delaunay triangle is constructed when only three forming points are co-circular.
2. Each Delaunay triangle corresponds to a Voronoi vertex, which is the center of triangle’s circumcircle (see Fig.1).
3. The interior of a Delaunay triangle’s circumcircle contains no forming point.
4. The boundary of a Delaunay triangulation is the convex hull of the forming points.

In our program, we generate the Delaunay triangles with MATLAB’s built-in corresponding function. Nevertheless, our algorithm should work to post-process any Delaunay triangulation, no matter from which software it is obtained. For a set of forming points defined by vectors \( x \) and \( y \), \( \text{ delaunay}(x,y) \) returns a set of triangles \( \text{ TRI } \) so that no point is contained in any triangle’s circumscribed circle. Each row of the \( m \times 3 \) matrix \( \text{ TRI } \) (where \( m \) is the number of triangles) defines one such triangle and contains indices into \( x \) and \( y \).\(^3\)

2.4. STAGGERED GRID

Fig.2 shows one of the ways to arrange points. When we apply the Delaunay triangulation on this arrangement, we can see that degeneracies cannot be avoided. In Fig.2, each triangle’s circumcircle contains a fourth point which violates the first property described in Section 2.3. To avoid degeneracies in our program, we arrange our points as shown in Fig.3 to get a so called “staggered grid” (trigonal lattice). In Fig.3, we can see that each Delaunay-triangle’s circumcircle
has exactly three points which are co-circular. We also arranged the points so that the Delaunay triangles will be equilateral.

3. CONSTRAINED DELAUNAY TRIANGULATION

3.1. INTRODUCTION

In our simulation of granular particles, we want to create a mesh only in the fluid space, no line should pass through a particle. The idea was, that if we generated a staggered grid in the way mentioned in Section 2.4, then we create granular particles on the grid and use the vertices of the particles as additional forming points. We remove points of the staggered grid close to or inside the particles and perform the Delaunay triangulation, see Fig. 4. To eliminate the meshes which are inside the particles, we assigned an “owner number” to each point. The points on particles’ outlines (which are also the vertices of the particles) will receive the owner number of each particle \((1, 2, 3, \ldots, n)\) which they belong to \((n\) is total number of particles). The points on the boundaries will receive the owner number of \(n + 1\) for the bottom boundary, \(n + 2\) for the right boundary, \(n + 3\) for the top boundary, and \(n + 4\) for the left boundary. Finally, for points which are not on any particles’ outlines or boundaries, we assign the owner number of \(n + 5\). We use \(n + 5\) instead of e.g. zero or negative numbers, because it will be easier to detect the points of the staggered grid (which also include the points on the boundaries) by determining if the owner number is larger than the number of particles \(n\) in our program. To remove the Delaunay triangles which were created inside the particles, we scan through the triangles and search for triangles which contain all three points with the same owner number while the owner number must be equal or smaller than \(n\). Removing these triangles leads to the result in Fig. 5. Those triangles which can not be removed are not fully inside a particle. All triangles which the program was unable to remove are overlapping with particle edges. If we would force the program to remove these triangles, there would be some empty space and such a triangulation which is not space-filling is not acceptable for the finite element method. So, we need to modify the triangular mesh so that no edges intersect the particles’ edges. This modification will result in an effective “constrained Delaunay triangulation”.

3.2. SINGLE CONSTRAINT

3.2.1. CHECK THE INTERSECTION

Fig. 6 shows a set of points arranged as staggered grid. Inside these points, we introduced a line segment between two extra points. which we treat in the following as a constraint through which no
grid edge should cut. As the result of the Delaunay triangulation for these points, we can see that some triangles are overlapping with the constraint. We can detect these triangles with Algorithm 1, for which we explain here the necessary data structures. Each row in \( \text{Tri} \text{Connect} \) and \( \text{Tri} \text{NotConnect} \) contains three indices for the \( x \) and \( y \)-coordinates of a triangle's vertices. This data structure is similar to the tri in Section 2.3 which is produced by MATLAB's Delaunay triangulation. We construct triangles which have only one segment intersecting with the constraint into \( \text{Tri} \text{Connect} \), and triangles which have two segments intersecting with the constraint into \( \text{Tri} \text{NotConnect} \). We also arrange the indices inside \( \text{Tri} \text{Connect} \) and \( \text{Tri} \text{NotConnect} \) so that the data structure will be:

1. For array \( \text{Tri} \text{Connect} = \{ P_1, P_2, P_3 \} \), \( P_1 \) is the point on the constraint. \( P_2 \) and \( P_3 \) span the segment which intersects the constraint.
2. For array \( \text{Tri} \text{NotConnect} = \{ P_1, P_2, P_3 \} \), \( P_1 \) and \( P_3 \) span the segment which intersects the constraint. \( P_2 \) and \( P_3 \) span the segment which intersects the constraint.

The intersection a triangle's segment and the constraint is calculated by the method from O'Rourke.\(^3\) In Fig. 7, let \( A = [A_x, A_y] \), \( B = [B_x, B_y] \) be two-dimensional position vectors which span the line segment of a triangle, and \( C = [C_x, C_y] \), \( D = [D_x, D_y] \) be position vectors which span the constraint:

\[
\begin{align*}
    r &= \frac{(A_y - C_y)(D_x - C_x) - (A_x - C_x)(D_y - C_y)}{(B_x - A_x)(D_y - C_y) - (B_y - A_y)(D_x - C_x)}, \\
    s &= \frac{(A_x - C_x)(B_y - A_y) - (A_y - C_y)(B_x - A_x)}{(B_x - A_x)(D_y - C_y) - (B_y - A_y)(D_x - C_x)}.
\end{align*}
\]

(2)

The segment and the constraint are:
1. intersecting if \( (0 \leq r \leq 1) \) and \( (0 \leq s \leq 1) \),
2. parallel if denominator in eq. (2) is zero,
3. coincident if numerator in eq. (2) is zero.

However, case 3 will not occur in our algorithm because we only check the intersection if the segment does not coincide with the constraint. Using Fig. 6 as an example, from Algorithm 1 we get the value of \( \text{Tri} \text{Connect} \) and \( \text{Tri} \text{NotConnect} \) as

\[
\begin{bmatrix}
    28 & 5 & 9 \\
    29 & 13 & 10
\end{bmatrix}, \quad
\begin{bmatrix}
    13 & 9 & 10 \\
    6 & 5 & 9 \\
    6 & 10 & 9
\end{bmatrix}.
\]

(4)

Note that if an overlap occurs between a triangular mesh and a constraint, there must be two triangles which are connected with the constraint's endpoints. In other words, if we detect an overlap and need to modify the mesh on a constraint, the number of rows in \( \text{Tri} \text{Connect} \) must be two. There is also a possibility that array \( \text{Tri} \text{NotConnect} \) is empty.

3.2.2. SEPARATE OVERLAPPED REGION

From Algorithm 1, we get a set of triangles in \( \text{Tri} \text{Connect} \) and \( \text{Tri} \text{NotConnect} \) as,

\[
\begin{align*}
    \text{Tri} \text{Connect} = \begin{bmatrix}
        T_{c11} & T_{c12} & T_{c13} \\
        T_{c21} & T_{c22} & T_{c23}
    \end{bmatrix}, \quad
    \text{Tri} \text{NotConnect} = \begin{bmatrix}
        T_{n11} & T_{n12} & T_{n13} \\
        T_{n21} & T_{n22} & T_{n23} \\
        \vdots & \vdots & \vdots \\
        T_{nm1} & T_{nm2} & T_{nm3}
    \end{bmatrix},
\end{align*}
\]

(5)

where \( m \) is the number of triangles inside the array. In Fig. 8 we can see that these triangles form a non-convex polygonal region. Before we retriangulate the region, we will use Algorithm 2 to separate the region into two parts while keeping the shape of the non-convex polygonal region unchanged.
Algorithm 1 Detect triangles which intersect with a constraint.

for all Delaunay triangles ([P₁, P₂, P₃]) do
    intersectionₙ ← 0
    Reset array Triₐ
    Reset array Triₖ
    if (P₂ is not on the constraint) and (P₃ is not on the constraint) then
        if Segment A intersects with the constraint then
            intersectionₙ ← intersectionₙ + 1
            Store index of P₁ into Triₐ
        else
            Store index of P₁ into Triₖ
        end if
    else
        Store index of P₂ into Triₖ
    end if
    if (P₁ is not on the constraint) and (P₃ is not on the constraint) then
        if Segment B intersects with the constraint then
            intersectionₙ ← intersectionₙ + 1
            Store index of P₂ into Triₐ
        else
            Store index of P₂ into Triₖ
        end if
    else
        Store index of P₃ into Triₖ
    end if
    if (P₂ is not on the constraint) and (P₁ is not on the constraint) then
        if Segment C intersects with the constraint then
            intersectionₙ ← intersectionₙ + 1
            Store index of P₃ into Triₐ
        else
            Store index of P₃ into Triₖ
        end if
    else
        Store index of P₃ into Triₖ
    end if
    if intersectionₙ = 1 then
        Store [Triₐ, Triₖ] into TriConnect
    else if intersectionₙ = 2 then
        Store [Triₐ, Triₖ] into TriNotConnect
    end if
end for

In Algorithm 2, \( p(i) \) is the position vector of the triangle's vertex \( i \). In our example with Fig. 6 the value of Region₁ and Region₂ will be: Region₁ = [28, 9, 13, 29] and Region₂ = [28, 5, 6, 10, 29].

Fig. 8 Separation of the overlapped region into two non-convex regions.
Algorithm 2 Separate the intersecting region into two non-convex polygonal shapes.

Add $Tc_{11}$ into Array $Region_A$
Add $Tc_{12}$ into Array $Region_B$
Vector$_{constraint} = p(Tc_{21}) - p(Tc_{11})$
Vector$_{segment} = p(Tc_{12}) - p(Tc_{11})$
If Vector$_{constraint}$ × Vector$_{segment} > 0$ then
  Add $Tc_{12}$ into $Region_A$
  Add $Tc_{13}$ into $Region_B$
Else
  Add $Tc_{12}$ into $Region_B$
  Add $Tc_{13}$ into $Region_A$
End if
While Array $Tri_{notConnect}$ is not empty do
  For $i = 0$ to $m$ do
    If segment $[Tn_{i1}, Tn_{i2}] = line [Region_A(end), Region_B(end)]$ then
      Vector$_{segment} = p(Tn_{i2}) - p(Tc_{11})$
      If Vector$_{constraint}$ × Vector$_{segment} > 0$ then
        Add $Tn_{i2}$ into $Region_B$
      Else
        Add $Tn_{i2}$ into $Region_B$
      End if
    Remove row $i$ from Array $Tri_{notConnect}$
    Break
  Else if segment $[Tn_{i2}, Tn_{i3}] = line [Region_A(end), Region_B(end)]$ then
    Vector$_{segment} = p(Tn_{i2}) - p(Tc_{11})$
    If Vector$_{constraint}$ × Vector$_{segment} > 0$ then
      Add $Tn_{i1}$ into $Region_A$
    Else
      Add $Tn_{i1}$ into $Region_B$
    End if
    Remove row $i$ from Array $Tri_{notConnect}$
    Break
  End if
End for
End while
Add $Tc_{21}$ into Array $Region_A$
Add $Tc_{21}$ into Array $Region_B$

3.2.3. RETRIANGULATION

After separating the intersecting region into two parts, we retriangulate each part. Since either part might be non-convex, we cannot use the Delaunay triangulation because it will create edges outside of the region. Therefore we retriangulate $Region_A$ with Algorithm 3. Retriangulation of $Region_B$ can be done by changing $(V_A \times V_B < 0)$ to $(V_A \times V_B > 0)$ in Algorithm 3. In the example with Fig. 6, the value of $NewTri_A$ and $NewTri_B$ will be:

$$NewTri_A = \begin{bmatrix} 9 & 13 & 29 \\ 28 & 9 & 29 \end{bmatrix}, \quad NewTri_B = \begin{bmatrix} 28 & 5 & 6 \\ 6 & 10 & 29 \\ 28 & 6 & 29 \end{bmatrix}.$$
Algorithm 3 Retriangulation of Region$_A$.

\begin{algorithm}
\textbf{while} (Number of indices in Region$_A$) $\geq 3$ \textbf{do}
\begin{itemize}
  \item Reset array NewRegion$_A$
  \item $i = 1$
  \item add Region$_{A,1}$ into NewRegion$_A$
  \item \textbf{while} $i \neq$ (Number of indices in Region$_A$) \textbf{do}
  \begin{itemize}
    \item if $i = \text{(Number of indices in Region$_A$)} - 1$ then
      \begin{itemize}
        \item add Region$_{A,i+1}$ into NewRegion$_A$
      \end{itemize}
    \item break
  \end{itemize}
  \item $V_A = p(\text{Region}_{A,i+1}) - p(\text{Region}_{A,i})$
  \item $V_B = p(\text{Region}_{A,i+2}) - p(\text{Region}_{A,i+1})$
  \item if $V_A \times V_B < 0$ then
    \begin{itemize}
      \item add [Region$_{A,i}$, Region$_{A,i+1}$, Region$_{A,i+2}$] into NewTri$_A$
    \end{itemize}
  \item add Region$_{A,i+2}$ into NewRegion$_A$
  \item $i = i + 2$
  \item else
    \begin{itemize}
      \item add Region$_{A,i+1}$ into NewRegion$_A$
      \item $i = i + 1$
    \end{itemize}
  \item end if
\end{itemize}
\item Region$_A = \text{NewRegion}_A$
\item end while
\end{algorithm}

Fig. 9 Result of constrained Delaunay triangulation of a staggered grid with a constraint line, (a) before triangulation and (b) after triangulation.

Triangles which overlap with the constraint are detected in Algorithm 1. Now we will remove them from the list of the original Delaunay triangles. Then we add NewTri$_A$ and NewTri$_B$ into that list. This results in a constrained Delaunay triangulation like in Fig. 9. In Fig. 9(b) we can see that inside a constrained Delaunay triangle's circumcircle other triangles' vertices are contained. This is not possible in a rigorous Delaunay triangulation. Therefore the “quality” of the constrained Delaunay triangles is certainly not as “good” as that of a rigorous Delaunay triangulation. To improve the “quality” of the mesh, we will introduce a relaxation algorithm in Section 4.4.

3.3. APPLYING THE CONSTRAINED DELAUNAY TRIANGULATION ON THE SIMULATION

In our simulation, we treat each edge of the polygonal particles as one constraint. In Fig. 4 we applied the Delaunay triangulation on the simulation geometry, and obtained a set of triangles. For each constraint, we scan all triangles and check if any overlaps occur using Algorithm 1. Then we apply Algorithm 2 and 3 to generate constrained Delaunay triangles. After applying the constrained
Delaunay triangulation to the example in Fig. 4, we search for triangles which contain all three forming points with the same owner number equal or smaller than the total number of particles \( n \). In Fig. 10 (gray line), we successfully removed every triangle inside the particles. We also applied the constrained Delaunay triangulation with more particles and the algorithm also worked here, as shown in Fig. 11.

4. RELAXATION ALGORITHM

In Section 3 we successfully generated a constrained Delaunay mesh for the simulation of fluids in which granular particles are suspended. As we can see from Fig. 10 (gray line), the triangular mesh generated around the polygonal particle is not satisfying as it contains degenerate triangles. For the finite element method, triangular meshes should be as close to equilateral as possible and certainly not degenerate. A large angle in a triangle gives a bad approximation of the derivatives and thus a large error, so as a rule of thumb, all angles should be smaller that 135°. In this section, we are going to introduce a method to systematically improve the quality of our triangular mesh with a "relaxation algorithm" where we treat the sides of a triangular mesh as springs and relax them towards equilateral shape.

4.1. ZERO-TH-ORDER APPROXIMATION

We start from the second-order Verlet method,\(^6\) which is convenient to obtain the position vector \( \mathbf{r}(t) \) from position dependant accelerations \( \mathbf{a} \) as

\[
\mathbf{r}_{n+1} = 2\mathbf{r}_n - \mathbf{r}_{n-1} + \tau^2\mathbf{a}_n + O(\tau^4),
\]

\[
= \mathbf{r}_n - \frac{\mathbf{r}_{n-1} - \mathbf{r}_n}{\tau} \tau + \tau^2\mathbf{a}_n + O(\tau^4),
\]

where in eq. (8) a combination of previous positions can be interpreted as the velocity vectors \( \mathbf{v}(t) \). Approximating \( \mathbf{r}_{n-1} = \mathbf{r}_n \) once, (a zeroth-order approximation with a first-order truncation error) in eq. (8), yields a zeroth-order time integrator. This will eliminate the \( \mathbf{v}(t) \) term in eq (8) by introducing an error of first order in the Verlet-method eq. (7) and leads to

\[
\mathbf{r}_{n+1} = \mathbf{r}_n + \tau^2\mathbf{a}_n + O(\tau).
\]

We show how this method will lead to convergence of the particles' positions towards its equilibrium position \( x_0 \) for one dimension. In that case if a particle is at position \( x_n = x_0 + \rho \) in a potential

\[
\Psi(\rho) = \psi_0 + \psi_2\rho^2 \ldots
\]

For \( \psi_2 > 0 \) and the equilibrium at \( x_0 \), with the time-integration according to eq. (9), the particle position will converge towards \( x_0 \). The acceleration of the particle (for unit mass) is

\[
a(\rho) = -\nabla\Psi(\rho) = -2\psi_2\rho + \ldots
\]
Then, the new position is
\[
x_{n+1} = x_0 + \rho - 2\psi_2 \rho \tau^2 + \ldots
\]
\[
= x_0 + \rho \left(1 - 2\psi_2 \tau^2\right),
\]
so that the new position \(x_{n+1}\) is closer to \(x_0\) than the old position \(x_n = x_0 + \rho\) for sufficiently small timesteps \(\tau\) (\(0 < \tau < 1/\sqrt{\psi_2}\)), because there first-order (velocity-) term in eq. (8) has been eliminated. Only for the force equilibrium at \(\rho = 0\), no force acts on the particle and the particle will remain there. The argument is still valid for time-dependent equilibria with \(x_0(t), \rho(t), \Psi(t)\ldots\) and higher dimensions. This method will relax the initial value from its initial state to a stationary state.

4.2. RELAXATION IN THE ONE-DIMENSIONAL CASE

We show now how the relaxation performs in the one dimensional case for a set of points spaced randomly at \(x_1, x_2, \ldots, x_m\) like in Fig. 12. To obtain equidistant points, we introduce linear springs with spring constant \(k\). The force \(F_i\) between neighboring points depends on the average distance between the points, equal to (system length)/(number of points-1)
\[
\bar{x} = \frac{\sum_{i=1}^{m-1} (x_{i+1} - x_i)}{m}.
\]

Then one obtains Hooke’s law for a linear chain
\[
F_i = k \left( (x_{i+1} - x_i) - \bar{x} - [(x_i - x_{i-1}) - \bar{x}] \right),
\]
which reduces to
\[
\bar{x}_i = \frac{k}{m} (x_{i+1} + x_{i-1} - 2x_i).
\]

We can see from this equation that point \(x_i\) will experience an attracting force from the neighboring points if the distance between them is larger than \(\bar{x}\), and a repulsive force if the distance is smaller than \(\bar{x}\). With the zeroth-order method from eq. (9) we obtain the result in Fig. 13 for initially randomly spaced points for \(k = m = 1\). The algorithm successfully equilibrates the points’ positions towards equidistant spacings.

4.3. RELAXATION IN THE TWO-DIMENSIONAL CASE

In the two-dimensional case, we will use the zeroth-order method to relax the vertices of a triangular mesh to obtain a mesh with triangles which are as close to equilateral as possible. In Fig. 14, \(r_1, r_2,\) and \(r_3\) are the position vectors of the vertices. With the oriented particle connections \(l_{12} = r_1 - r_2\) and \(l_{13} = r_1 - r_3\) the force \(F_1\) on \(r_1\) can be written as:
\[
F_1 = - \left( |l_{12}| - \bar{l} \right) \frac{l_{12}}{|l_{12}|} - \left( |l_{13}| - \bar{l} \right) \frac{l_{13}}{|l_{13}|}.
\]
We define \( \bar{l} \) as the mean length of all three of the triangles edges. Like in Section 4.2, point \( r_1 \) will experience an attracting force from \( r_2 \) or \( r_3 \) if the distance between them is larger than \( \bar{l} \), and a repulsive force if the distance is smaller than \( \bar{l} \). We use our force definition and the zeroth-order method to run the relaxation algorithm on triangular meshes with different distributions of points. During the relaxation, the Delaunay triangulation can be performed again after some relaxation steps in order to prevent triangles from overlapping and to improve the quality of the grid. Initially we positioned more points on the left side of the grid like in Fig. 15(a) so that we get more, but smaller triangles on the left side. This will allow to show the effect of our relaxation algorithm on an inhomogeneous point distribution. The points on the four corners of the boundaries are fixed while points on the boundary can only move along the boundaries. The position of the boundary is prohibited from moving during the relaxation process, else the mesh would contract like a taut membrane. As we can see from Fig. 15(b), the algorithm tried to shift the points to the right. The number of triangles close to equilateral increases and the mesh becomes “nicer” than before.

**Fig. 15** Result of relaxation algorithm on the randomly generated Delaunay triangular mesh, before relaxation (a) and after relaxation (b): The strong size differences between the largest and smallest triangles is significantly reduced.

### 4.4. RELAXATION OF PARTS OF THE STAGGERED GRID.

Next, we applied the algorithm to the cut-out of a staggered grid in Fig. 16. The cut is chosen incommensurably, so that the nodes do not fall on the boundary, which results in a wide distribution of mesh sizes. We then use the Delaunay triangulation to generate the mesh (Fig 16, gray line). Finally, we applied our relaxation algorithm to optimize the mesh (Fig 16, black line). There is a considerable improvement especially in the shape of the triangles near the boundaries, while in the middle, the structure of the staggered grid is maintained. It is also gratifying to know that where is nothing to optimize, our algorithm does not interfere with the configuration.

Further, we applied our relaxation algorithm to the mesh in Section 3.3 in which six particles were suspended in the shape of a cannonball-stacking for the simulation with fluid. From Fig. 10 and Fig. 17, we can see that the improvement close to the particles and especially in the “pore space” between the particles is not as significant as in the previous examples. As the mesh points on the corners of the particles will not allow changes, the mesh between the particles cannot be optimized. One possibility of further improvement is to allow the points between the corners of the particles to move along the edges. However, before we make further modifications, we want to implement the algorithm in the actual finite element simulation of Suzuki\(^7\) after Gresko-Sani\(^1\). For a “better” mesh, by definition the error should be smaller than for a “not so good” mesh. If an adaptive time-integration method is employed, the timestep will increases for smaller error in the spatial discretization. An adaptive-timestep-adaptive-mesh algorithm will run with larger (and less) timesteps if the grid is “better” (produces less error due to the spatial discretization). This will allow to judge also the quality of our algorithm, as well as of other modifications, like shifts.
of edges between adjoining triangles. The actual time consumption of our remeshing algorithm is not a concern: For a fluid simulation around granular particles, the fluid dynamics part can be expected to take much more computer time than the remeshing or the relaxation algorithm. That does not mean that the time consumption for a simulation with particles will be the same as without particles for the same fluid volume: The particle interaction may introduce a smaller timescale. Preliminary simulations nevertheless indicate that for the Reynolds numbers achievable, the collision time for simulations with fluid were larger than for simulations without fluid.

5. SUMMARY

In this research, we successfully developed an automatic mesh generator by using a constrained Delaunay triangulation. In the future, we want to use this algorithm for on-the-fly remeshing in simulations with granular particles in fluids at varying concentrations. The results in Section 4.3 show that our relaxation algorithm works well for optimizing the triangular mesh in the fluid space. However, the mesh in the pore space between the granular particles is hardly affected. The points on the edges and corners of the granular particles vertices are fixed, while there are too few other points to equilibrate the neighboring triangles. We may have to modify the algorithm so that points between the vertices of the particles are introduced which can move along the edges to test the stability and accuracy of the fluid simulation with different numbers of auxiliary points. This should allow the simulation of flow in porous media with satisfying accuracy in the future.

REFERENCES