**EDF R&D**

FLUID DYNAMICS, POWER GENERATION AND ENVIRONMENT DEPARTMENT
SINGLE PHASE THERMAL-HYDRAULICS GROUP

6, QUAI WATIER
F-78401 CHATOU CEDEX

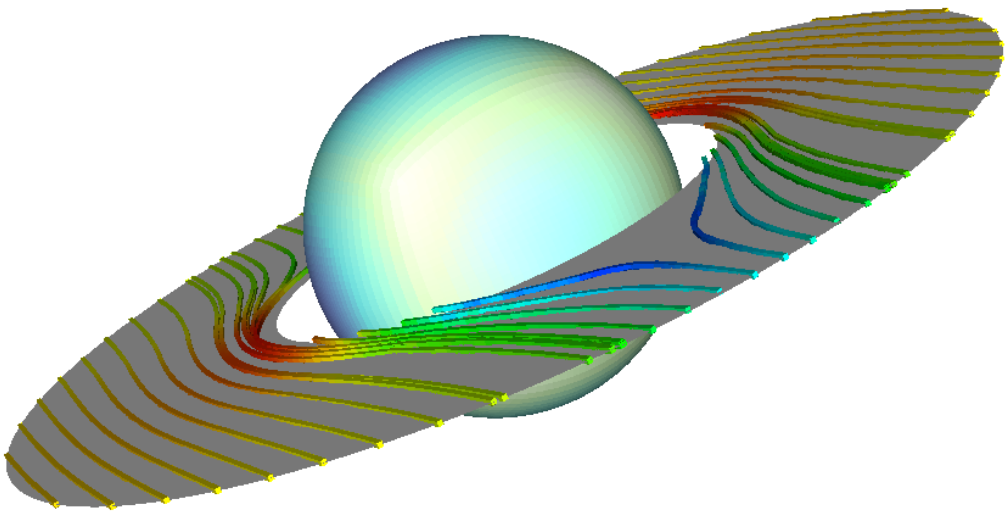
TEL: 33 1 30 87 75 40
FAX: 33 1 30 87 79 16

JUNE 2014

*Code_Saturne* documentation

# *Code_Saturne* version 2.0: Preprocessor Theory Guide

contact: saturne-support@edf.fr



*EXECUTIVE SUMMARY*

The preprocessing stages and tools of *Code_Saturne* simplify the use of this tool, by allowing import of multiple meshes of different formats, by allowing joining of non conforming meshes, and by checking the consistency of the resulting calculation mesh.

This document describes the principles of the geometrical algorithms which are used, allowing the user to better understand the way mesh joining works and how some parameters may influence the result.

**SUMMARY**

# 1 Geometric Algorithms

In this chapter, we will describe algorithms used for several operations done by *Code_Saturne*.

## 1.1 Geometric Quantities

Face normals (whose lengths are equal to face surfaces) as well as face centers of gravity and cell centers are computed directly by the kernel, though face normals may also be computed by the preprocessor for conforming joining.

### 1.1.1 Normals and Face Centers

To calculate face normals, we taken care to use an algorithm that is correct for any planar simple polygon, including non-convex cases. The principle is as follows : take an arbitrary point $P_a$ in the same plane as the polygon, then compute the sum of the vector normals of triangles $\{P_a, P_i, P_{i+1}\}$, where $\{P_1, P_2, ..., P_i, ..., P_n\}$ are the polygon vertices and $P_{n+1} \equiv P_0$. As shown on figure 1, some normals have a "positive" contribution while others have a "negative" contribution (taking into account the ordering of the polygon's vertices). The length of the final normal obtained is equal to the polygon's surface.

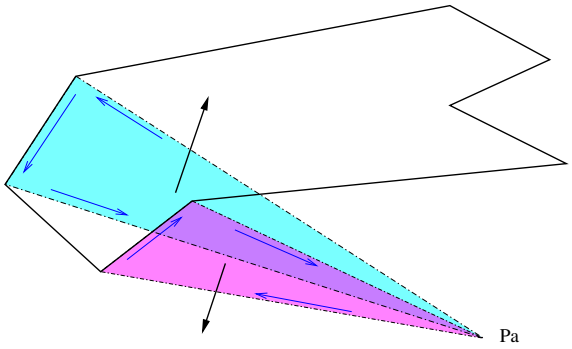



FIG. 1 – Face normals calculation principle

In our implementation, we take the "arbitrary" $P_a$ point as the center of the polygon's vertices, so as to limit precision problems due to truncation errors and to ensure that the chosen point is on the polygon's plane.

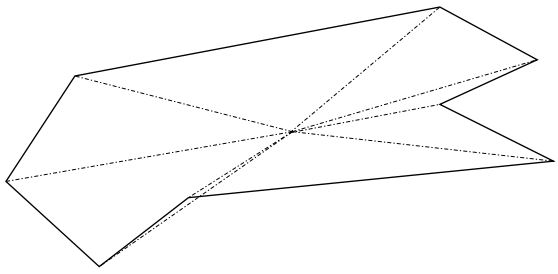


FIG. 2 – Triangles for calculation of face quantities

A face's center is equal to the weighted center $G$ of triangles $T_i$ defined as $\{P_a, P_i, P_{i+1}\}$ and whose centers are noted $G_i$. Let $O$ be the center of the coordinate system and $\overrightarrow{n_f}$ the face normal, then :

$$\overrightarrow{OG} = \frac{\sum_{i=1}^{n} \text{surf}(T_i).\overrightarrow{OG_i}}{\sum_{i=1}^{n} \text{surf}(T_i)} \qquad \text{avec} \quad \text{surf}(T_i) = \frac{\overrightarrow{n_{T_i}}.\overrightarrow{n_f}}{\| \overrightarrow{n_f} \|}$$

It is important to use the signed surface of each triangle so that this formula remains true in the case of non convex faces.

In real cases, some faces are nor perfectly planar. In this case, a slight error is introduced, but it is difficult to choose an exact and practical (implementation and computing cost wise) definition of a polygon's surface when its edges do not all lie in a same plane.

So as to limit errors due to warped faces, we compare the contribution of a given face to the the neighboring cell volumes (through Stoke's formula) with the contribution obtained from the separate triangles $\{P_a, P_i, P_{i+1}\}$, and we translate the initial center of gravity along the face normal axis so as to obtain the same contribution.

### 1.1.2 Cell Centers

If we consider that in theory, the Finite Volume method uses constant per-cell values, we can make without a precise knowledge of a given cell's center, as any point inside the cell could be chosen. In practice, precision (spatial order) depends on a good choice of the cell's center, as this point is used for the computation of values and gradients at mesh faces. We do not compute the center a the circumscribed sphere as a cell center, as this notion is usually linked to tetrahedral meshes, and is not easily defined and calculated for general polyhedra.

Let us consider a cell $\mathcal{C}$ with $p$ faces of centers of gravity $G_k$ and surfaces of norm $S_k$. If $O$ is the origin of the coordinate system, $\mathcal{C}$'s center $G$ is defined as :

$$\overrightarrow{OG} = \frac{\sum_{k=1}^{p} S_k . \overrightarrow{OG_k}}{\sum_{k=1}^{p} S_k}$$

An older algorithm computed a cell $\mathcal{C}$'s center of gravity as the center of gravity of its vertices $q$ of coordinates $X_l$ :

$$\overrightarrow{OG} = \sum_{l=1}^{q} \frac{\overrightarrow{OX_l}}{q}$$

In most cases, the newer algorithm gives better results, though there are exceptions (notably near the axis of partial meshes with rotational symmetry).

On figure 3, we show the center of gravity calculated with both algorithms on a 2D cell. On the left, we have a simple cell. On the right, we have added additional vertices as they occur in the case of a joining of non conforming faces. The position of the center of gravity is stable using the newer algorithm, whilst this point is shifted towards the joined sub-faces with the older, vertex-based algorithm (which worsens the mesh quality).

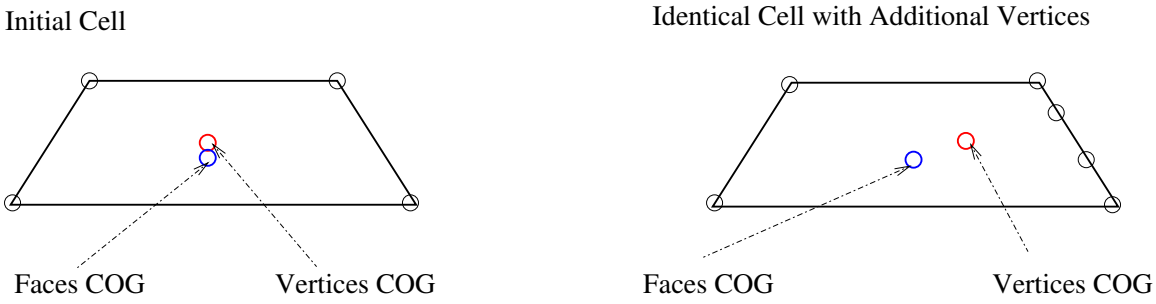



FIG. 3 – Choice of cell center

On figure 4, we show the possible effect of the choice of a cell's COG on the position of line segments joining the COG's of neighboring cells after a joining of non-conforming cells.

We see here that the vertex-based algorithm tends to increase non-orthogonality of faces, compared to the present face-based algorithm.

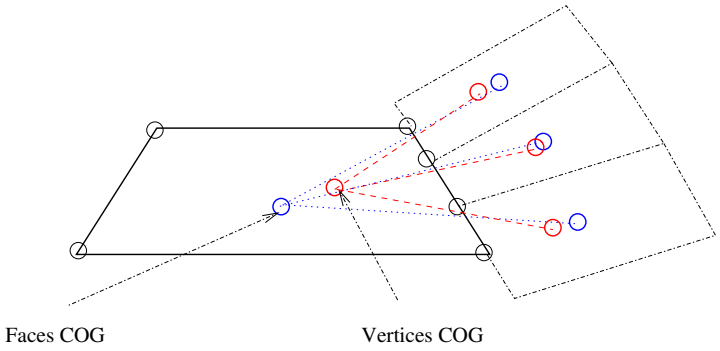



FIG. 4 – Choice of cell center and face orthogonality

## 1.2 Conforming Joining

The use of non conforming meshes is one of *Code_Saturne*'s key features, and the associated algorithms constitute the most complex part of the code's preprocessor. The idea is to build new faces corresponding to the intersections of the initial faces to be joined. Those initial faces are then replaced by their covering built from the new faces, as shown on figure 5 (from a 2D sideways perspective) :

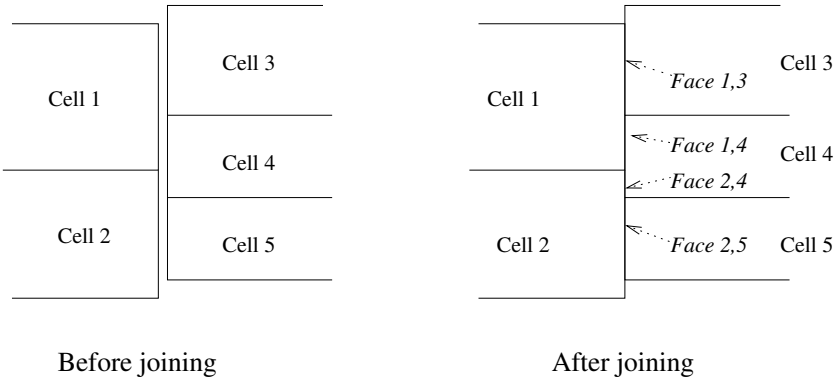



FIG. 5 – Principle of face joinings

We speak of *conforming joining*, as the mesh resulting from this joining is conforming, whereas the initial mesh was not.

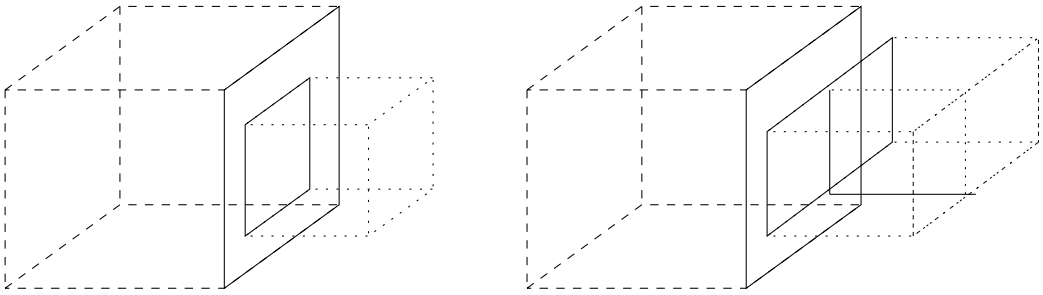
The number of faces of a cell of which one side has been joined will be greater or equal to the initial number of faces, and the new faces resulting from the joining will not always be triangles or quadrangles, even if all of the initial faces were of theses types. For this reason, the data representations of *Code_Saturne* and its preprocessor are designed with arbitrary simple polygonal faces and polyhedral cells in mind. We exclude polygons and polyhedra with holes from this representation, as shown on figure 6. Thus the cells shown in figure 6 cannot be joined, as this would require opening a "hole" in one of the larger cell's faces. In the case of figure 6b, we have no such problem, as the addition of another smaller cell splits the larger face into pieces that do not contain holes.

### 1.2.1 Robustness Factors

We have sought to build a joining algorithm that could function with a minimum of user input, on a wide variety of cases.

Several criteria were deemed important :

1. **determinism** : we want to be able to predict the algorithm's behavior. We especially want the algorithm



a) Impossible Definition b) Possible Definition

FIG. 6 – Possible case joinings

to produce the same results whether some mesh $A$ was joined to mesh $B$ or $B$ to $A$. This might not be perfectly true in the implementation due to truncation errors, but the main point is that the user should not have to worry about the order in which he enters his meshes for the best computational results. [1]

2. **non planar surfaces** : We must be able to join both curved surface meshes and meshes of surfaces assembled from a series of planar sections, but whose normal is not necessarily a continuous function of space, as shown on figure 7.

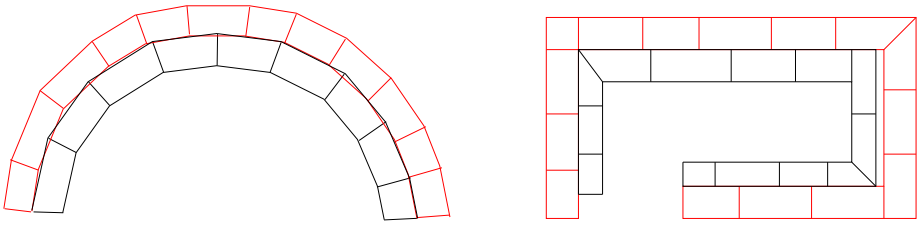


FIG. 7 – Initial surfaces

3. **spacing between meshes** : the surfaces to be joined may not match perfectly, due to truncation errors or precision differences, or to the approximation of curved surfaces by a set of planar faces. The algorithm must not leave gaps where none are desired.

### 1.2.2 Basic Principle

Let us consider two surfaces to join, as in figure 8 : We seek to determine the intersections of the edges of the mesh faces, and to split these edges at those intersections, as shown on figure 9. We will describe more precisely what we mean by "intersection" of two edges in a later subsection, as the notion involves spanning of small gaps in our case.

The next step consists of reconstructing sub-faces derived from the initial faces. Starting from an edge of an initial face, we try to find closed loops, choosing at each vertex the leftmost edge (as seen standing on that face, normal pointing upwards, facing in the direction of the current edge), until we have returned to the starting vertex. This way, we find the shortest loop turning in the trigonometric direction. Each face to be joined is replaced by its covering of sub-faces constructed in this manner.

When traversing the loops, we must be careful to stay close to the plane of the original face. We thus consider only the edges belonging to a face whose normal has a similar direction to that of the face being subdivided (i.e. the absolute value of the dot product of the two unitary normals should be close to 1).

[1] The geometry produced by a joining is in theory independent from the order mesh inputs, but mesh numbering is not. The input order used for a calculation should thus be kept for any calculation restart.

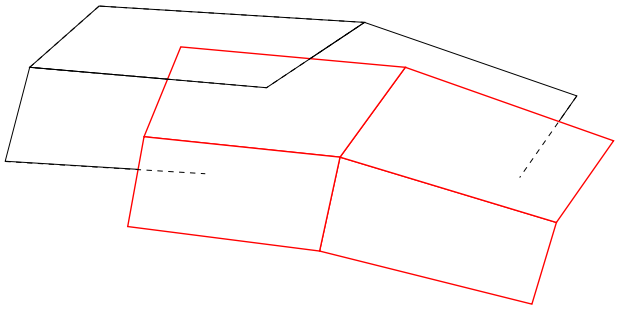


FIG. 8 – Surfaces to be joined

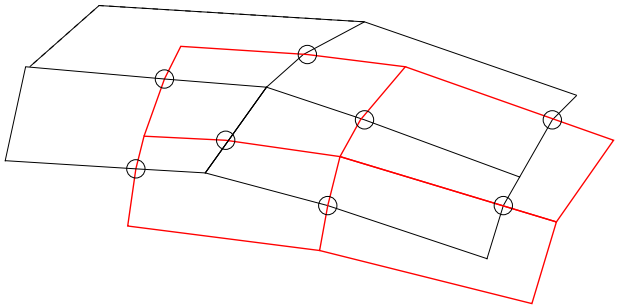


FIG. 9 – After edge intersections

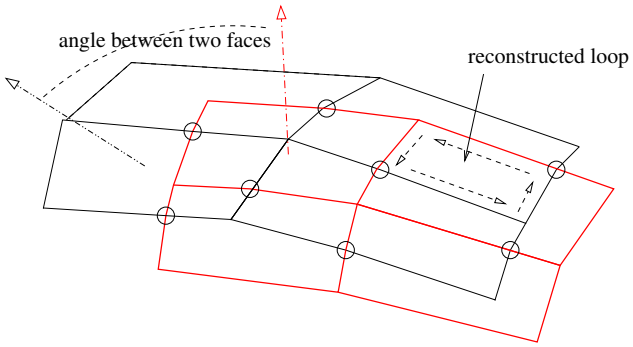



FIG. 10 – Sub-face reconstruction.

Once all the sub-faces are built, we should have obtained for two tangent initial faces two topologically identical sub-faces, each descending from one of the initial faces and thus belonging to a different cell. All that is required at this stage is to merge those two sub-faces, conserving the properties of both. The merged sub-face thus belongs to two cells, and becomes an internal face. The joining is thus finalized.

### 1.2.3 Simplification of Face Joinings

For a finite-volume code such as *Code_Saturne*, it is best that faces belonging to one same cell have neighboring sizes. This is hard to ensure when non-conforming border faces are split so as to be joined in a conforming way. On figure 11, we see that this can produce faces of highly varying sizes when splitting a face for conformal joining.

It is possible to simplify the covering of a face so as to limit this effect, by moving vertices slightly on each side of the covering. We seek to move vertices so as to simplify the covering while deforming the mesh as little as possible.

One covering example is presented figure 11, where we show several simplification opportunities. We note that all these possibilities are associated with a given edge. Similar possibilities associated with other edges are not shown so that the figure remains readable. After simplification, we obtain the situation of figure 12.

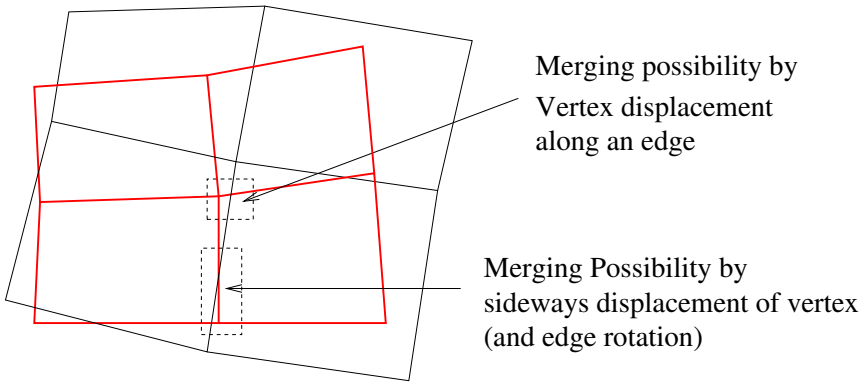



FIG. 11 – Simplification possibilities

After simplification, we have the following situation :

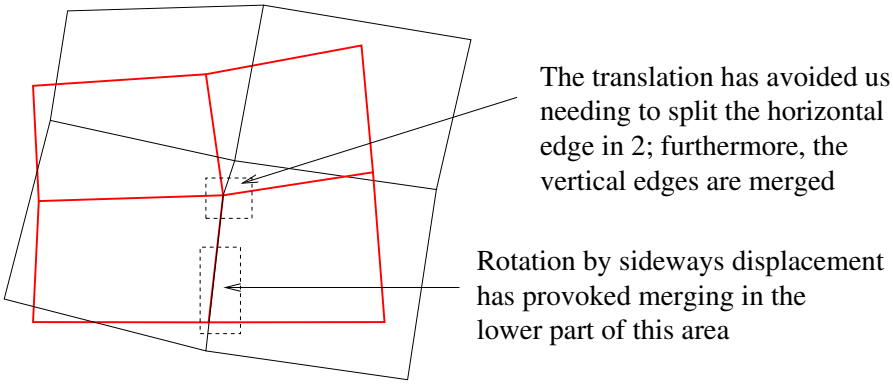



FIG. 12 – Faces after simplification

### 1.2.4 Processing

The algorithm's starting point is the search for intersections of edges belonging to the faces selected for joining. In 3D, we do not restrict ourselves to "true" intersections, but we consider that two edges intersect as soon as the minimum distance between those edges is smaller than a certain tolerance.

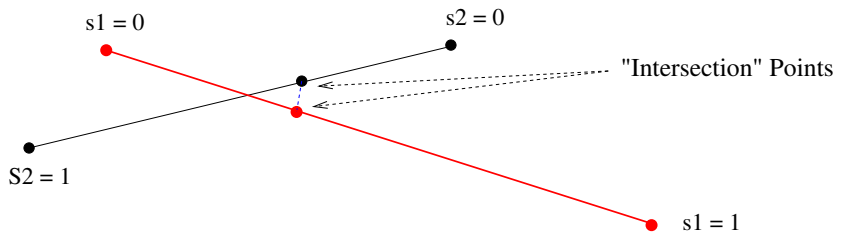



FIG. 13 – Intersection of edges in 3d space

To each vertex we associate a maximal distance, some factor of the length of the smallest edge incident to that vertex. This factor is adjustable (we use 0.1 by default), but should always be less than 0.5.

We consider a neighborhood of an edge defined by the spheres associated to the minimal distances around each vertex and the shell joining this two spheres, as shown on figure 14. More precisely, at a point on the edge of linear abscissa $s$, the neighborhood is defined to be the sphere of radius $d_{max}(s) = (1-s)d_{max} \mid_{s=0} +s.d_{max} \mid_{s=1}$.

We will thus have intersection between edges $E1$ and $E2$ as soon as the point of $E1$ closest to $E2$ is within the neighborhood of $E2$, and that simultaneously, the point of $E2$ closest to $E1$ is inside the neighborhood of $E1$.

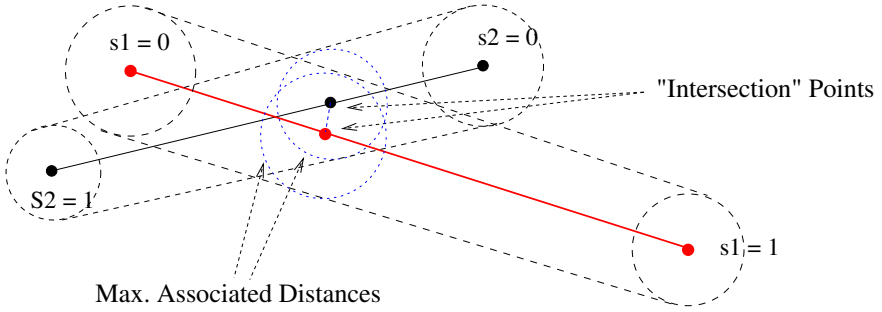



FIG. 14 – Tolerances for intersection of edges

If edge $E2$ cuts the neighborhood of a vertex of edge $E1$, and this vertex is also in $E2$'s neighborhood, we choose this vertex to define the intersection rather than the point of $E1$ closest to $E2$. This avoids needlessly cutting edges. We thus search for intersections with the following order of priority : vertex-vertex, vertex-edge, then edge-edge. If the neighborhoods associated with two edges intersect, but the criterion :
$\exists P1 \in A1, \exists P2 \in A2, d(P1,P2) < min(d_{max}(P1), d_{max}(P2))$
is not met, we do not have intersection. These cases are shown on figure 15.

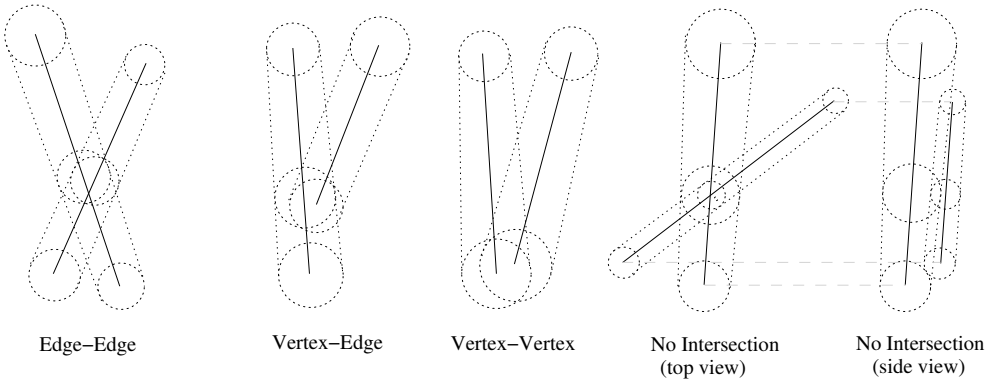



FIG. 15 – Tolerances for intersection of edges

### 1.2.5 Problems Arising From the Merging of Two Neighboring Vertices

If we have determined that a vertex $V_1$ should be merged with a vertex $V_2$ and independently that this vertex $V_2$ should be merged with a vertex $V_3$, then $V_1$ and $V_3$ should be merged as a result, even though these vertices share no intersection. We refer to this problem as merging transitivity and show theoretical situations leading to it on figure16.

On figure17, we show cases more characteristic of what we can obtain with real meshes, given that the definition of local tolerances reduces the risk and possible cases of this type of situation.

Figure 18 illustrates the effect of a combination of merges on vertices belonging to a same edge. We see that in this case, edges initially going through vertices $G$ and $J$ are strongly deformed (i.e. cut into sub-edges that are not well aligned). Without transitivity, edges going through vertices descended only from the merging of $(G,H)$ on one hand and $(L,J)$ on the other hand would be much closer to the initial edges.

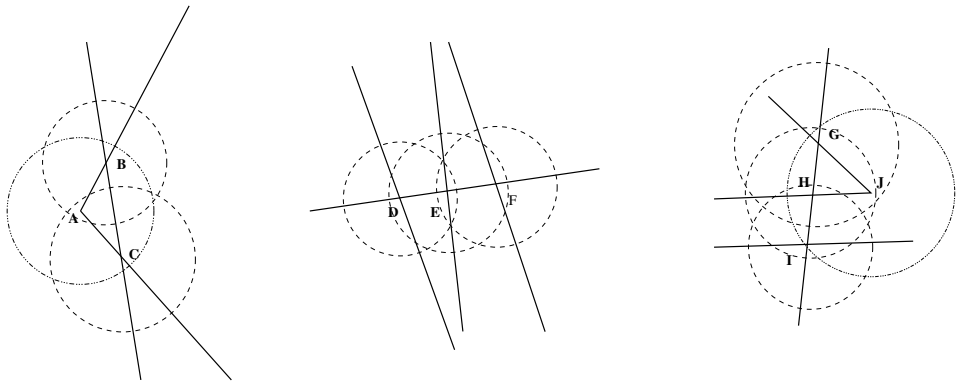



FIG. 16 – Merging transitivity.

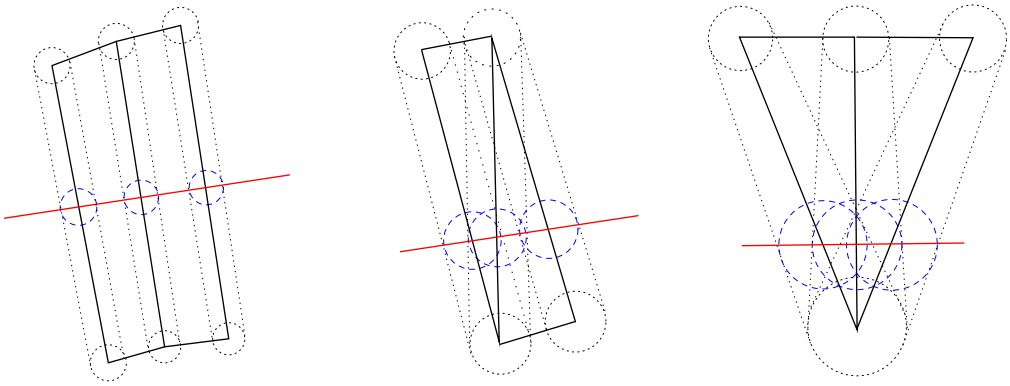


FIG. 17 – Merging transitivity (real cases)

To avoid excessive simplifications of the mesh arising from a combination of merges, we check for every edge if such situations occur. If such is the case, we compute a local multiplicative factor ($< 1$) for the intersection/merge tolerance associated with vertices of this edge so as to merge as many vertices on this edge without provoking undesired merges through transitivity.

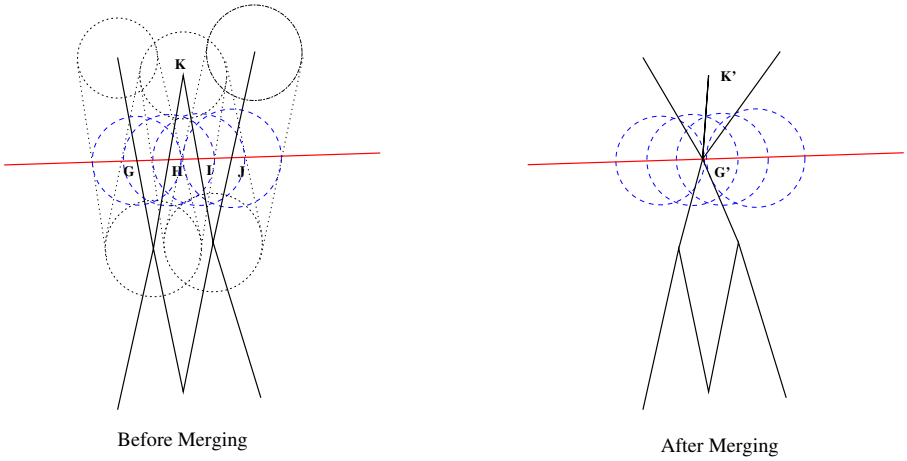



FIG. 18 – Merging transitivity for an edge

Finally, on figure 19, we show the possible effect of merge transitivity on vertices belonging to several edges. Here, even if the local tolerance reduction avoids the merging of vertices $(G, I)$ through combination of merges $(G, H)$ and $(H, I)$, this merging could be provoked by transitivity of merges $(H, I)$, $(H, J)$, and $(G, J)$. In

addition, $I$ and $J$ would then be merged when they should not be.

In this specific example, the local tolerance reduction on each edge should avoid merge $(H, I)$, because the tolerances associated with $G$ and $H$ are greater than that associated with $I$ ; the local tolerance reduction should thus conserve merge $(G, H)$ and avoid merge $(H, I)$. We will then not have merging of $(I, J)$ by transitivity, and only $(G, H, J)$ will be merged.

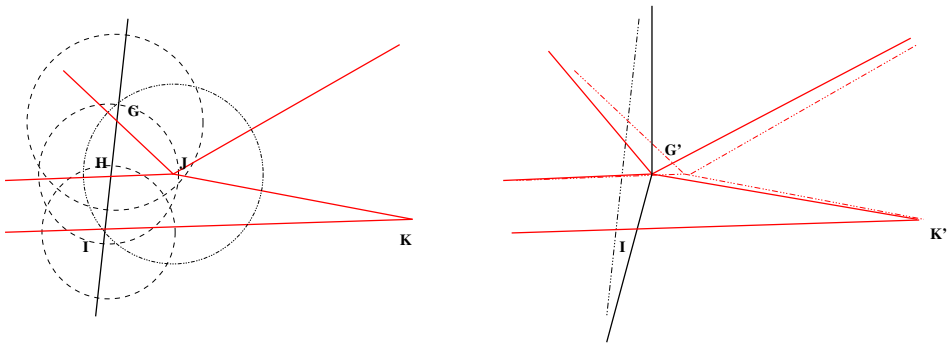



FIG. 19 – Avoided merging transitivity for several edges

On the configuration shown figure 20, where tolerances associated with vertices $G$, $H$, and $I$ are different from those of figure 19, the local tolerance reduction will first merging of $(G, H)$ at first, but $(H, I)$ will be merged, which in combination with merge $(H, J)$ will in the end lead to merging of $(G, H, I, J)$ into $G'$.

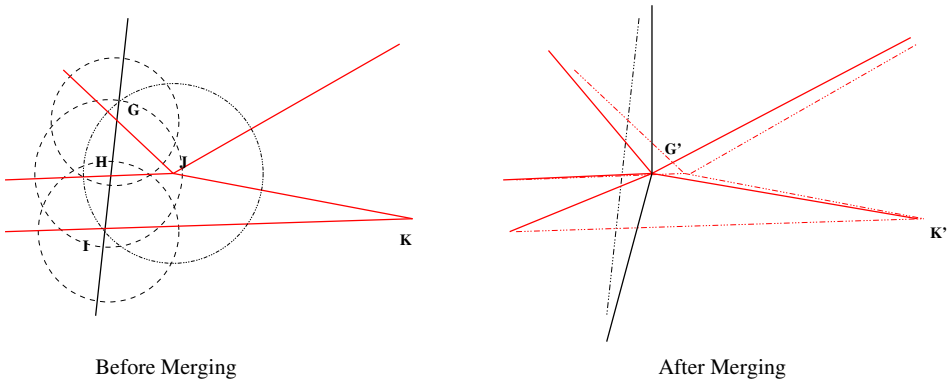



FIG. 20 – Merging transitivity for several edges

No special handling is implemented for this last case : local tolerance reduction is only done on a per-edge basis. To account for all cases while maintaining the algorithm's theoretical independence to the faces and edges orderings, it would in theory be possible to run multiple passes, so as to reduce local tolerances not only as regards merging of vertices on a single edge but also intersection detection. On the downside, the impact of such a tolerance reduction on the algorithm's output would be more difficult to predict, and the implementation would be yet more complex. As we have not so far encountered any robustness problems that and adjustment of the default tolerance did not solve (and even this is rarely required for a planar joining), we have chosen not to make the algorithm more complex, as the tolerance reduction which avoids per-edge merge transitivity seems to solve most problems.

### 1.2.6 Algorithm Optimization

Certain factors influence both memory an CPU requirements. We always try to optimize both, with a slight priority regarding memory requirements.

When searching for edge intersections, we try to keep the number of intersection tests to a minimum. We compute bounding boxes associated with each edge, and run a full edge intersection test only for edges whose bounding boxes intersect, in which case the probability of having a true intersection is high. These boxes are aligned with the coordinate system axes, and are large enough to contain not an edge but also the neighborhood associated with its tolerance, as shown on figure 21.

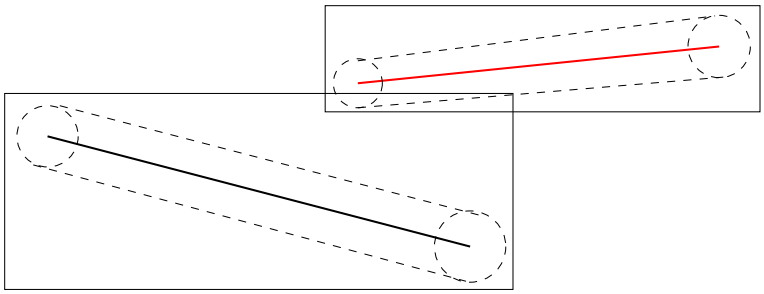


FIG. 21 – Bounding boxes for edges

We loop on all selected faces, using a hybrid dichotomy-traversal algorithm to search for all edges intersecting the current edge. Let $i$ be the index of the current edge, we only search for intersections with edges of index $j$ such that $j > i$, as the other cases should have been detected when $j$ was the current edge.

### 1.2.7 Influence on mesh quality

It is preferable for a FV solver such as *Code_Saturne* that the mesh be as "orthogonal" as possible (a face is perfectly orthogonal when the segment joining its center of mass to the center of the other cell to which it belongs is perfectly aligned with its normal). It is also important to avoid non planar faces [2]. By the joining algorithm's principle, orthogonal faces are split into several non-orthogonal sub-faces. In addition, the higher the tolerance setting, the more merging of neighboring vertices will tend to warp faces on the sides of cells with joined faces.

It is thus important to know when building a mesh that a mesh built by joining two perfectly regular hexahedral meshes may be of poor quality, especially if a high tolerance value was used and the cell-sizes of each mesh are very different. It is thus important to use the mesh quality criteria visualizations available in *Code_Saturne*, and to avoid arbitrary joinings in sensible areas of the mesh. When possible, joining faces of which one set is already a subdivision of another (to construct local refinements for example) is recommended.

# 2 Periodicity

We use an extension of the non-conforming faces joining algorithm to build periodic structures. The basic principle is described figure 22 :

– Let us select a set of boundary faces. These faces (and their edges and vertices) are duplicated, and the copy is moved according to the periodic step (a combination of translation and rotation). A link between original and duplicate entities is kept.
– We use a conforming joining on the union of selected faces and their duplicates. This joining will slightly deform the mesh so that vertices very close to each other may be merged, and periodic faces may be split into conforming sub-faces (if they are not already conforming).
– If necessary, the splitting of duplicated, moved, and joined faces is also applied to the faces from which they were descended.
– Duplicated entities which were not joined (i.e. excess entities) are deleted.

[2] Computation of face COG's includes a correction such that the contribution of a warped face to a cell's volume is the same as that of the same face split into triangles joining that face's COG and outer edges, but this correction may not be enough for second order values.

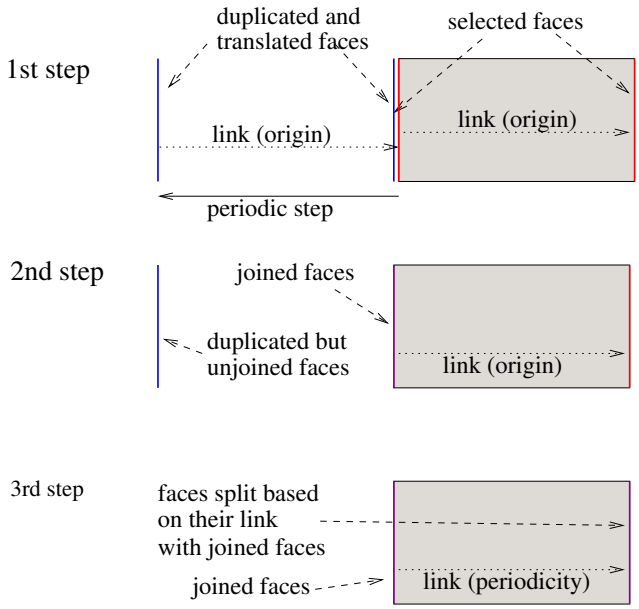



FIG. 22 – Periodic joining principle (translation example)

It is thus not necessary to use periodic boundary conditions that the periodic surfaces be meshed in a periodic manner, though it is always best not to make to much use of the comfort provided by conforming joinings, as it can lower mesh quality (and as a consequence, quality of computational results).

We note that it could seem simpler to separate periodic faces in two sets of "base" and "periodic" faces, so as to duplicate and transform only the first set. This would allow for some algorithm simplifications and optimizations, but here we gave a higher priority to the consistency of user input for specification of face selections, and the definition of two separate sets of faces would have made user input more complex. As for the basic conforming joining, it is usually not necessary to specify face selections for relatively simple meshes (in which case all faces on the mesh boundary are selected).

# 3 Triangulation of faces

Face triangulation (handled by the FVM library) is done in two stages. The first stage uses an *ear cutting* algorithm, which can triangulate any planar polygon, whether convex or not. The triangulation is arbitrary, as it depends on the vertex chosen to start the loop around the polygon.

The second stage consists of flipping edges so that the final triangulation is constrained Delaunay, which leads to a more regular triangulation.

## 3.1 Initial triangulation

The algorithm used is based on the one described in [2]. Its principle is illustrated figure 23. We start by checking if the triangle defined by the first vertices of the polygon, $(P_0, P_1, P_2)$ is an "ear", that is if it is interior to the polygon and does not intersect it. As this is the case on this example, we obtain a first triangle, ad we must then process the remaining part of the polygon. At the next stage triangle $(P_0, P_2, P_3)$ is also an ear, and may be removed.

At stage 2, we see that the next triangle which is a candidate for removal, $(P_0, P_3, P_4)$ is not an ear, as it is not contained in the remaining polygon. We thus shift the indexes of vertices to consider, and see at stage $4$ that triangle $(P_3, P_4, P_5)$ is an ear and may be removed.

The algorithm is built in such a way that a triangle is selected based on the last vertex of the last triangle

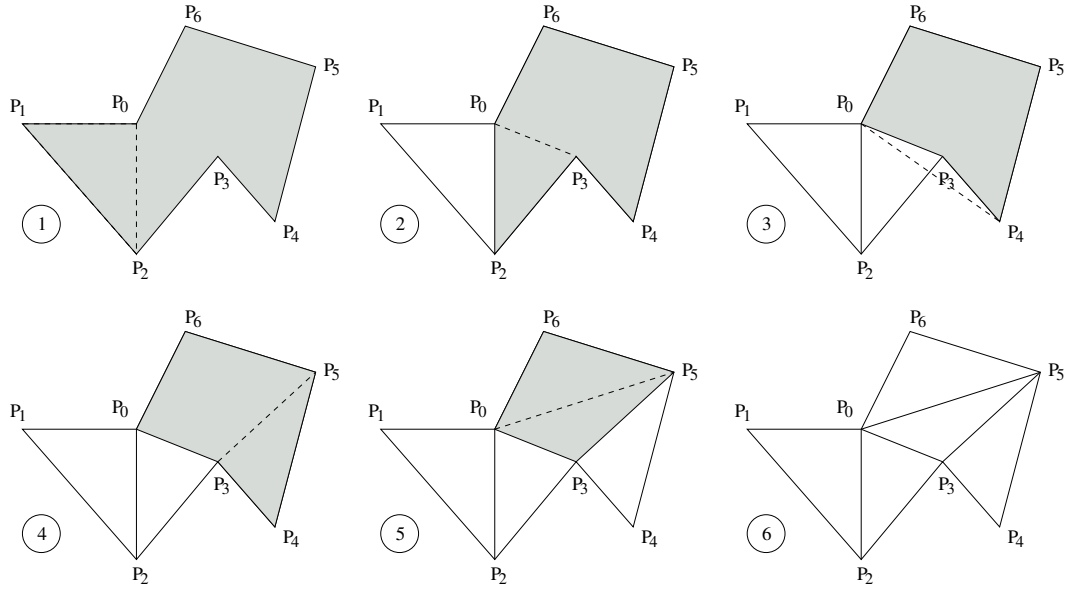



FIG. 23 – Principle of face triangulation

considered (starting from triangle $(P_0, P_1, P_2)$). Thus, we consider at stage 5 the triangle ending with $P_5$, that is $(P_0, P_3, P_5)$. Once this triangle is removed, the remaining polygon is a triangle, and its handling is trivial.

## 3.2 Improving the Triangulation

We show on figures 24 and 25 two examples of a triangulation on similar polygons whose vertices are numbered in a different manner.

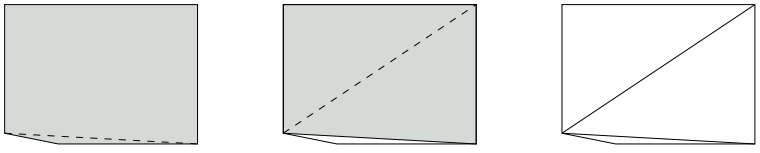



FIG. 24 – Triangulation example (1)

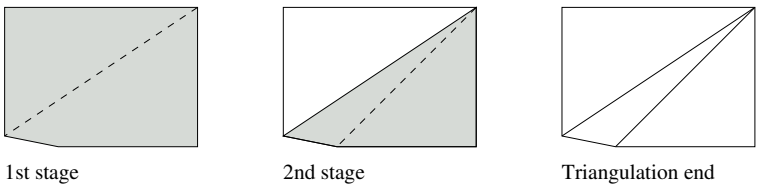



FIG. 25 – Triangulation example (2)

Not only is the obtained triangulation different, but it has a tendency to produce very flat triangles. Once a first triangulation is obtained, we apply a corrective algorithm, based on edge flips so as to respect the Delaunay condition.

This condition is illustrated figure 26. In the first case, edge $\overline{P_iP_j}$ does not respect the condition, as vertex $P_l$ is contained in the circle going through $P_i, P_j, P_k$. In the second case, edge $\overline{P_kP_l}$ respects this condition, as $P_i$ is not contained in the circle going through $P_j, P_k, P_l$.

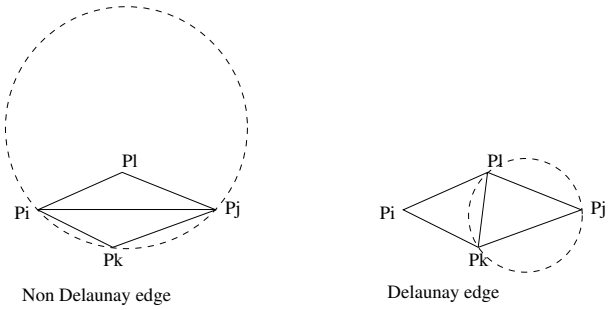



FIG. 26 – Delaunay condition (2)

If triangles $(P_i, P_k, P_j)$ and $(P_i, P_j, P_l)$ originated from the initial triangulation of a same polygon, they would thus be replaced by triangles $(P_i, P_k, P_l)$ and $(P_l, P_k, P_j)$, which respect the Delaunay condition. In the case of a quadrangle, we would be done, but with a more complex initial polygon, these new triangles could be replaced by others, depending on their neighbors from the same polygon.

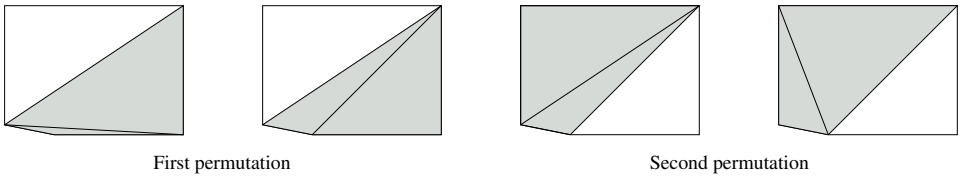



FIG. 27 – Edge flip example (1)

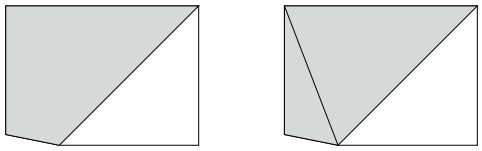


FIG. 28 – Edge flip example (2)

On figures 27 and 28, we illustrate this algorithm on the same examples as before (figures 24 and 25). We see that the final result is the same. In theory, the edge flipping algorithm always converges. To avoid issues due to truncation errors, we allow a tolerance before deciding to flip two edges. Thus, we allow that the final triangulation only "almost" respect the Delaunay condition.

In some cases, especially that of perfect rectangles, two different triangulations may both respect the Delaunay condition, notwithstanding truncation errors. For periodicity, we must avoid having two periodic faces triangulated in a non-periodic manner (for example, along different diagonals of a quadrangle). In this specific case, we triangulate only one face using the geometric algorithm, and then apply the same triangulation to it's periodic face, rather then triangulate both faces independently.

# 4 Bibliography

[1] Y. FOURNIER
*Définition du module Enveloppe pour le Solveur Commun*,
EDF Report HE-41/99/017/A, 1999

[2] T. THEUSSL
*A Review of Two Simple Polygon Triangulation Algorithms*,
"Special Topics in Computer Graphics 1998" course
http ://www.cg.tuwien.ac.at/ theussl/

[3] G. T. TOUSSAINT
*Efficient Triangulation of Simple Polygons*,
The Visual Computer, vol. 7, 1991, pp. 280-295

[4] H. ELGINDY, H. EVERETT, G. T. TOUSSAINT
*Slicing an Ear Using Prune and Search*,
Pattern Recognition Letters, vol. 14, September 1993, pp. 719-722.

[5] J.R. SHEWCHUK
*Lecture Notes on Delaunay Mesh Generation*,
Lecture notes
http ://www.cs.berkeley.edu/ jrs/meshf99/

[6] M. BERN, D. EPPSTEIN
*Mesh Generation and Optimal Triangulation*,
Computing in Euclidean Geometry, 2nd ed., D.-Z. Du and F.K. Hwang, eds., World Scientific, 1995, pp. 47-123
http ://www.ics.uci.edu/ eppstein/pubs/geom-tri.html