Robustness and Efficiency of Geometric Programs The Predicate Construction Kit (PCK)

Bruno Lévy

Robustness and Efficiency of Geometric Programs The Predicate Construction Kit (PCK)[☆]

Bruno Lévy[a]

[a] *Inria Nancy Grand Est / Project Alice, Villers-lès-Nancy, 54600, France.*

Abstract

In this article, I focus on the robustness of *geometric programs* (e.g., Delaunay triangulation, intersection between surfacic or volumetric meshes, Voronoi-based meshing ...) w.r.t. numerical degeneracies. Some of these geometric programs require "exotic" predicates, not available in standard libraries (e.g., J.-R. Shewchuk's implementation and CGAL). I propose a complete methodology and a sample Open Source implementation of a toolset (PCK: Predicate Construction Kit) that makes it reasonably easy to design geometric programs free of numerical errors. The C++ code of the predicates is automatically generated from its formula, written in a simple specification language. Robustness is obtained through a combination of arithmetic filters, expansion arithmetics and symbolic perturbation.

As an example of my approach, I give the formulas and PCK source-code for the 4 predicates used to compute the intersection between a 3d Voronoi diagram and a tetrahedral mesh, as well as symbolic perturbations that provalby escapes the corner cases. This allows to robustly compute the intersection between a Voronoi diagram and a triangle mesh, or the intersection between a Voronoi diagram and a tetrahedral mesh. Such an algorithm may have several applications, including surface and volume meshing based on Lloyd relaxation.

Keywords: geometric predicates, arbitrary precision, symbolic perturbation, expansion arithmetics

1. Introduction

In this article, I focus on the robustness of *geometric programs* w.r.t. numerical degeneracies, and propose a complete methodology implemented in a toolset to easily design geometric programs free of numerical problems.

Some classical tasks in geometric modeling, including computing the Delaunay triangulation, computing the intersection between surfacic and/or

[☆] Open Source implementation: `http://alice.loria.fr/index.php/software/4-library/75-geogram.html`

volumetric meshes, and meshing using Voronoi diagrams are examples involving such geometric programs. A geometric programs take as an input a set of objects (points, lines, segments, triangles ...) and return a combinatorial structure (e.g. a mesh) that inter-connects the elements of the input.

1.1. Geometric Predicates

Geometric predicates are central components of geometric algorithms. They are functions that take as input a small set of geometric objects and return a binary (or ternary, see below) answer. For instance, the orientation predicate in 2d indicates for a set of points $\mathbf{p}_1$, $\mathbf{p}_2$, $\mathbf{p}_3$ whether the angle between $\overrightarrow{\mathbf{p}_1, \mathbf{p}_2}$ and $\overrightarrow{\mathbf{p}_1, \mathbf{p}_3}$ is a right turn or a left turn (or no turn at all if $\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3$ are aligned, thus "ternary" answer). Typically, in C/C++, the 2d orientation predicate is implemented as a function:

```
int orient2d(double* p1, double* p2, double* p3)
```

that returns -1 for a left turn, +1 for a right turn and 0 if the three points are aligned. The result of this function is used by the geometric algorithm, typically to decide which pairs of points / triples of points should be connected by an edge/triangle in a mesh, in other words, the predicates determine all the "combinatorial decisions" taken by the algorithm.

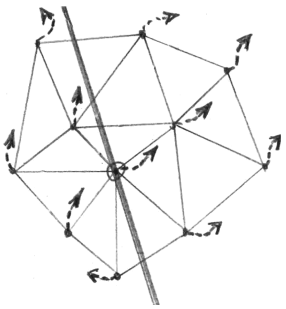
The mathematical definition of geometric predicates are most of the time quite simple, for instance, for the `orient2d` predicate mentioned above, this corresponds to the sign of the determinant of the two vectors. However, there are two major difficulties for implementing a geometric predicate :

1. **double precision does not suffice:** When using standard floating point `double`s, for some configurations, the answer given by the implementation will differ from the exact value. More importantly, it may differ *inconsistently*. For instance, the same predicate may assert that $\mathbf{p}_1$ is strictly above $\mathbf{p}_2$, and that $\mathbf{p}_2$ is strictly above $\mathbf{p}_1$! This clearly leads to erroneous combinatorics (inconsistent mesh);
2. **corner cases are tricky:** there are configurations where the predicate answers 0. For instance, in the case of `orient2d`, this corresponds to three points that are aligned. If you use the predicate to compute the intersection between a polygon and a straight line, this corresponds to the configuration where a vertex of the polygon lies exactly on the straight line. While such configurations are reasonably easy to handle in 2d, this quickly becomes very complicated for 3d intersection problems.

Two techniques have been developed to handle both difficulties :

Arbitrary precision: to deal with the first issue, the idea is to replace standard `double` floating point computation with custom types [17], that can represent numbers with arbitrary precision, using a dynamically allocated space (more on this below);

Avoiding corner cases with 'Simulation of Simplicity': to deal with the second issue, a natural (naive) idea would be to "move the points a little bit" whenever a degeneracy is encountered. Unfortunately this naive idea does not work in general. Imagine that the same point $\mathbf{p}$ is used several times as an argument of the predicate, for instance $P(\mathbf{p}, q, r)$ and $P(s, t, \mathbf{p})$, and imagine that the second configuration $P(s, t, \mathbf{p})$ is degenerate, then, if $\mathbf{p}$ is moved, in order to keep combinatorial decisions coherent, the information should be "back propagated in time" to the first invocation. A solution to this problem was introduced in [5], that defines a *globally consistent* perturbation and applies it *symbolically* : Imagine that all the points $\mathbf{p}_i$ follow a set of trajectories, parameterized by time t, and starting from their actual positions. Then each point $\mathbf{p}_i$ is replaced with a function $\mathbf{p}_i(t)$ such that $\mathbf{p}_i(0) = \mathbf{p}_i$ (dashed arrows on the figure). Whenever a predicate $P(\mathbf{p}, \mathbf{q}, \mathbf{r})$ returns 0 (such as the circled vertex when classified w.r.t. the solid line), the idea will be to replace the answer of the predicate with the (symbolically computed) limit: $\lim_{t \to 0} P(\mathbf{p}(t), \mathbf{q}(t), \mathbf{r}(t))$. In practice, to derive the implementation of a predicate P from its formula, this means choosing a perturbation $\mathbf{p}_i(t)$ (typically an exponent of t that depends on i), then computing the Taylor expansion of $P(\mathbf{p}_1(t), \mathbf{p}_2(t), \mathbf{p}_3(t))$ in function of t. The implementation tests all the terms of the Taylor expansions in increasing powers of t, and returns the sign of the first non-zero one. Note that the constant term corresponds to the unperturbed predicate.



At first sight, this may seem a pretty convoluted / overly complicated way of handling these corner cases, but from a software design point of view, this has very interesting properties : most geometric programs depend on a very small number of predicates. For instance, a robust implementation of Delaunay triangulation only needs two predicates, `orient()` and `in_sphere()`, and an algorithm to compute mesh intersections will mostly depend on some forms of `orient()` (more on this below). If your predicates implement the Simulation of Simplicity, this means that they will never answer 0, in other words, **corner cases are never encountered**. This has a dramatic influence on the development of the geometric program : imagine an algorithm that computes the intersection between two volumetric meshes. There is a wide variety of degenerate configurations that can occur

(vertex on a face, edge on a face, edge on an edge, coplanar faces ...). Using the symbolically perturbed predicates, these configurations never occur. **By simply replacing a couple of functions, all the corner cases "math-emagically" disappear !** The price to pay is that since the algorithm behaves as if the point was not on the plane, the algorithm will generate zero-area/zero-volume elements (see results in Section 4), but they can be filtered-out by a post-processing phase, that is much easier to develop than the algorithm that handles all corner cases explicitly. More importantly, it also makes it much easier to check the **correctness** of the program.

Efficiency/arithmetic filters At this point, to implement the predicate, one will derive its mathematical expression, compute the symbolic perturbation, and implement the formula for all the terms of the Taylor expansion with exact arithmetics (e.g., [17], described in the next section). However, exact arithmetics costs between $10\times$ and $50\times$ the time of standard floating point double precision arithmetics ! For this reason, to improve performances, it is interesting to have a way of determining the answer in the "easy cases", where we are sure that the result computed using double precision floating points is exact. In other words, this corresponds to configuration where we are sure that **the sign** of the computed floating point number is correct. Meyer and Pion introduced a method that computes bounds from the formula of the predicate [12]. If the absolute value of the result is larger than the bound, then they prove that the sign is correct. In addition, the bounds can be dynamically adjusted to the input, thus making it even more efficient (with these dynamic bounds, a quick decision can be taken in a much larger number of cases).

Summary: a predicate implementation depends on the following three components :

1. an **arithmetic filter :**[12], that gives an answer in the "easy cases" where the exact answer can be determined using standard floating point arithmetics. This is the key to *efficiency* (obtaining correctness and robustness without paying the price for it !);
2. **exact arithmetics :**[17], used when the filter could not determine the answer, this is the key to *correctness* and *robustness*;
3. **symbolic perturbation :**[5], that "escape" the corner cases by (symbolically) considering the limit of a globally consistent perturbation applied to the input. This is the key to *correctness*, *robustness* and *ease of implementation* of the geometric program on top of the predicate.

For Delaunay triangulation, predicates are well studied and understood, and several implementations are available, such as [4] and Shewchuk's code [17], that are widely used.

However, some mesh generation algorithm that we develop, such as the flow simulator with adaptive gridding [10, 11], our adaptive meshing algorithms [16, 15], and our hex-dominant meshing algorithm [2], require to compute the intersection between a 3D Voronoi diagram and a tetrahedral mesh. Robustness is a key issue, for instance our adaptive meshing algorithm optimizes a Voronoi mesh such that some facets of the Voronoi cells conform to the interfaces of a volumetric structural model (horizons and faults). In terms of mesh intersections, this means that we create degenerate configurations. Unfortunately, **the predicates that we need are "exotic"** and therefore not available in the standard packages.

Exact predicates "without the agonizing pain":[1] In the programming libraries mentioned above, the expertise on the three components often depend on a limited number of individuals (J. Shewchuk, S. Pion, O. Devillers ...). Since we will need to implement a whole family of more general predicates, my goal is now to develop a methodology such that *any practitioner of the field could create his own predicates.* Concerning the first layer (filters), this goal is already reached by the FPG tool [12]. Concerning the two other ones, *some new programming tools are needed.* In particular, we need an efficient and easy-to-use implementation of exact arithmetics.

A first solution: exact arithmetics using dynamic integers: There are already several existing programming libraries that implement exact arithmetics. For instance, when considering integer arithmetics, it is possible to represent an integer of arbitrary precision by an array of integers $a[0] \dots a[l-1]$. Such an array represents the integer x given by :

$$x = \sum_{i=0}^{l} 2^{32*i} a[i]$$

Then, it is possible to implement algebraic operations $(+, -, *, /)$, in a way that is very similar to computing by hand (except that 'digits' are numbers between 0 and $2^{32} - 1$ instead of $0, 1, \dots 9$). Some programming libraries, such as GMP [6], provide efficient implementations of these integer-arithmetics operations, optimized from both an algorithmic point of view (Fourier transform for large numbers) and implementation point of view (assembly for some architectures). Once you have this implementation of arithmetic operations for integers of arbitrary length, it is possible to implement a floating point number as a couple of two (arbitrary length) integers, that represent the mantissa and the exponent. Arithmetic operations are then implemented, making sure that each significant bit is represented in

[1] Tribute to Jonathan Shewchuk's "Conjugate Gradient without the agonizing pain"

the result, using dynamic allocation. This is the strategy taken by [13], a widely used programming library. At this point, one may directly use MPFR to implement the predicates, however, this makes the software dependent on MPFR and GMP, that are quite difficult to install/compile on non-Unix platforms. Note that MPFR is a very general software library, that implements not only basic algebraic operations, but also transcendental functions, accurate control of the rounding modes, ... In our context, to implement the geometric predicates, we only need three algebraic operations $(+, -, *)$. Note also that in the end, we only need to evaluate the sign (positive, zero or negative) of the result. Is there a way of having a more "minimalist" implementation, easy to compile, and with only the functionality that we need ?

A minimalist solution: expansion arithmetics: We now take again a look at Shewchuck's predicates[17], but this time, instead of taking the point of view of a user of the predicates, we need to understand the underlying principles, in order to be able to derive our owns predicates. Shewchuk's predicates use a form of exact arithmetics that is different from the arbitrary-length integers mentioned in the previous paragraph. The idea is again to represent a number as a set of components $x[0] \dots x[l-1]$, but this time each component is a floating point number, and the array represents the real number given by :

$$x = \sum_{i=0}^{l} x[i]$$

In addition, all the operations are specified to ensure an important property, that the sign of x is completely determined by the size of the last component:

$$\text{Sign}(x) = \text{Sign}(x[l-1])$$

In a nutshell, this property is enforced by ensuring that all the components are sorted by increasing exponents, and by ensuring that the exponents are sufficiently separated (or *non-overlapping*) so that the sum $x[0]+\dots...x[l-2]$ has a smaller magnitude than the least significant bit of $x[l-1]$. An array of floating point numbers that satisfies this condition is referred to as a *non-overlapping expansion*. Shewchuk provides C implementations for the following operations:

- **two-sum(***double*, *double***)** $\rightarrow (2)expansion$
- **grow-expansion(***double*, $(k)expansion$**)** $\rightarrow (k+1)expansion$
- **expansion-sum(**$(k)expansion$, $(l)expansion$**)** $\rightarrow (k+l)expansion$
- **scale-expansion(***double*, $(k)expansion$**)** $\rightarrow (2k)expansion$

that compute the sum of two doubles, add a double to an expansion, add two expansions and scale an expansion by a double (where the term $(k)expansion$ denotes an expansion of length k). He also derives the proof that each operation satisfies the non-overlapping property.

Starting from these 4 functions as building blocs, I implemented **expansion-product**, by recursively calling **scale-expansion** and **expansion-sum** (a method referred to as *distillation* in Shewchuk's article). This gives a class `expansion` with functions for the three algebraic operations $(+, -, *)$ and obtaining the sign at the end. To ensure that the code will remain efficient, even in a multithreaded environment, my `expansion` class allocates space on the stack instead of the heap, since there is (in most systems) a global lock on the heap, not well suited to multithreaded environments. In addition to the (low-level) `expansion` class, I implemented an (easier to use but less efficient) `expansion_nt` wrapper (expansion "number type"), that overloads the operators $(+, -, *)$. The resulting library is minimalist and compact (only a few hundred lines of C++ code, or 3000 lines if comments are counted). I experimented it on several OSes and architectures (PC Windows, PC Unix, Mac, Intel MIC 64-cores, Android phones with ARM v7 chips...). The C++ source code (BSD license) is available from the following address: `http://gforge.inria.fr/geogram`, together with a script that generate the C++ function of a predicate from its formula and its symbolic perturbation.

2. Computing a Restricted Voronoi Diagram

I shall now explain how to use the Predicate Construction Kit to (robustly) compute the intersection between a 3D Voronoi diagram and a tetrahedral mesh. This is the key component to several 3D meshing algorithm that we are currently investigating, such as : adaptive Voronoi meshing for flow simulation [10, 11], adaptive simplicial meshing [16, 15], and hex-dominant meshing [2]. In the context of 3d modelling for oil and gas exploration, the degenerate configurations are encountered very often, due to both the geometry of the input (thin layers, onlaps, ...) and the nature of the algorithms [3].

For instance, the algorithm in [10, 11] aims at generating a Voronoi diagram with cell facets that match some predefined interfaces (such as the boundaries between geological layers and geological faults), which will create degenerate configurations when computing the intersections between the cells and those interfaces. For this reason, in order to have a robust implementation of this software component, we have no other choice than carefully analyzing the predicates involved in the intersection algorithm, and developing a robust version of them. Before detailing the predicates, I shall present the general algorithm (next page) that computes the intersections :

Data: A tetrahedral mesh M and a set of points Y
Result: The intersection $\mathrm{Vor}_W(Y) \cap M$

S: Stack(couple(tet index, point index))
foreach *tetrahedron* $t \in M$ **do**
  if t is not marked **then**
    (1) $i \leftarrow i \mid \mathrm{Del}(\mathbf{y}_i) \cap t \neq \emptyset$
    Mark(t,i)
    Push(S, (t,i))
    while S is not empty **do**
      (t,i) $\leftarrow$ Pop(S)
      (2) P: Convex $\leftarrow \mathrm{Vor}(\mathbf{y}_i) \cap t$
      (3) do something with(P)
      (4) **foreach** j neighbor of i in P **do**
        if (t, j) is not marked **then**
          Mark(t, j); Push(S, (t, j))
        end
      end
      (5) **foreach** t' neighbor of t in P **do**
        if (t', i) is not marked **then**
          Mark(t', i); Push(S, (t', i))
        end
      end
    end
  end
end

Algorithm 1: Computing $\mathrm{Del}(Y) \cap M$ by propagation

The algorithm works by propagating simultaneously over the tetrahedra and the power cells. It traverses all the couples (t, i) such that the tetrahedron t has a non-empty intersection with the power cell of $\mathbf{y}_i$. (1): Propagation is initialized by starting from an arbitrary tetrahedron t and a point $\mathbf{y}_i$ that has a non-empty intersection between its power cell and t. I use the point $\mathbf{y}_i$ that minimizes its power distance $\|\mathbf{y}_i - .\|^2 - w_i$ to one of the vertices of t. (2): a tetrahedron t and a Voronoi cell $\mathrm{Del}\mathbf{y}_i$ can be both described as the intersection of half-spaces, as well as the intersection $t \cap \mathrm{Del}\mathbf{y}_i$, computed using re-entrant clipping (each half-space is removed iteratively). I use two version of the algorithm, a non-robust one that uses floating point arithmetics, and a robust one. (3) each intersection $P = t \cap \mathrm{Vor}\mathbf{y}_i$ is passed back to client code (that does what it needed to do with it). The convex P is illustrated in the (2d) figure 1 as the grayed area (in 3d, P is a convex polyhedron). The algorithm then propagates to both neighboring tetrahedra and points (points are neighbors if they are connected with a Delaunay edge). (4): each facet of P generated by a Voronoi cell facet triggers propagation to a neighboring point. In the 2d example of the figure, this corresponds to edges e_2 and e_3 that trigger propagation to points y_{j_1} and y_{j_2} respectively. (5): each portion of a facet of t that remains in P triggers a propagation to a neighboring tetrahedron t'. In the 2d example of Figure 1, this corresponds to edges e_1 and e_4 that trigger a propagation to triangles t_2 and t_1 respectively.

This algorithm is parallelized, by partitioning the mesh M into M_1, M_2, $\ldots M_{nb\_cores}$ and by computing in each thread $M_{thrd} \cap \mathrm{Del}(Y)$.

3. The Predicates

In the algorithm, the convex clipping operation (step 2) is the only moment where exact predicates are used. This operations takes as an input a tetrahedron and a Voronoi cell, and returns the intersection between them. It can be implemented in an iterative/reentrant manner: the Voronoi cell can be described as the intersection of half-spaces (the bisectors of the Delaunay edges). Each iteration will "chop-off" a half-space from the result. It means that the only needed predicate is a form of `orient3d`, that classifies a point with respect to a bisector. However, the setting is significantly more complicated, since starting from the second facet of the Voronoi cell, we need to classify points that were the result of previous intersections. This difficulty can be dealt with by replacing in each predicate the intersection points with their expression in function of the data (tetrahedral mesh vertices and Delaunay vertices). Depending on the involved points, several configurations can occur. Fortunately, the total number of configurations remains reasonably small (four configurations in 3d). The rest of this section introduces the relevant definitions, gives the general formula for all possible configurations in arbitrary dimension, together with the symbolic perturbations that allow

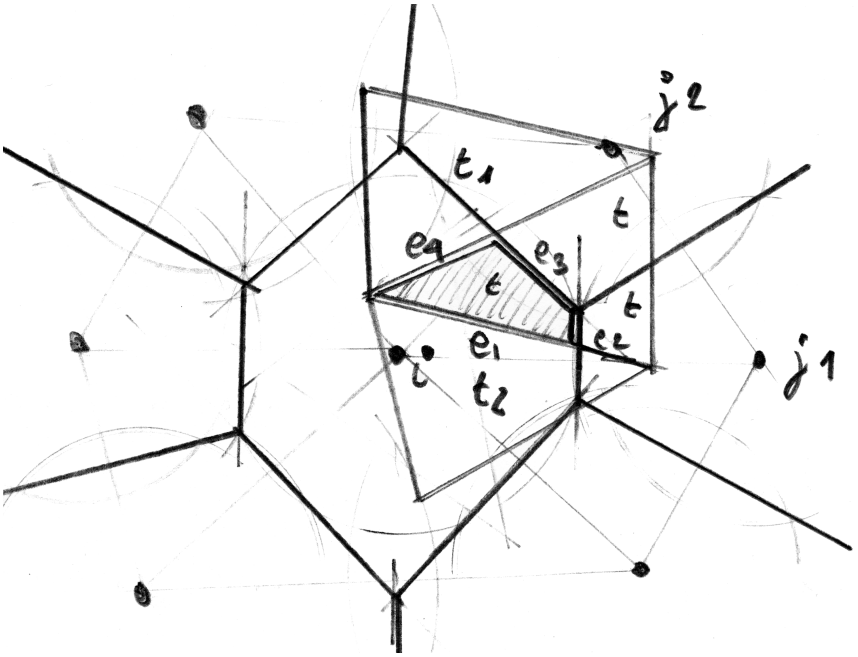


Figure 1: Computing the intersection between a power diagram and a tetrahedral mesh by propagation.

to escape degenerate configurations. The formulas in our our 3d setting are also given. The implementation of the predicates is freely available in PCK (`http://gforge.inria.fr/geogram`). The derivations are provided below for the interested reader.

Warning - boring material ahead!
The rest of this section may be skipped in a first reading.

3.1. Definitions

Definition 1. *The (additively)* weighted distance *(or power distance)* $d_W(\mathbf{p},\mathbf{q})$ *between a weighted point* $(\mathbf{p},w)$ *and a point* $\mathbf{q}$ *is defined by :*

$$d_W(\mathbf{p},\mathbf{q}) \stackrel{def}{=} \|\mathbf{p}-\mathbf{q}\|^2 - w$$

Definition 2. *The* bisector $\Pi_W(\mathbf{p}_0,\mathbf{p}_1)$ *of the pair of weighted points* $(\mathbf{p}_0,w_0),(\mathbf{p}_1,w_1)$ *is defined by :*

$$\begin{array}{rcll}
\Pi_W(\mathbf{p}_0,\mathbf{p}_1) & \stackrel{def}{=} & \{\mathbf{q} & | \quad d_W(\mathbf{p}_0,\mathbf{q}) = d_W(\mathbf{p}_1,\mathbf{q})\} \\
& = & \{\mathbf{q} & | \quad \|\mathbf{p}_1-\mathbf{q}\|^2 - \|\mathbf{p}_0-\mathbf{q}\|^2 + w_0 - w_1 = 0\} \\
& = & \{\mathbf{q} & | \quad (\mathbf{p}_1-\mathbf{q}+\mathbf{p}_0-\mathbf{q})^t(\mathbf{p}_1-\mathbf{q}-\mathbf{p}_0+\mathbf{q}) + w_0 - w_1 = 0\} \\
& = & \{\mathbf{q} & | \quad \|\mathbf{p}_1-\mathbf{p}_0\|^2 - 2(\mathbf{q}-\mathbf{p}_0)^t(\mathbf{p}_1-\mathbf{p}_0) + w_0 - w_1 = 0\}
\end{array}$$

Observation 1. *The intersection* $\mathbf{q}$ *between* d *bisectors* $\Pi_w(\mathbf{p}_0,\mathbf{p}_1),\dots\Pi_w(\mathbf{p}_0,\mathbf{p}_d)$ *and a* d*-simplex* $\mathbf{\Delta}(\mathbf{q}_0,\mathbf{q}_1\dots\mathbf{q}_d)$ *can be obtained in barycentric form as* $\mathbf{q}=\sum_{i=0}^{d}\lambda_i\mathbf{q}_i$*. The barycentric coordinates* (λ_i) *are given by :*

$$\Delta\begin{bmatrix}\lambda_0\\ \lambda_1\\ \vdots\\ \lambda_d\end{bmatrix} \quad = \quad B'\left(\begin{bmatrix}l_1\\ \vdots\\ l_d\end{bmatrix}+\begin{bmatrix}-w1+w0\\ \vdots\\ -w_d+w_0\end{bmatrix}\right)+b$$

where: $\underset{\substack{1\le i\le d\\ 0\le j\le d}}{a_{i,j}}=2(\mathbf{q}_j-\mathbf{p}_0)^t(\mathbf{p}_i-\mathbf{p}_0) \quad ; \quad \underset{1\le i\le d}{l_i}=\|\mathbf{p}_i-\mathbf{p}_0\|^2$

$$A_{d+1\times d+1}=\left(\begin{array}{ccc}1&\dots&1\\ \hline a_{1,0}&\dots&a_{1,d}\\ \vdots&&\vdots\\ a_{d,0}&\dots&a_{d,d}\end{array}\right) \quad ; \quad \Delta=|A| \quad ; \quad B=\Delta A^{-1}=\left(\begin{array}{c|c}b & B'\\ \ & \ \end{array}\right)$$

Proof.

The intersection $\mathbf{q}$ satisfies : $\begin{cases}\sum_{i=0}^{d}\lambda_i=1 \quad ; \quad \mathbf{q}=\sum_{i=0}^{d}\lambda_i\mathbf{q}_i\\ \forall 1\le i\le d, \mathbf{q}\in\Pi_W(\mathbf{p}_0,\mathbf{p}_i)\end{cases}$

in matrix form, using the equation of $\Pi_W(\mathbf{p}_0,\mathbf{p}_i)$ (Definition 2) :

$$\begin{pmatrix}1&\dots&1\\ 2(\mathbf{q}_0-\mathbf{p}_0)^t(\mathbf{p}_1-\mathbf{p}_0)&\dots&2(\mathbf{q}_d-\mathbf{p}_0)^t(\mathbf{p}_1-\mathbf{p}_0)\\ \vdots&&\vdots\\ 2(\mathbf{q}_0-\mathbf{p}_0)^t(\mathbf{p}_d-\mathbf{p}_0)&\dots&2(\mathbf{q}_d-\mathbf{p}_0)^t(\mathbf{p}_d-\mathbf{p}_0)\end{pmatrix}\begin{bmatrix}\lambda_0\\ \vdots\\ \lambda_d\end{bmatrix} \quad = \quad \begin{bmatrix}1\\ \|\mathbf{p}_1-\mathbf{p}_0\|^2+w_0-w_1\\ \|\mathbf{p}_2-\mathbf{p}_0\|^2+w_0-w_2\\ \vdots\\ \|\mathbf{p}_d-\mathbf{p}_0\|^2+w_0-w_d\end{bmatrix}$$

or with the notations above : $\underbrace{\begin{pmatrix}1&1&\dots&1\\ a_{1,0}&a_{1,1}&&a_{1,d}\\ \vdots&&&\vdots\\ a_{d,0}&a_{d,1}&&a_{d,d}\end{pmatrix}}_{A}\begin{bmatrix}\lambda_0\\ \lambda_1\\ \vdots\\ \lambda_d\end{bmatrix} \quad = \quad \begin{bmatrix}1\\ l_1\\ \vdots\\ l_d\end{bmatrix}+\begin{bmatrix}0\\ -w_1+w_0\\ -w_2+w_0\\ \vdots\\ -w_d+w_0\end{bmatrix}$

thus we have :

$$\Delta \begin{bmatrix} \lambda_0 \\ \lambda_1 \\ \vdots \\ \lambda_d \end{bmatrix} = \underbrace{A^{-1}}_{B} \left(\begin{bmatrix} 1 \\ l_1 \\ \vdots \\ l_d \end{bmatrix} + \begin{bmatrix} 0 \\ -w_1 + w_0 \\ \vdots \\ -w_d + w_0 \end{bmatrix} \right)$$

$$= \left(\begin{array}{c|c} b & B' \end{array} \right) \begin{bmatrix} 1 \\ \hline l_1 - w_1 + w_0 \\ \vdots \\ l_d - w_d + w_0 \end{bmatrix} = B' \left(\begin{bmatrix} l_1 \\ \vdots \\ l_d \end{bmatrix} + \begin{bmatrix} -w_1 + w_0 \\ \vdots \\ -w_d + w_0 \end{bmatrix} \right) + b$$

□

3.2. General Formulation of the Predicates

Definition 3. *The predicates* $\text{side}_1(\mathbf{p}_0, \mathbf{p}_1, \mathbf{q})$ *determines the position of* $\mathbf{q}$ *relative to the bisector* $\Pi_W(\mathbf{p}_0, \mathbf{p}_1)$ *:*

$$\text{side}_1(\mathbf{p}_0, \mathbf{p}_1, \mathbf{q}) \text{ returns } \begin{cases} +1 & \text{if } d_W(\mathbf{q}, \mathbf{p}_0) < d_W(\mathbf{q}, \mathbf{p}_1) \\ 0 & \text{if } d_W(\mathbf{q}, \mathbf{p}_0) = d_W(\mathbf{q}, \mathbf{p}_1) \\ -1 & \text{if } d_W(\mathbf{q}, \mathbf{p}_0) > d_W(\mathbf{q}, \mathbf{p}_1) \end{cases}$$

Using the definition of d_W *(Definition 1),* side_1 *is given by :*

$$\begin{aligned} \text{side}_1(\mathbf{p}_0, \mathbf{p}_1, \mathbf{q}) &\stackrel{def}{=} d_w(\mathbf{p}_1, \mathbf{q}) - d_w(\mathbf{p}_0, \mathbf{q}) \\ &= Sign\left(\|\mathbf{p}_1 - \mathbf{p}_0\|^2 - 2(\mathbf{q} - \mathbf{p}_0)^t(\mathbf{p}_1 - \mathbf{p}_0) + w_0 - w_1 \right) \end{aligned}$$

Main Term and Simulation of Simplicity for $\text{side}_1(\mathbf{p}_0, \mathbf{p}_1, \mathbf{q})$**.**

- Main term: $\text{Sign}(l - a)$,
  where $a = 2(\mathbf{q} - \mathbf{p}_0)^t(\mathbf{p}_1 - \mathbf{p}_0)$ and $l = \|\mathbf{p}_1 - \mathbf{p}_0\|^2$
- Perturbation of $\mathbf{p}_0$: +1
- Perturbation of $\mathbf{p}_1$: -1

Definition 4. *The predicates* $\text{side}_{d+1}(\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_{d+1}, \mathbf{q}_0, \mathbf{q}_1, \ldots, \mathbf{q}_d)$ *determines the position of* $\mathbf{q} = \Pi_W(\mathbf{p}_0, \mathbf{p}_1) \cap \Pi_W(\mathbf{p}_0, \mathbf{p}_2) \ldots \cap \Pi_W(\mathbf{p}_0, \mathbf{p}_d) \cap \mathbf{\Delta}(\mathbf{q}_0, \ldots, \mathbf{q}_d)$ *relative to* $\Pi_W(\mathbf{p}_0, \mathbf{p}_{d+1})$ *:*

$$\text{side}_{d+1}(\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_{d+1}, \mathbf{q}_0, \mathbf{q}_1, \ldots, \mathbf{q}_d) \stackrel{def}{=} \text{side}_1(\mathbf{p}_0, \mathbf{p}_{d+1}, \mathbf{q})$$

where $\mathbf{q} = \Pi_W(\mathbf{p}_0, \mathbf{p}_1) \cap \Pi_W(\mathbf{p}_0, \mathbf{p}_2) \ldots \cap \Pi_W(\mathbf{p}_0, \mathbf{p}_d) \cap \mathbf{\Delta}(\mathbf{q}_0, \ldots, \mathbf{q}_d)$

Main Term and Simulation of Simplicity for $\text{side}_{d+1}(\mathbf{p}_0, \ldots, \mathbf{p}_{d+1}, \mathbf{q}_0, \ldots, \mathbf{q}_{d+1})$**.**

- Main term: $\text{Sign}\left(\Delta l_{d+1} - \begin{bmatrix} a_{d+1,0} \ldots a_{d+1,d} \end{bmatrix} \left(B' \begin{bmatrix} l_1 \\ \vdots \\ l_d \end{bmatrix} + b \right)\right) \times \text{Sign}(\Delta)$
- Perturbation of $\mathbf{p}_0$: $\text{Sign}\left(-\sum_{i=0}^{d}\sum_{j=1}^{d} b'_{i,j} a_{d+1,i} + \Delta\right) \times \text{Sign}(\Delta)$
- Perturbation of $\mathbf{p}_i; 1 \leq i \leq d$: $\text{Sign}\left(\sum_{j=0}^{d} (a_{d+1,j} b'_{j,i})\right) \times \text{Sign}(\Delta)$
- Perturbation of $\mathbf{p}_{d+1}$: -1

where:

$$\begin{cases} a_{\substack{i,j \\ 1\leq i\leq d+1 \\ 0\leq j\leq d}} & = & 2(\mathbf{q}_j - \mathbf{p}_0)^t(\mathbf{p}_i - \mathbf{p}_0) \\ l_{\substack{i \\ 1\leq i\leq d}} & = & \|\mathbf{p}_i - \mathbf{p}_0\|^2 \end{cases} \quad ; \quad A_{d+1\times d+1} = \left(\begin{array}{ccc} 1 & \ldots & 1 \\ \hline a_{1,0} & \ldots & a_{1,d} \\ \vdots & & \vdots \\ a_{d,0} & \ldots & a_{d,d} \end{array}\right)$$

$$\Delta = |A| \quad ; \quad B_{d+1\times d+1} = \Delta A^{-1} = \left(\begin{array}{c|c} b & B' \end{array}\right) = \left(\begin{array}{c|c} b_{0,0} & b'_{0,1} \ldots b'_{0,d} \\ b_{1,0} & b'_{1,1} \ldots b'_{1,d} \\ \vdots & \vdots \\ b_{d,0} & b'_{d,1} \ldots b'_{d,d} \end{array}\right)$$

Proof. To obtain the expression of side_{d+1}, we inject the expression $\Delta(\mathbf{q} - \mathbf{p}_0) = \sum_{i=0}^{d} \Delta\lambda_i(\mathbf{q}_i - \mathbf{p}_0)$ into $\text{side}_1(\mathbf{p}_0, \mathbf{p}_{d+1}, \Delta\mathbf{q}) \times \text{Sign}(\Delta)$:

$\text{side}_{d+1}(\mathbf{p}_0, \ldots, \mathbf{p}_{d+1}, \mathbf{q}_0, \ldots, \mathbf{q}_{d+1}) =$

$$\text{Sign}\left(\Delta \underbrace{\|\mathbf{p}_{d+1} - \mathbf{p}_0\|^2}_{\stackrel{\text{def}}{=} l_{d+1}} - \sum_{i=0}^{d} \Delta\lambda_i \underbrace{2(\mathbf{q}_i - \mathbf{p}_0)^t(\mathbf{p}_{d+1} - \mathbf{p}_0)}_{\stackrel{\text{def}}{=} a_{d+1,i}} + \Delta(w_0 - w_{d+1})\right) \times \text{Sign}(\Delta) =$$

$$\text{Sign}\left(\Delta l_{d+1} - \sum_{i=0}^{d} \Delta\lambda_i a_{d+1,i} + \Delta(w_0 - w_{d+1})\right) \times \text{Sign}(\Delta) =$$

$$\text{Sign}\left(\Delta l_{d+1} - \begin{bmatrix} a_{d+1,0} \ldots a_{d+1,d} \end{bmatrix} \left(B' \begin{bmatrix} l_1 - w_1 + w_0 \\ \vdots \\ l_d - w_d + w_0 \end{bmatrix} + b \right) + \Delta(w_0 - w_{d+1})\right) \times \text{Sign}(\Delta)$$

Each perturbation associated with each $\mathbf{p}_i$ is retrieved as the term that has w_i as a factor. □

I now make a simple observation that proves the robustness of the perturbation, i.e. its ability to escape from any singular configuration :

Observation 2. *There is always at least one term in the Taylor expansion of* $\text{side}_{d+1}()$ *perturbed by* $w_i = \epsilon^i$ *that is non-zero.*

Proof. The term associated with $\mathbf{p}_{d+1}$ is $-\text{Sign}(\Delta) \times \text{Sign}(\Delta) = -1$. □

3.3. The Predicates for Surfacic and Volumetric Meshing

For surfacic meshing, we need the expression of side_1 (point $\mathbf{q}$ is directly given), side_2 (point $\mathbf{q}$ is defined as the intersection between a bisector and a segment) and side_3 (point $\mathbf{q}$ is defined as the intersection between two bisectors and a triangle). For volumetric meshing, in addition we need the expression of side_4 (point $\mathbf{q}$ is defined as the intersection between three bisectors and a tetrahedron).

Definition 5. *The predicate* $\text{side}_2(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \mathbf{q}_0, \mathbf{q}_1)$ *determines the position of* $\mathbf{q} = \Pi_W(\mathbf{p}_0, \mathbf{p}_1) \cap [\mathbf{q}_0, \mathbf{q}_1]$ *relative to* $\Pi_W(\mathbf{p}_0, \mathbf{p}_2)$ *:*

$$\text{side}_2(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \mathbf{q}_0, \mathbf{q}_1) = \text{side}_1(\mathbf{p}_0, \mathbf{p}_1, \mathbf{q})$$

$$\textit{where } \mathbf{q} = \Pi_W(\mathbf{p}_0, \mathbf{p}_1) \cap [\mathbf{q}_0, \mathbf{q}_1]$$

Main Term and Simulation of Simplicity for $\text{side}_2(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \mathbf{q}_0, \mathbf{q}_1)$**.**

- Main term: $\text{Sign}\left(\Delta l_2 - \begin{bmatrix} a_{2,0} & a_{2,1} \end{bmatrix} \begin{bmatrix} -l_1 + a_{1,1} \\ l_1 - a_{1,0} \end{bmatrix}\right) \times \text{Sign}(\Delta)$
- Perturbation of $\mathbf{p}_0$: $\text{Sign}(\Delta + a_{2,0} - a_{2,1}) \times \text{Sign}(\Delta)$
- Perturbation of $\mathbf{p}_1$: $\text{Sign}(-a_{2,0} + a_{2,1}) \times \text{Sign}(\Delta)$
- Perturbation of $\mathbf{p}_2$: -1

where $a_{i,j} = 2(\mathbf{p}_i - \mathbf{p}_0)^t(\mathbf{q}_j - \mathbf{p}_0)$; $l_i = \|\mathbf{p}_i - \mathbf{p}_0\|^2$; $\Delta = a_{1,1} - a_{1,0}$

Proof. These expressions are obtained by specializing the definition of side_{d+1} (Definition 3) with $d + 1 = 2$. The A and B matrices are then as follows :

$$A = \begin{pmatrix} 1 & 1 \\ a_{1,0} & a_{1,1} \end{pmatrix} \quad ; \quad B = \begin{pmatrix} a_{1,1} & -1 \\ -a_{1,0} & 1 \end{pmatrix} \quad ; \quad b = \begin{bmatrix} a_{1,1} \\ -a_{1,0} \end{bmatrix} \quad ; \quad B' = \begin{pmatrix} -1 \\ 1 \end{pmatrix}$$

□

Definition 6. *The predicate* $\text{side}_3(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \mathbf{q}_0, \mathbf{q}_1, \mathbf{q}_2)$ *determines the position of* $\mathbf{q} = \Pi_W(\mathbf{p}_0, \mathbf{p}_1) \cap \Pi_W(\mathbf{p}_0, \mathbf{p}_2) \cap \mathbf{\Delta}(\mathbf{q}_0, \mathbf{q}_1, \mathbf{q}_2)$ *relative to* $\Pi_W(\mathbf{p}_0, \mathbf{p}_3)$ *:*

$$\text{side}_3(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \mathbf{q}_0, \mathbf{q}_1, \mathbf{q}_2) = \text{side}_1(\mathbf{p}_0, \mathbf{p}_1, \mathbf{q})$$

$$\textit{where } \mathbf{q} = \Pi_W(\mathbf{p}_0, \mathbf{p}_1) \cap \Pi_W(\mathbf{p}_0, \mathbf{p}_2) \cap \mathbf{\Delta}(\mathbf{q}_0, \mathbf{q}_1, \mathbf{q}_2)$$

Main Term and Simulation of Simplicity for $\mathrm{side}_3(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \mathbf{q}_0, \mathbf{q}_1, \mathbf{q}_2)$**.**

- Main term: $\mathrm{Sign}\left(\Delta l_3 - \begin{bmatrix} a_{3,0} & a_{3,1} & a_{3,2} \end{bmatrix} \Delta \begin{bmatrix} \lambda_0 \\ \lambda_1 \\ \lambda_2 \end{bmatrix}\right) \times \mathrm{Sign}(\Delta)$
- Perturbation of $\mathbf{p}_0$:
  $\mathrm{Sign}\left(\Delta - a_{3,0}(b'_{0,1} + b'_{0,2}) - a_{3,1}(b'_{1,1} + b'_{1,2}) - a_{3,2}(b'_{2,1} + b'_{2,2})\right) \times \mathrm{Sign}(\Delta)$
- Perturbation of $\mathbf{p}_1$: $\mathrm{Sign}(a_{3,0}b'_{0,1} + a_{3,1}b'_{1,1} + a_{3,2}b'_{2,1}) \times \mathrm{Sign}(\Delta)$
- Perturbation of $\mathbf{p}_2$: $\mathrm{Sign}(a_{3,0}b'_{0,2} + a_{3,1}b'_{1,2} + a_{3,2}b'_{2,2}) \times \mathrm{Sign}(\Delta)$
- Perturbation of $\mathbf{p}_3$: -1

where $a_{i,j} = 2(\mathbf{p}_i - \mathbf{p}_0)^t(\mathbf{q}_j - \mathbf{p}_0) \quad ; \quad l_i = \|\mathbf{p}_i - \mathbf{p}_0\|^2 \quad ; \quad \Delta = b_{0,0} + b_{1,0} + b_{2,0}$

$$B = \Delta A^{-1} = \begin{pmatrix} \begin{vmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{vmatrix} & (a_{2,1} - a_{2,2}) & (a_{1,2} - a_{1,1}) \\ \begin{vmatrix} a_{1,2} & a_{1,0} \\ a_{2,2} & a_{2,0} \end{vmatrix} & (a_{2,2} - a_{2,0}) & (a_{1,0} - a_{1,2}) \\ \begin{vmatrix} a_{1,0} & a_{1,1} \\ a_{2,0} & a_{2,1} \end{vmatrix} & (a_{2,0} - a_{2,1}) & (a_{1,1} - a_{1,0}) \end{pmatrix} = \left(\begin{array}{c|cc} b_{0,0} & b'_{0,1} & b'_{0,2} \\ b_{1,0} & b'_{1,1} & b'_{1,2} \\ b_{2,0} & b'_{2,1} & b'_{2,2} \end{array}\right)$$

Definition 7. *The predicate* $\mathrm{side}_4(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \mathbf{p}_4, \mathbf{q}_0, \mathbf{q}_1, \mathbf{q}_2, \mathbf{q}_3)$ *determines the position of* $\mathbf{q} = \Pi_W(\mathbf{p}_0, \mathbf{p}_1) \cap \Pi_W(\mathbf{p}_0, \mathbf{p}_2) \cap \Pi_W(\mathbf{p}_0, \mathbf{p}_3) \cap \mathbf{\Delta}(\mathbf{q}_0, \mathbf{q}_1, \mathbf{q}_2, \mathbf{q}_3)$ *relative to* $\Pi_W(\mathbf{p}_0, \mathbf{p}_4)$ *:*

$$\mathrm{side}_3(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \mathbf{p}_4, \mathbf{q}_0, \mathbf{q}_1, \mathbf{q}_2, \mathbf{q}_3) = \mathrm{side}_1(\mathbf{p}_0, \mathbf{p}_1, \mathbf{q})$$

where $\mathbf{q} = \Pi_W(\mathbf{p}_0, \mathbf{p}_1) \cap \Pi_W(\mathbf{p}_0, \mathbf{p}_2) \cap \Pi_W(\mathbf{p}_0, \mathbf{p}_3) \cap \mathbf{\Delta}(\mathbf{q}_0, \mathbf{q}_1, \mathbf{q}_2, \mathbf{q}_3)$

The expression of the main term and perturbations is not given here for keeping the length of the article reasonable. The reader may refer to the companion PCK (Predicate Construction Kit) sourcecode.

Note that dimension 3 is a particular case (i.e. the dimension of the ambient space coincides with the dimension of the simplex). In this case, three bisectors in generic position are sufficient to determine a point in $3d$ (the simplex is no-longer needed). Interestingly, computations are much simpler in that case. For this reason, we introduce the following version of the predicate :

Definition 8. *The predicate* $\mathrm{side}_4^{3d}(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \mathbf{p}_4)$ *determines the position of* $\mathbf{q} = \Pi_W(\mathbf{p}_0, \mathbf{p}_1) \cap \Pi_W(\mathbf{p}_0, \mathbf{p}_2) \cap \Pi_W(\mathbf{p}_0, \mathbf{p}_3)$ *relative to* $\Pi_W(\mathbf{p}_0, \mathbf{p}_4)$ *:*

$$\text{side}_3^{3d}(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \mathbf{p}_4) = \text{side}_1(\mathbf{p}_0, \mathbf{p}_1, \mathbf{q})$$

$$\textit{where } \mathbf{q} = \Pi_W(\mathbf{p}_0, \mathbf{p}_1) \cap \Pi_W(\mathbf{p}_0, \mathbf{p}_2) \cap \Pi_W(\mathbf{p}_0, \mathbf{p}_3)$$

The predicate side_4^{3d} is equivalent to insphere. See [5] for more details. Using similar derivations (based on Kramer's formula for solving linear systems), we obtain :

Main Term and Simulation of Simplicity for $\text{side}_4^{3d}(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \mathbf{p}_4)$**.**

- Main term:
$$\text{Sign}(\Delta) \times \text{Sign}\begin{pmatrix} \begin{vmatrix} (\mathbf{p}_1 - \mathbf{p}_0)^t & -\|\mathbf{p}_1 - \mathbf{p}_0\|^2 \\ (\mathbf{p}_2 - \mathbf{p}_0)^t & -\|\mathbf{p}_2 - \mathbf{p}_0\|^2 \\ (\mathbf{p}_3 - \mathbf{p}_0)^t & -\|\mathbf{p}_3 - \mathbf{p}_0\|^2 \\ (\mathbf{p}_4 - \mathbf{p}_0)^t & -\|\mathbf{p}_4 - \mathbf{p}_0\|^2 \end{vmatrix} \end{pmatrix} \text{ where } \Delta = \begin{vmatrix} (\mathbf{p}_1 - \mathbf{p}_0)^t \\ (\mathbf{p}_2 - \mathbf{p}_0)^t \\ (\mathbf{p}_3 - \mathbf{p}_0)^t \end{vmatrix}$$

- Perturbation for $\mathbf{p}_0$:
$$\text{Sign}(\Delta) \times \text{Sign}\begin{pmatrix} -\begin{vmatrix} (\mathbf{p}_2 - \mathbf{p}_0)^t \\ (\mathbf{p}_3 - \mathbf{p}_0)^t \\ (\mathbf{p}_4 - \mathbf{p}_0)^t \end{vmatrix} + \begin{vmatrix} (\mathbf{p}_1 - \mathbf{p}_0)^t \\ (\mathbf{p}_3 - \mathbf{p}_0)^t \\ (\mathbf{p}_4 - \mathbf{p}_0)^t \end{vmatrix} - \begin{vmatrix} (\mathbf{p}_1 - \mathbf{p}_0)^t \\ (\mathbf{p}_2 - \mathbf{p}_0)^t \\ (\mathbf{p}_4 - \mathbf{p}_0)^t \end{vmatrix} \end{pmatrix}$$

- Perturbation for $\mathbf{p}_1$: $\text{Sign}(\Delta) \times \text{Sign}\begin{pmatrix} \begin{vmatrix} (\mathbf{p}_2 - \mathbf{p}_0)^t \\ (\mathbf{p}_3 - \mathbf{p}_0)^t \\ (\mathbf{p}_4 - \mathbf{p}_0)^t \end{vmatrix} \end{pmatrix}$

- Perturbation for $\mathbf{p}_2$: $-\text{Sign}(\Delta) \times \text{Sign}\begin{pmatrix} \begin{vmatrix} (\mathbf{p}_1 - \mathbf{p}_0)^t \\ (\mathbf{p}_3 - \mathbf{p}_0)^t \\ (\mathbf{p}_4 - \mathbf{p}_0)^t \end{vmatrix} \end{pmatrix}$

- Perturbation for $\mathbf{p}_3$: $\text{Sign}(\Delta) \times \text{Sign}\begin{pmatrix} \begin{vmatrix} (\mathbf{p}_1 - \mathbf{p}_0)^t \\ (\mathbf{p}_2 - \mathbf{p}_0)^t \\ (\mathbf{p}_4 - \mathbf{p}_0)^t \end{vmatrix} \end{pmatrix}$

- Perturbation for $\mathbf{p}_4$: -1

4. Results and conclusions

To test the algorithm, I first tryed very degenerate configurations, such as the two ones depicted in Figures 2 and 3. In the 2D configuration shown in Figure 2, the Voronoi diagram and the mesh coincide exactly, thus causing singularities everywhere. The symbolic perturbations successfully escape from these singularities (as predicted by Observation 2). The 3D configuration shown in Figure 3 consists in computing the intersection between the Voronoi diagram of two parallel grids of points and a square equidistant

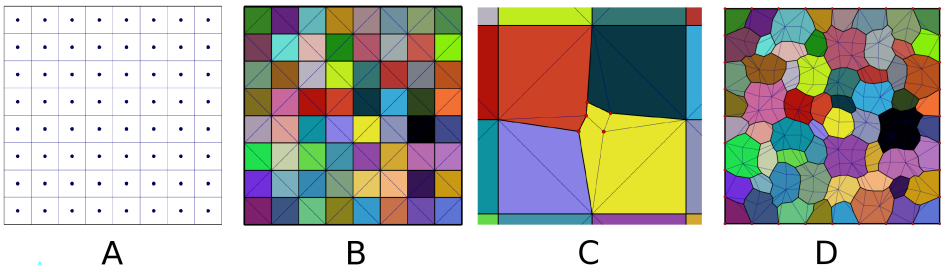


Figure 2: A very degenerate example. A: computing the intersection between the Voronoi diagram of the black points and the grid. B: the result, where the Voronoi edges coincide everywhere with the grid. C: "exploded view" of the combinatorics that was generated by the symbolic perturbation (the five red vertices are at the same geometric location, and all the three triangles and two quads have zero area). D: exploded view of the combinatorics of the whole mesh.

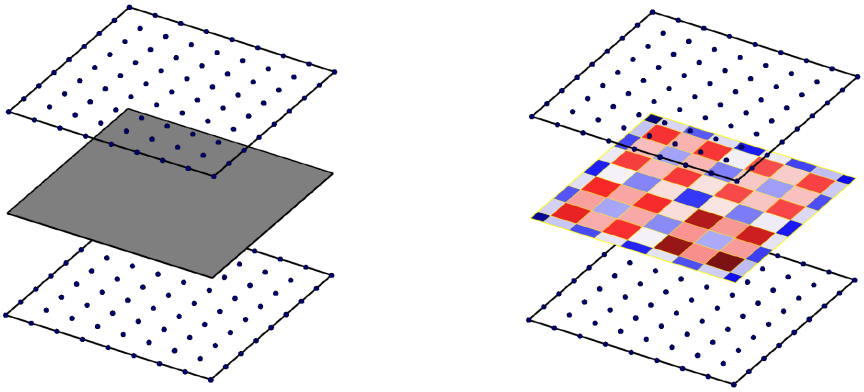


Figure 3: Another degenerate example in 3D. The Voronoi cells of two parallel grid of points are parallelepipeds that meet on a plane equidistant to both planes.

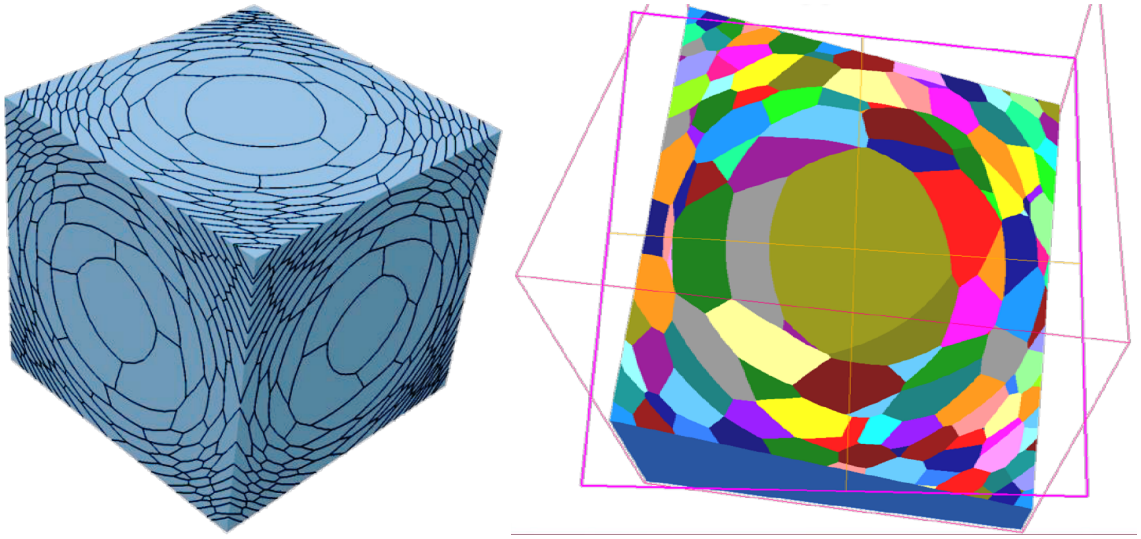


Figure 4: Using restricted Voronoi diagram in 6d to compute an anisotropic Voronoi diagram.

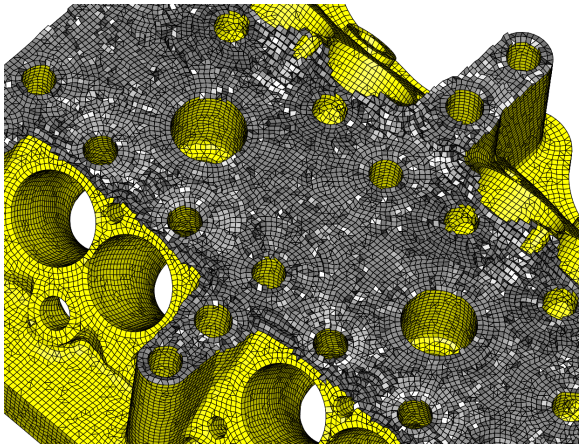


Figure 5: Hex-dominant meshing of a mechanical part ("crash-test" for the algorithm).

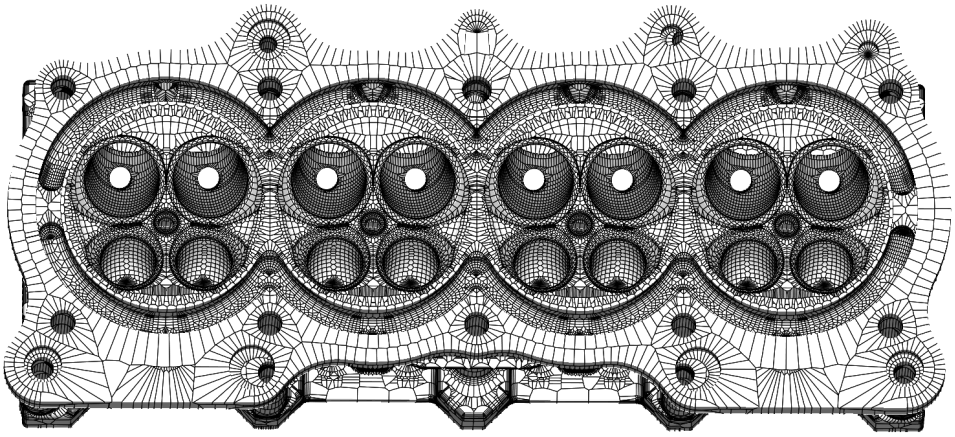


Figure 6: Restricted Voronoi diagram with a mechanical part ("crash-test" for the algorithm).

Figure 7: Interpolation between an armadillo and a sphere computed by semi-discrete optimal transport.

to both grids of points (left), in such a way that many Voronoi facets fall exactly on the square (right).

I experimented the algorithm in different application settings, comprising mesh generation with both Voronoi diagrams, Voronoi diagrams with anisotropy, Voronoi diagrams with the L_p metric (for hex-dominant mesh generation).

Anisotropic mesh generation :. Making the techniques robust to high anisotropy is very important for several simulation techniques, since it allows dramatically reducing the number of mesh elements while preserving the accuracy of the simulation. Figure 4 shows the first results obtained with an anisotropic Voronoi diagram. To my knowledge, this is the first time an Anisotropic Voronoi diagram can be computed in 3d. It uses the idea of replacing anisotropy with a higher-dimensional embedding [14]. In this specific case, I used an explicit 6d embedding, designed to create the variation of anisotropy shown on the figure. To represent anisotropy, the method uses a fine 150^3 grid embedded in 6d space, and the dimension-independent expression of the predicates.

hex-dominant meshing :. The L_p-CVT algorithm [8] provides a possible means of generating hex-dominant meshes. However it creates degenerate configuration in the Delaunay triangulation, with many co-spherical vertices, leading to difficulties in the predicates. The symbolic perturbation successfully escapes the singularities (as proved in theory), resulting in an algorithm that is robust in both the surfacic case (Figure 5) and the volumetric case (Figure 6). The shown mechanical part is a "crash test" for the algorithm: in addition with the degeneracies caused by Voronoi vertices aligned on a hex grid, it introduces coplanar surfaces and corcircular vertices in the mesh, an interesting benchmark for the predicates.

optimal transport in 3d :. Optimal Transport is an active research topic in mathematics [18]. From a practical point of view, it is a promising way of designing new numerical solution mechanisms for some Partial Derivative Equations. Initially, Optimal Transport studies the following optimization problem, that tries to find the "cheapest" application T that deforms a source measure μ onto a target measure ν (Monge's problem):

$$\min_T \int_\Omega c(\mathbf{x}, T(\mathbf{x}))d\mu \qquad \text{subject to: } \nu = T\sharp\mu$$

where μ and ν are two probability measures on Ω and where $c(\mathbf{x}, \mathbf{y})$ is the "transport cost" (e.g., $\|\mathbf{y} - \mathbf{x}\|^2$ in L_2 optimal transport) and where the constraint $\nu = T\sharp\mu$ means that T "preserves mass" (T pushes μ onto ν). Kantorovich introduces a relaxation of the problem that restores its symmetry and that exhibits some regularity in the dual formulation, thus allowing

to study existence and uniqueness, depending on the cost function $c(.,.)$. The value of the objective function is a *distance between measures*, called the Wasserstein measure $W(\mu,\nu)$, that can be used to construct a *metric space*, thus providing means of interpolating between different measures.

In the specific case where μ has a density u (i.e., $\mu(B) = \int_B u(x)dx$) and $\nu = \sum_{i=1}^{k} \lambda_i \delta_{\underline{\mathbf{y}}_i}$ is a sum of Dirac masses, Aurenhammer *et.al.* remarked that the pre-image of the Dirac masses correspond to a power diagram [1]. The same result can be also directly obtained by specializing Optimal Transport theory (Kantorovich duality) to the semi-discrete case. Interestingly, the derivations of the proof in [1] can be translated into an algorithm : the weights $\mathbf{W} = (w_i)_1^k$ of the power diagram that determines the optimal transport can be determined as the (unique) maximum of a concave function $f(\mathbf{W})$. However, the resulting algorithm has a slow speed of convergence. For this reason, Mérigot proposed an algorithm and experimented it in 2d [9]. To be generalized to the 3d case, the method needs to robustly compute the intersection between a power diagram and a tetrahedral mesh. This can be implemented using my PCK system, as I did in [7] (see Figure 7).

compatibility with other programming libraries :. the exact precision number type in my PCK system comes with two different APIs: the low-level one, that is most efficient, and the higher-level one, easier to use (but slower), that provides a "number type" `expansion_nt` with operator overloads (`+,-,*` and sign computation). We experimenting "grafting" it into CGAL. For standard datasets, difference of timings could not be distinguished, this is because arithmetic filters find the answer in most cases (in general, far less than 1 percent of the calls requires exact precision). To further analyse performance, we compared the performance of the Delaunay triangulation with a degenerate dataset[2], with many points located on a regular grid, that nearly triggers the exact precision all the time. With CGAL's default arbitrary precision number type it takes 1.2 seconds, and with PCK it takes 1.6 seconds, therefore the price to pay for the additional flexibility of PCK seems to stay reasonable. Moreover, we can probably still gain performance by writing some template specializations that avoid converting between number types and that use the more efficient lower-level API of PCK.

conclusions :. computing the intersection between a 3d Voronoi diagram and a tetrahedral mesh requires some "exotic" predicates. They are not theoretically challenging, but their algebraic expression is quite involved. Transforming these predicates into a practical implementation required to develop some new unconventional software tools (and to use some existing ones). I made the PCK software tools presented here available under

[2] with Andreas Fabri, during the 2015 CGAL meeting in Nancy

the BSD open-source license. The PCK code is fully documented, simple, portable and easy to compile. I think that the methodology presented here can be reused for a wider set of predicates. This leads to several suggestions for future work and open questions :

- In its current state, the PCK code generation tool is a set of macros and scripts. It would be possible to write a cleaner (and more powerful) version completely in C++ (as an extension of FPG for instance);
- is the code completely correct ? It was extensively tested, but testing does not show the absence of bugs (it shows their presence, as it did several times in this specific case !). To gain better certification, is it possible to use formal proof assistant tools ? Such tools, such as Coq/Gasper, can analyze numerical code and prove some properties. It is so easy to do errors in this kind of code, that my point of view is that every possible way of certification should be used (experimental, theoretical, automatic formal verification).

Acknowledgements :. This research was funded by the following grants: ERC GOODSHAPE (ERC-StG-205693), ANR MORPHO and ANR BECASIM. I wish to thank the organizers of the International Meshing Roundtable for inviting me to give the talk that was at the origin of this article. I wish also to thank Jonathan Shewchuk for discussions, Andreas Fabri for a nice coding day in CGAL, and the anonymous reviewers for their feedback that helped to improve the article.

Appendix A. PCK source code for side_3:

```
Sign predicate(side3)(
    point(p0), point(p1), point(p2), point(p3),
    point(q0), point(q1), point(q2)
) {

    real l1 = sq_dist(p1,p0);
    real l2 = sq_dist(p2,p0);
    real l3 = sq_dist(p3,p0);

    real a10 = 2*dot_at(p1,q0,p0);
    real a11 = 2*dot_at(p1,q1,p0);
    real a12 = 2*dot_at(p1,q2,p0);
    real a20 = 2*dot_at(p2,q0,p0);
    real a21 = 2*dot_at(p2,q1,p0);
    real a22 = 2*dot_at(p2,q2,p0);
```

```
real a30 = 2*dot_at(p3,q0,p0);
real a31 = 2*dot_at(p3,q1,p0);
real a32 = 2*dot_at(p3,q2,p0);

/*
 * [ b00 b01 b02 ]           [  1   1   1  ]-1
 * [ b10 b11 b12 ] = Delta * [ a10 a11 a12 ]
 * [ b20 b21 b22 ]           [ a20 a21 a22 ]
 */

real b00 = a11*a22-a12*a21;
real b01 = a21-a22;
real b02 = a12-a11;
real b10 = a12*a20-a10*a22;
real b11 = a22-a20;
real b12 = a10-a12;
real b20 = a10*a21-a11*a20;
real b21 = a20-a21;
real b22 = a11-a10;


real Delta = b00+b10+b20;

/*
 *       [ Lambda0 ]   [ b01 b02 ]   [ l1 ]   [ b00 ]
 * Delta [ Lambda1 ] = [ b11 b12 ] * [    ] + [ b10 ]
 *       [ Lambda2 ]   [ b21 b22 ]   [ l2 ]   [ b20 ]
 */

real DeltaLambda0 = b01*l1+b02*l2+b00 ;
real DeltaLambda1 = b11*l1+b12*l2+b10 ;
real DeltaLambda2 = b21*l1+b22*l2+b20 ;

real r = Delta*l3 - (
       a30 * DeltaLambda0 +
       a31 * DeltaLambda1 +
       a32 * DeltaLambda2
) ;

Sign Delta_sign = sign(Delta) ;
Sign r_sign     = sign(r) ;

generic_predicate_result(Delta_sign*r_sign) ;
```

```
    begin_sos4(p0,p1,p2,p3)
       sos(p0, Sign(Delta_sign*sign(
                 Delta-((b01+b02)*a30+(b11+b12)*a31+(b21+b22)*a32)
       )))
       sos(p1, Sign(Delta_sign*sign((a30*b01)+(a31*b11)+(a32*b21))))
       sos(p2, Sign(Delta_sign*sign((a30*b02)+(a31*b12)+(a32*b22))))
       sos(p3, NEGATIVE)
    end_sos
}
```


[1] Aurenhammer, Hoffmann, Aronov, 1998. Minkowski-type thms. and least-squares clustering. Algorithmica 20 (1).

[2] Botella, A., Lévy, B., Caumon, G., "sep" 2013. Indirect hex-dominant mesh generation using a matching tetrahedra method. In: Proc. 33rd Gocad Meeting.

[3] Caumon, G., Laurent, G., Pellerin, J., Cherpeau, N., Lallier, F., Merland, R., Bonneau, F., 2013. Current bottlenecks in geomodeling workflows and ways forward. In: Garner, D., Thenin, D., Deutsch, C. V. (Eds.), Closing the Gap: Advances in Applied Geomodeling for Hydrocarbon Reservoirs. Vol. 20 of CSPG Memoir. Canadian Society of Petroleum Geologists, Ch. 4, pp. 43–52.

[4] CGAL, –. CGAL. Http://www.cgal.org.

[5] Edelsbrunner, H., Mücke, E. P., 1990. Simulation of simplicity. ACM TRANS. GRAPH 9 (1).

[6] GMP, –. GMP. Http://gmplib.org/.

[7] Lévy, B., 2014. `http://arxiv.org/abs/1409.1279`A num. algo. for L_2 semi-discrete optimal transport in 3d. ESAIM Math. Modeling and Analysis (accepted).

[8] Lévy, B., Liu, Y., 2010. Lp centroidal voronoi tesselation and its applications. ACM Transactions on Graphics (SIGGRAPH conference proceedings)Patent pending - FR 10/02920 (filed 07/09/10).

[9] Mérigot, Q., 2011. A multiscale approach to optimal transport. Comput. Graph. Forum 30 (5), 1583–1592.

[10] Merland, R., Caumon, G., Lévy, B., Collon-Drouaillet, P., 2014. Voronoi grids conforming to 3d structural features. Computational Geosciences, 11.

[11] Merland, R., Lévy, B., Caumon, G., September 2012. Voronoi grids conformal to 3d structural features. In: 13th European Conference on the Mathematics of Oil Recovery (ECMOR). Biarritz, France.

[12] Meyer, Pion, 2008. FPG: A code generator. In: Real Numbers and Computers. pp. 47–60.
URL http://hal.inria.fr/inria-00344297

[13] MPFR, –. MPFR. Http://www.mpfr.org.

[14] Nash, J. F., 1956. The imbedding problem for riemannian manifolds. Annals of Mathematics 63, 20–63.

[15] Pellerin, J., Lévy, B., Caumon, G., "oct" 2012. A voronoi-based hybrid meshing method. In: International Meshing Roundtable, Research Notes.

[16] Pellerin, J., Lévy, B., Caumon, G., Botella, A., "jan" 2014. Automatic surface remeshing of 3D structural models at specified resolution: a method based on voronoi diagrams. Computers & Geosciences 62, 103–116.

[17] Shewchuk, 1997. Adaptive precision floating-point arithmetic. Discrete & Computational Geometry 18 (3).

[18] Villani, C., 2009. Optimal transport : old and new. Grundlehren der math. Wissenschaften. Springer, Berlin.
URL http://opac.inria.fr/record=b1129524