Vorosweep: a fast generalized crystal growing Voronoi diagram generation algorithm.

T. Mouton[a,], E. Béchet[a]

[a] *Université de Liège, Aerospace and Mechanical Engineering Department, Chemin des Chevreuils, 1, 4000 Liège, Belgium*

1. Introduction

Voronoi Diagrams have a very wide range of applications in computer sciences: e.g. in motion planning, computer vision, mesh generation, as well as GIS, crystallography, chemistry, biology for fields outside of computer sciences. The Voronoi diagrams of sets of point sites in the euclidean 2D and 3D spaces is one of the most studied topic in computational geometry. Several algorithms are available for generating such diagrams and can handle huge datasets very efficiently. Nevertheless, and even in 2D, if we consider and other kinds of norms and generators like curves or areas, things have been less explored and in most of the cases, there are no fast and efficient algorithms available for their generation or if they exist, no implementation are known. We can even tell that no practically efficient algorithms are known for constructing a usable representation of generalized Voronoi diagrams, because of their intrinsic complexity. A solution is thus to approximate such diagrams and several attempts have been made in this direction. The Voronoi diagrams within a polygonal metric could be seen as good challengers but have been studied very little whereas their field of application could be large. Excepted for constant polygonal convex functions, including the L_∞-metric and the L_1-metric Voronoi diagrams, no algorithms can be found to generate such diagrams.

Many applications need generalized Voronoi diagrams and one of the most popular method used for generating this diagram is the raster-based methods which consists in computing the closest points for each pixel of a matrix of points. This obviously very simple implementation presents the drawback to be very time consuming and to have a very bad accuracy.

In this paper, we propose a new algorithm for generating quickly approximate generalized Voronoi diagrams of point sites associated to arbitrary convex distance metric. This algorithm produces connected cells by emulating the growth of crystals starting at the point sites, in order to reduce the complexity of the diagram. In the sequel, different principles adopted to decrease the

Email addresses: thibaud.mouton@gmail.com (T. Mouton), eric.bechet@ulg.ac.be (E. Béchet)

complexity of such diagrams will be described. Then the general algorithm and its implementation will be detailed. Finally, benchmarks will be given in order to demonstrate the efficiency of the algorithm as well as several examples that show its versatility.

2. 2D Voronoi diagrams

2.1. Definitions in the 2D euclidean metric

The Voronoi diagram for a set of points S in the 2D euclidean space E is one of the fundamental data structures of computational geometry and its properties have been studied extensively. We first give a few definitions. Let $S \in E$ a finite set of points:
To each point $p \in S$, the **Voronoi diagram** of S associates a region $VR(p)$ such that

$$VR(p) = \{x \in E \,|\, d(x, p) \leq d(x, q), \forall q \in S\} \tag{1}$$

It is a subdivision of the space where each point from S is associated with a region of the space closest to it. It is also called Dirichlet tessellation or Thiessen diagram.
A **site** is a defining object for a Voronoi diagram. Also called generator or source point.
A **Voronoi cell** is the set of points $x \in E$ closer to a single site or more generally to a set of sites. Voronoi region or face are equivalent names.
A **bisector** of 2 sites $p, q \in E$ is the separator of their Voronoi cell:

$$B(p, q) = \{x \in E \,|d(x, p) = d(x, q), \forall q \in S\} \tag{2}$$

A **Voronoi vertex** is the common points of at least 3 Voronoi cells and bounds 3 bisectors. Consequently, it is:

- equidistant to at least 3 sites of S
- closer to these sites than to any other site of S

A **geodesic** is a shortest path between 2 points p and q and is noted $SP(p, q)$. It has the property to be locally collinear to the gradient vector of the distance function from the considered point.

2.2. Generalizing the Voronoi diagrams

In the previous definition, Voronoi diagrams are defined by a single distance function d. This obviously restricts the variety of Voronoi diagrams that may be generated. As recalled by Emiris et al. [12], there are at least two ways to generalize the Voronoi diagrams. The first one consists in allowing non punctual sites, i.e. lines, circles, NURBS, polygons, etc under the Euclidean L_2-metric. The second one considers non constant distance functions, i.e. a domain on which a Riemannian metric field is defined. Because this generalization leads to very complicated diagrams, it can be relaxed by only considering the metric at the point sites, i.e. the function that defines the distance between one site and the other points of the space. It is the simplified definition followed by Labelle and Shewchuk [22]. In this case, a region $VR(p)$ is defined by:

$$VR(p) = \{x \in E \,|\, d_p(p,x) \leq d_q(q,x) \forall q \in S\} \tag{3}$$

Nevertheless, with such a definition, it is worth noting that in general $d_p(p,q) \neq d_q(q,p)$. This means that one site q will be seen by the site p at a different distance than site p from site q. Thus, the definition does not corresponds anymore to a metric per se, which must be symmetrical. The subdivisions into cells will however ensure that the global distance function remains symmetrical.

A third generalization can be envisaged which consists in constraining the diagram inside an arbitrary domain, i.e. by breaking the assumption that diagrams are generated on an infinite plane.

2.3. Classification of various Voronoi diagrams

We give here a classification of the most well known Voronoi diagrams resulting from different distance functions $d_q(\mathbf{q},\mathbf{x})$ with $\mathbf{q} = (q_x, q_y)$ and $\mathbf{x} = (x, y)$:

- VD of points under L_p-metric with p even: $d_q(\mathbf{q},\mathbf{x}) = (\mathbf{x}-\mathbf{q})^p = (x - q_x)^p + (y - q_y)^p$. It is also possible to combine it with an arbitrary rotation.
- Power (or Laguerre) diagrams of points with weight ω: $d_q(\mathbf{q},\mathbf{x}) = (\mathbf{x}-\mathbf{q})^2 - \omega_q^2$. This diagram is very similar in appearance to the euclidean Voronoi diagram except that the bisector can be move along the line $[\mathbf{qx}]$.
- Apollonius (or Additively weighted) diagrams of points with weight ω: $d_q(\mathbf{q},\mathbf{x}) = (\mathbf{x}-\mathbf{q}) - \omega_q$.
- Multiplicatively weighted diagrams of points with speed υ: $d_q(\mathbf{q},\mathbf{x}) = \upsilon_q * (\mathbf{x}-\mathbf{q})$.
- Möbius diagrams of points with speed υ and weight ω: $d_q(\mathbf{q},\mathbf{x}) = \upsilon_q * (\mathbf{x}-\mathbf{q})^2 - \omega_q^2$. They generalize power diagrams (when all υ_i are equals) and multiplicatively weighted diagrams (when all $\omega_i = 0$).
- Anisotropic diagrams of points with metric $\mathcal{M}$ (a symmetric positive definite matrix): $d_q(\mathbf{q},\mathbf{x}) = (\mathbf{x}-\mathbf{q})^t \mathcal{M}_q (\mathbf{x}-\mathbf{q})$ using the definition of Labelle and Shewchuk [22].

An alternative definition of the anisotropic diagrams of points as $VR(p) = \{x \in E \,|\, d_x(p,x) \leq d_x(q,x) \forall q \in S\}$ with distance $d_x(\mathbf{q},\mathbf{x}) = \sqrt{(\mathbf{x}-\mathbf{q})^t \mathcal{M}_x (\mathbf{x}-\mathbf{q})}$ is given in Du and Wang [10].

The list could be completed by any convex monotone function defined on the whole domain whose minimum is located at q. A huge inventory of distances already encountered are listed in Deza and Deza [9].

For non symmetric distance functions diagrams, we can cite the skew Voronoi diagrams for which the underlying geometry is not flat introduced by Aichholzer et al. [1].

Gorke [15] mentioned a combination of distances like the city Voronoi diagrams where the metric is induced by quickest paths according to the Manhattan metric and an accelerating transportation network consisting in non-intersecting axis-parallel line segments. Of course such a diagram could be extended to any kind of metric and using arbitrary transportation network.

2.4. Existing algorithms

Concerning algorithms, we have to distinguish between affine diagrams whose bisectors are planes and diagrams with curved bisectors. First ones include classical Voronoi diagrams and Laguerre diagrams and can be computed efficiently by the well known incremental method by Bowyer [6], the sweepline by Fortune [14] or the divide and conquer algorithm by Shamos and Hoey [25]. Other affine diagrams are special cases of Voronoi diagrams with an L_p-metric, when $p = 1$ or $p = \infty$. Their bisectors are in these case polylines constituted by at most 3 straight segments, making the computation rather simple and the previous algorithms still helpful with few modifications as mentioned by Shute et al. [27]. Then comes algorithms dealing with diagrams with algebraic bisectors (Möbius and anisotropic diagrams) and semi-algebraic bisectors (Apollonius diagrams). A good survey of these diagrams and proposed algorithms has been done by Boissonnat et al. [4]. Let us mention the work of Emiris and Karavelas [11], Boissonnat and Delage [3] and Karavelas and Yvinec [20] for the study and implementation of the Apollonius diagrams. A detailed implementation of the Möbius diagram is given by Delage [8].

Algorithms for anisotropic diagrams have been first proposed by Labelle and Shewchuk [22]. Their algorithm computes a subset of the lower envelope of the arrangement in E^3 of the paraboloids $z(\mathbf{x}) = (\mathbf{x} - \mathbf{q})^t \mathcal{M}_q (\mathbf{x} - \mathbf{q})$. By projecting the faces of the lower envelope down to E^2, the *minimization diagram* of the paraboloids is built, what they called an *anisotropic* Voronoi diagram. This approach has been then revisited by Boissonnat et al. [5] improving the computation complexity but no implementation is available to our knowledge.

For more complex diagrams like the ones mixing several types of distance, no exact and fast computational algorithm exists so far. Recently, Emiris et al. [12] proposed an algorithm for generating Voronoi diagrams of algebraic distance fields. They use a subdivision of a given domain into boxes. In each box, at most 3 distance fields are contributing to the Voronoi diagram and their method consists in filtering the appropriate fields. Their algorithm is able to handle polynomial or implicit distance fields.

Finally people in need of generalized Voronoi diagram frequently fall to discrete approaches like the raster-based method introduced by Hoff III et al. [17], where the diagram is computed using a polygonal mesh approximation of the distance functions and the Z-buffer of the GPU of the computer. In spite of being less accurate than vector-based method, it is very popular in many fields, considering the number of article citing this approach, which convinced us to create a non raster implementation of a generalized Voronoi diagram algorithm.

2.5. Constrained Voronoi diagrams

A further generalization of Voronoi diagrams appears if we do not consider an infinite planar domain but rather a finite and closed domain. Given a simple polygon in the plane and a set of k sites in its interior, a constrained Voronoi diagram of sites use the internal "geodesic" distance inside the polygon as the metric. This problem has been first settled by Aronov [2]. More generally, most of the previous generalization of Voronoi diagrams are only considering that distance is measured along geodesics that do not cross boundaries.

3. Designing a generalized Voronoi diagram algorithm

Most of the previous approaches used to solve this problem suffer from being exact methods that are not flexible. Many practical applications require to generate complex diagrams including several constraints and non algebraic distance functions. That is a good reason for designing a new algorithm able to generate an approximated Voronoi diagram using one or several kind of distance functions and able to generate at the same time a diagram whose cells are connected. The processing speed is also a mandatory requirement in order to be used. The goal of our study is to fill the gap between exact Voronoi diagrams computation and practical but less precise applications like the raster-based methods introduced by Hoff III et al. [17].

The new algorithm is based on the following principles:

- the crystal growth approach,
- the polygonal approximation of the wavefront,

and make use of the following tricks:

- add a dimension to the problem,
- build the so called "motorcycle graph" of rays emanating from the sites.

3.1. The crystal growth approach

In crystallography, the Voronoi diagram is used to model crystal growth when all the crystals start growing at the same time and have the same constant growth rate. If all the crystals have the same constant growth rate, but start growing at different times, the additively weighted Voronoi diagram models the growth. In the additively weighted Voronoi diagram, each site p has a weight $w(p)$. The separator of two sites in the additively weighted Voronoi diagram is a part of a hyperbola. For many physical systems, crystals having all the same constant growth rate may not be a valid assumption. If each crystal has a different growth rate, a multiplicatively weighted Voronoi diagram will model it accurately. Unfortunately, a strict multiplicatively weighted Voronoi diagram produces regions that are not connected, making its computation rather expensive. Moreover, this sort of diagram cannot model the physical settings, for which the domain are always connected.

In order to solve this problem, Schaudt and Drysdale [23] proposed to create the voronoi regions by imitating the physic of crystal and producing a diagram without nonconnected cells. The distance from a site to a point in its region is not anymore measured along a straight line but using the shortest path lying entirely within the region. This ensures to generate a diagram whose cells are connected and contain their generator site.

In order to precise this idea, we sum up the analytic study of Kobayashi and Sugihara [21] and we refer to Fig. 1. More precision can be found in Appendix A.

Let P1 and P2 be two crystal generators with weights (i.e., the growth speeds) v_1 and v_2, respectively. Let us assume that $v_1 < v_2$ and define $k = v_2/v_1$ as well as the coordinates of the generators be $P_1 = (0, 0)$ and $P_2 = (a, 0)$.

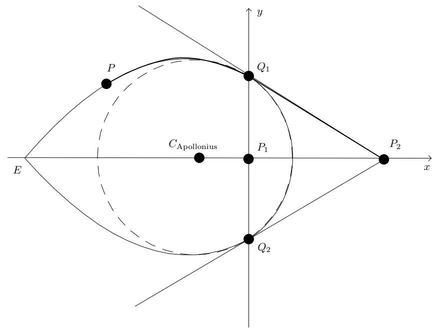


Figure 1: Multiplicatively weighted Voronoi cells.

First, we consider the portion of the boundary still visible from both of the crystals. Any point $P = (x, y)$ on this portion of the boundary satisfies:

$$\frac{\|P_1 - P\|}{v_1} = \frac{\|P_2 - P\|}{v_2}$$

This leads to

$$\left(x + \frac{a}{k^2 - 1}\right)^2 + y^2 = \frac{a^2k^2}{(k^2 - 1)^2}$$

which is the equation of the *Apollonius circle* whose center is $C_{Apollonius} = \left(-\frac{a}{k^2-1}, 0\right)$.

Nevertheless, for $t > t_2$ the distance from P_2 is measured in a straight line, crossing the cell of P_1. If we want to avoid this and consider a path only traveling through the P_2 cell, we have to cut the path into a straight part and a curved one, hidden from P_2. The part hidden from P_2 starts at a tangent point Q_1 and is under the line P_2Q_1, in the area $x < 0$. If now we focus on the boundary between the cell P_1 and P_2, i.e. their bisector, the equation of the boundary is not a circle anymore. This portion can be written as a polar equation $\mathbf{r}(\theta) = (r(\theta).\cos(\theta), r(\theta).\sin(\theta))$ with $r(\theta) = |\mathbf{r}(\theta)|$. $r(\theta)$ is the distance from the origin to the boundary point in the direction that forms an angle of θ, $\pi/2 < \theta < \pi$, with respect to the positive x-axis, so $r(\theta) = \|P_1 - P\|$.

It can be shown that the equation of the boundary for $x < 0$ is:

$$r(\theta) = \frac{a}{\sqrt{k^2-1}} e^{\frac{\theta - \pi/2}{\sqrt{k^2-1}}}, \quad \frac{\pi}{2} \leq \theta \leq \pi \tag{4}$$

This is the equation of a logarithmic spiral centered at P_1. This example is a good illustration of the crystal growth strategy. Nevertheless, this produces even for simple problems quite complicated bisector equations. The equation would be much more complex for anisotropic diagrams and in any case, it is practically impossible to compute every analytic solution of generalized bisectors, so that our goal is not to explicitly define bisectors but rather to approximate their geometry.

It should be noted that we are looking for the geometry of the bisector when one crystal totally encloses the second. Otherwise the equation is known and corresponds to a conic. The kind of bisector that is obtained is given in Fig. 2. It is also a piecewise spiral path. The method used to obtain such a result is given in Appendix B.

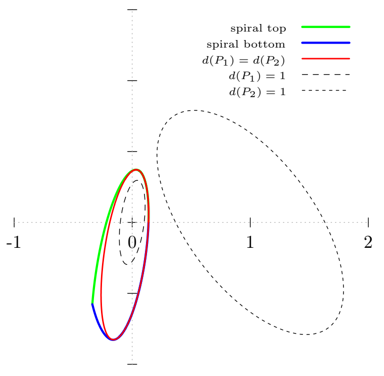


Figure 2: Anisotropic Voronoi cells.

3.2. *The wavefront expansion and the polygonal approximation*

Since our goal is to deal with arbitrary geodesics, it is obvious that the most natural and even only way to solve such a problem consists in using a wavefront approach, which is also known as the *continuous Dijkstra paradigm.*

Definition 1. *A wavefront starting at a point p_i is the set of points fulfilling the equation $d_i(p_i, q) = t$ at time $t \geq 0$ where $d_i(p_i, q)$ is the distance function associated to p_i. It is also commonly called a levelset.*

It worth noting that each point of the wavefront at time t is reached by the quickest possible path starting from P. Nevertheless, depending on the metric of the different sites and the configuration of the domain, wavefronts can be very complicated to handle. This complexity implies a very high computational cost that would make any exact method unusable.

In order to tackle this problem, the solution we are proposing is to use a polygonal approximation of the wavefronts. A wavefront at its early stage, i.e. before it encounters any other wavefront, is a closed curve. A polygonal approximation of a curve allows to drastically simplify any of the computations at the cost of losing accuracy. However, it can still be a good compromise between speed and precision if the discretization is made judiciously, by defining the size of the sides of the polygon accordingly to the local curvature of the wavefront. Furthermore, a given precision can be reached by considering a smaller discretization, but with an increased computational cost.

Definition 2. *A polygonal wavefront is a polygon whose shape is defined by the metric of the site it comes from. More precisely, the polygonal wavefront of a site p_i at time t is a polygon whose vertices q_j fulfill the equation $d_i(p_i, q_j) = t$.*

These polygonal wavefronts can be now seen as sets of vertices and edges expanding in the plane. The respective traces that they leave behind are by consequence lines and triangles. It is obvious that a vertex of a wavefront stops when it meets another vertex or edge. Knowing all the meeting points is basically what we want to compute and this problem looks very similar to the so called *motorcycle graphs*.

3.3. The motorcycle graph

The motorcycle graphs has been defined as follows by Eppstein and Erickson [13]: A motorcycle is a point moving with constant speed on a straight line. Let us consider n motorcycles $(m_1, ..., m_n)$, where each motorcycle m_i has its own constant velocity vector $v_i \in R^2$ and a start point $p_i \in R^2$, with $1 \leq i \leq n$. The trajectory $\{p_i + t \times v_i, t \geq 0\}$ is called the track of m_i. While a motorcycle moves it leaves a trace behind. When a motorcycle reaches the trace of another motorcycle then it stops driving – it crashes –, but its trace remains. It is also possible that motorcycles never crash. Following this terminology such motorcycles are said to have escaped. Fig. 3 illustrates a motorcycle graph with six motorcycles where only one manages to escape.

A naive way to generate such a diagram is to compute all intersection and then filter the ones that are not necessary. It is easy to see that there can be up to n^2 intersections among the motorcycle tracks. This algorithm has an obvious $O(n^2)$ complexity which is impractical for large datasets.
However, observing that no two motorcycle traces can be intersected by a third one in both interiors, we can deduce that this leads to at most n intersections among the traces ($O(n)$ theorical complexity). It is thus only necessary to

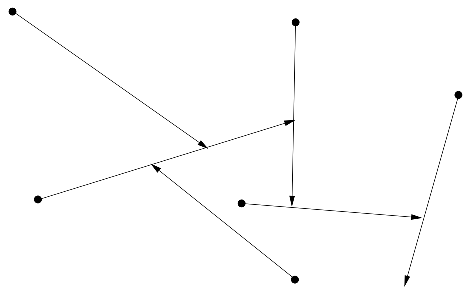


Figure 3: A motorcycle graph

compute these specific intersections. Several studies propose to decrease the complexity of the naive approach. The first of them (Eppstein and Erickson [13]) decreased the complexity to $O(n^{17/11+\epsilon})$. Then Cheng and Vigneron [7] proposed to generate a $1/\sqrt{n}$-cutting and exploiting the arrangements to bring a $O(n\sqrt{n}\log n)$ algorithm. Finally Huber and Held [18] gave a practical implementation of a $O(n\log n)$ in average algorithm. This last algorithm considers a geometrical hashing to achieve this average complexity assuming that the data are uniformly distributed. Recently Vigneron and Yan [29] proposed a new and faster algorithm for this problem but the implementation is far more tedious because of the use of a ray shooting datastructure as well as another one for halving ray shooting queries.

3.4. Adding a dimension

The different Voronoi diagrams of R^2 given in the list of section 2.3 have a nice geometrical interpretation. They can be seen as the projection on the R^2 plane of the lower envelope of geometrical entities of R^3. For diagrams whose distance is not squared, these tridimensional entities are cones (see Fig. 4). In the case of squared distance, they are paraboloids (e.g. Laguerre diagrams), or more generally elliptic paraboloids (e.g. Anisotropic diagrams) since paraboloids are special cases of a symmetric anisotropic metric. In the following, we assume that the metric is non squared.

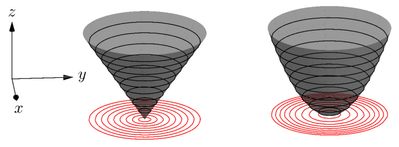


Figure 4: Lower envelopes of non squared and squared distances

This geometrical interpretation can be of great help for implementing the wavefront expansion by adding a third dimension to it. This 3-dimensional component represents the time that passed since the start of the wavefront in the xy-plane. Consequently wavefront edges are now tracing out planar polygons. Similarly, a vertex trace is the intersection line of the two adjacent planar polygons. The final Voronoi diagram is obtained by the projection of these polygons on the xy-plane, i.e. by removing the third dimension. Similarly, all the obstacles in the domain can be represented as vertically unbounded surfaces.

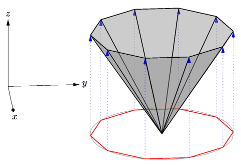


Figure 5: Adding dimension to the polygonal wavefront

3.5. *Put it all together*

What we need is clearly not a motorcycle graph but rather an algorithm able to catch intersections between advancing vertices and advancing lines. Advancing vertices are like motorcycles leaving a track behind them. Advancing lines can be seen as a straight rope that two motorcycles are handling between them, on which other motorcycles can crash as well. Nevertheless the analogy with motorcycles would stop there because complicated operations like splitting and merging will occurs at the same time. Based on these considerations, it is clear that we need a variant of the exact motorcycle graph.

To go further, let us recall that the geometrical interpretation of the Voronoi diagram of points whose distance is not squared is the lower envelope of cones. Since our wavefront is a polygonal approximation of an implicit curve, the cones will indeed be discrete cones, with pieces of planes on their sides. Starting from their apex, we want to compute the intersection of these cones. More precisely, we want to detect the intersection between planes and a lines that separate 2 planes.

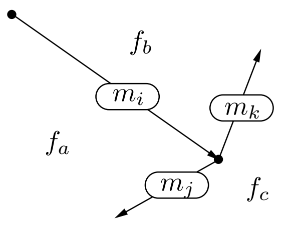


Figure 6: Motorcycle split.

This can be efficiently achieved by using the same kind of geometrical hashing used in Huber and Held [18]. The difference is that not only lines are stored in the hashing structure by also facets of the cones.

We now will diverge from the original motorcycle graphs because in our case, when a crash occurs, the motorcycle not only stops but 2 new motorcycles start from the crash point. Fig. 6 illustrates such a situation. Let a line m_i crashing into a facet f_c. m_i is separating facets f_a and f_b. When a crash occurs, m_i stops and create 2 new lines m_j and m_k which are now separating respectively facets f_a and f_c, and f_b and f_c. These new lines are then inserted into the hashing structure. The front of facet f_c is now split in two.

4. Implementation

The Vorosweep algorithm we are proposing and have implemented is driven by events and consists mainly into simple geometric objects expanding in the space and triggering the events. In the following, both topological data structure and events will be described.

4.1. Topological structure

The algorithm simulates the expansion of wavefronts in the plane. As we explained, these wavefronts can be very complex geometrical objects and a first step is to use a simplified description by discretizing them into piecewiese linear curves. Then by adding the time dimension, we got the cones stated in section 3.4, made of pieces of plane and lines separating the polygon.

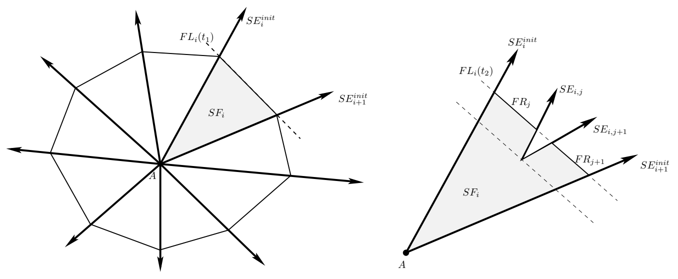
In the following, we will detail the most notable geometrical objects of the algorithm which are the **convex generators**, **sweepedges** and **sweepfacets**, **fronts** and **frontline**.

A **convex generators** *cg* is composed of the *apex A* of the future cone and a collection of planar **sweepfacets**. Intersections of *sweepfacets* generate the two initial linear **sweepedges** that bound each of the *sweepfacets*.

This two initial *sweepedges* SE_i^{init} and SE_{i+1}^{init} define the angular sector outside which the *sweepfacet* SF_i is not defined. A *sweepfacet* SF_i contains a **frontline** $FL_i(t)$ which represents the locus of points of SF_i at time t. A *frontline* contains itself one or several **fronts** FR_{ij} since the advancing *frontline* can be split into several parts (see Fig. 7(a) and Fig. 7(b)). A *front* has at all time a link to the two *sweepedges* that bound it and reciprocally a *sweepedge* always knows which *front* it belongs to. This is the topological structure of the algorithm allowing to generate the discrete Voronoi diagram.

Let us mention the **borders** objects that are used to circumscribe the domain on which the Voronoi diagram is defined. They are non expending vertical static planes bounded by vertical rays. Each border is adjacent to two other borders.

The whole data structure and the main relationships are detailed in the diagram of Fig. 8. It can be seen that sweepedges, bordersweepedges and sweepfacets are specialized sweepobjects types that derive from a main sweepobject class. Dotted arrows show the links between each entity. All these cross references ensure a quick and efficient traversal of the structure and a fast check of the topological consistency.



(a) Sweepobjects at t_1 (b) Sweepobjects at t_2

Figure 7: Sweepobjects before and after a split of the frontline

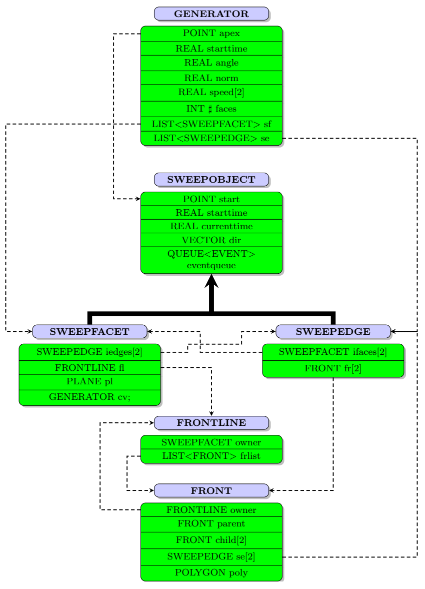


Figure 8: Sweeping datastructure

4.2. Underlying grid

In order to speedup the overall algorithm, a geometrical hashing structure, namely a bucket grid, is used. It is in fact a slightly modified version of datastructure used in the motorgraph algorithm described in Huber and Held [18]. The main difference consists in that in each bucket of the grid are not only registering the edges, but also the facets and the borders that are crossing the bucket. The main effect is that when entering in a bucket, the number of element to test for intersection is limited. By consequence, the total number of intersections to be computed for a sweepobject before it stops will be much lower than the whole set of entities.

The size of such a grid is very important because it forces the number of elements per bucket. This influences the number of elements to search among when looking for collisions. If the size is too small, most of the time is spent in switching from one bucket to another. This size is thus computed from the initial number n of sweeping objects. By following the similarity with the motorcycle graphs, we set the size of the grid as $\sqrt{n}$.

A similar but unidimensional grid structure is used for the borders, but registering only bordersweepedges. This allows to efficiently handle collisions between bordersweepedges. Its size is derived from the number of bucket crossed in the 2D grid.

A major drawback of such a method is that it requires the data to be well distributed over the domain so that each bucket will contain approximately the same number of elements. Nevertheless, we believe that this assumption can be acceptable most of the time.

4.3. Generator definition

A *generator* is defined by the planes described by a sweeping of the angle around its main axis. Each plane generated is the support of one *sweepfacet*. *Sweepedges* are then defined by intersecting each neighboring sweepfacet.

In order to be as general as possible, cones are based on *superellipses* also known a *Lamé curves* which are generalization of ellipses. A superellipse is a closed curve defined by the following implicit equation:

$$\left(\frac{x}{a}\right)^n + \left(\frac{y}{b}\right)^n = 1 \tag{5}$$

where a and b are the size (positive real number) of the major and minor axes and n is a rational number (see Jaklic et al. [19] for further precision).

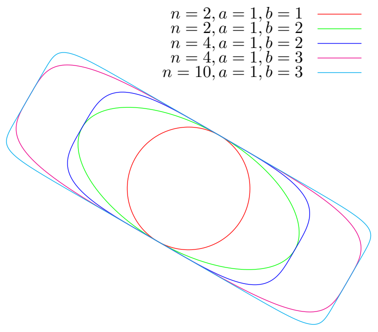


Figure 9: Superellipses with rotation = $\pi/3$

It comes that the parametric equation is:

$$f(t) = \begin{cases} x = a\cos^{2/n}(t) \\ y = b\sin^{2/n}(t) \end{cases}, \quad 0 \leq t \leq 2\pi \tag{6}$$

The related cone resulting from this general shape has the following equation:

$$\left(\frac{x}{a}\right)^n + \left(\frac{y}{b}\right)^n - z^n = 0 \tag{7}$$

leading to the general parameterization:

$$g(t,s) = \begin{cases} x = s\ a\cos^{2/n}(t) \\ y = s\ b\sin^{2/n}(t) \\ z = s \end{cases}, \quad \begin{matrix} 0 \leq t \leq 2\pi \\ 0 \leq s \leq \infty \end{matrix} \tag{8}$$

It should be noted that cos and sin exponentiations are a signed power function such that $\cos^n(t) = sign(cos(t))|cos(t)|^n$.

Tangential vectors are defined by

$$\mathbf{g}_t(t,s) = \frac{\partial g}{\partial t} = \begin{bmatrix} -\frac{a\ \cos^{\frac{2}{p}-1}(t)\ \sin(t)}{p} \\ \frac{b\ \cos(t)\ \sin^{\frac{2}{p}-1}(t)}{p} \\ 0 \end{bmatrix} \tag{9}$$

$$\mathbf{g}_s(t,s) = \frac{\partial g}{\partial s} = \begin{bmatrix} a\cos^{2/n}(t) \\ b\sin^{2/n}(t) \\ 1 \end{bmatrix} \tag{10}$$

It follows that the normal vector of such cones is defined by

$$\mathbf{n}(t,s) = \mathbf{g_t} \otimes \mathbf{g}_s \tag{11}$$

Rotation of angle θ is finally achieved by multiplying $\mathbf{n}(t,s)$ with $\mathbf{R}(\theta)$ so that

$$\mathbf{n}_\theta(t,s) = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \mathbf{n}(t,s) \tag{12}$$

Sweepfacets are finally generated from the set $\left\{\mathbf{n}_\theta(\frac{2k\pi}{n}, s), k = 0, 1, ..., n\right\}$ and the sweepedges result from the intersection of each of 2 successive sweepfacets.

4.4. Event structure

The algorithm consists in processing a suite of events. An event consists in a sweepobject at a given position with an occurring time. This occurring time is used to sort and trigger each event. These events fill an event queue sorted from the earliest to the oldest. The event queue is implemented as *binary search tree* which is the classical implementation of the *set* container of the standard C++ library. Each kind of event has a corresponding handler that contains appropriate methods to process it. It is important to describe each event to understand the propagation of the wavefronts:

1. A **NEWFACETSWITCH** event corresponds to the insertion of a newly created facet in the diagram. Insertion consists in finding the underlying bucket B_{ij} on which the apex A of the facet is located and registering the facet in this bucket. Then, next events are produced. The next events are generated by computing the crossings of the frontline with the corners of the bucket B_{ij} and then by checking the potential collisions with another entity registered into B_{ij}. All events are inserted into the local priority queue of the facet and at the end, the earliest event is inserted into the global priority queue.

2. A **NEWEDGESWITCH** event is the insertion of an edge into the event structure. It is worth noting that the z direction of the edge has be equal or greater than 0 otherwise it would mean that the edge is "coming back in the past". Like the *NEWFACETSWITCH*, the edge is registered into the bucket and next events are generated. The first next event is the location and the time at which the edge will leave the bucket into which it started. Other events are potential collisions with facets registered in the bucket and collisions with other edges that are attached to the 2 *fronts* of the edge. Like for facets, all events are inserted into the edge and only the earliest is inserted into the global priority queue.

3. A **NEWGENERATOR** event create a new *convex generator* at its apex position (x, y, t_0). If $t_0 > 0.0$, it first checks if no other face has already covered the position (x, y). If the position has been already covered, the generator is not created, otherwise it generates the n different *sweepfacets* and their respective *sweepedges*. Once faces and edges are created, the corresponding *NEWEDGESWITCH* and *NEWFACETSWITCH* events are emitted.

4. An **EDGEEDGECRASH** event is a collision between two *sweepedges*. In other words, it corresponds to a *front* closing and to a *Voronoi vertex* creation. This event can lead to several different situations:

   (a) a peak is created if no new edges are created (see Fig. 10(a)).

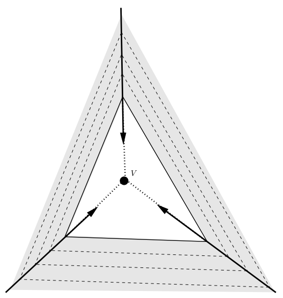
   (b) a new edge is created from the merge (see Fig. 10(b)).

   (c) several new edges and facets are created by "rotating" around the *Voronoi vertex* (see Fig. 10(c)).

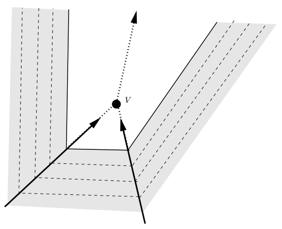
   (d) two fronts merge into an only one creating an "island" (see Fig. 10(d)).

   Fig. 10 illustrates these different situations.

5. An **EDGEFACETCRASH** event is a collision between a *sweepedge* and a *sweepfacet*. This event splits the front of the facet into two different fronts. Two new sweepedges are by consequence created and the corresponding *NEWEDGESWITCH* events are emitted (see Fig. 11).

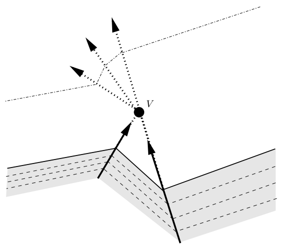


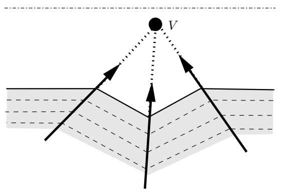
(a) Merge type 1.



(b) Merge type 2.

(c) Merge type 3.





(d) Merge type 4.

Figure 10: The different edge merging scenarios.

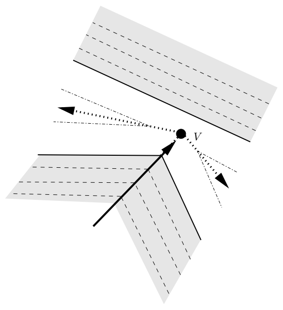


Figure 11: The edge to facet crash event.

6. An **EDGESWITCH** event occurs when an edge leaves a bucket and enters into another one. When entering into a new bucket, the sweepedge is registered into the bucket and collisions with facets and borders contained into the bucket are computed. If valid collisions are found, corresponding events are emitted and push into of the event queue of the sweepedge and the sweepfacet. Finally, the earliest event of the sweepedge is pushed into the general event queue. *FACETSWITCH* events for both facets attached to the sweepedge are generated.

7. A **FACETSWITCH** event occurs when a facet enters into a new bucket. As for an *EDGESWITCH* event, collisions with sweepedges registered into the bucket are computed and corresponding events are emitted.

   Fig. 12 illustrates 3 *SWITCH* events. Events 1 and 2 corresponds to *EDGESWITCH* events of edges SE_i and SE_{i+1} respectively. They will insert edges SE_i and SE_{i+1} into buckets $B_{j,k+1}$ and $B_{j+1,k}$ respectively. They will emit a *FACETSWITCH* event when being processed so that they are called indirect events. Event 3 is only a *FACETSWITCH* event that will insert SF_i into bucket $B_{j+1,k+1}$, it is a direct event because it has been emitted when SF_i from vertex A has been inserted.

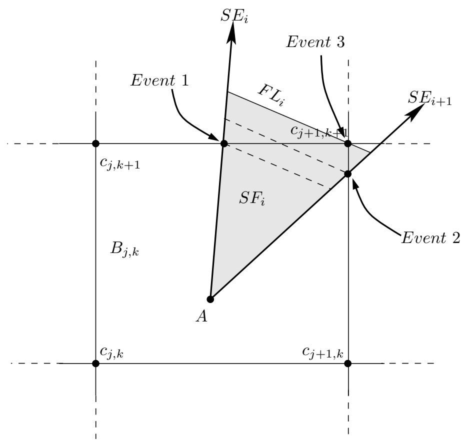


Figure 12: The facetswitch event.

8. A **NEWBORDEREDGESWITCH** event is similar to the *NEWEDGESWITCH* except that the concerned sweepedge is specialized into following a border.

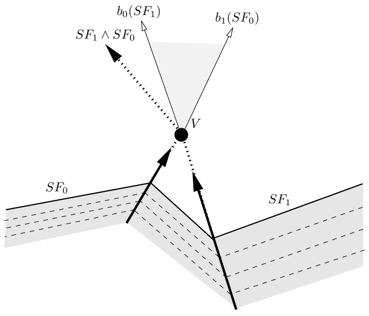
9. A **BORDEREDGESWITCH** event is the counterpart of the *EDGESWITCH* event for borderbuckets, since borders contain a 1D grid.

10. An **EDGEBORDERCRASH** event corresponds to the collision of a sweepedge and a border. In this case, the sweepedge is stopped and two sweepborderedges are created.

11. A **FACETBORDERCRASH** event occurs when a facet hits the vertex of a border. In this case, the front on which the bounding edge crashes is split into 2 other fronts and 2 sweepborderedges are created following the 2 edges of the border adjacent to the hit vertex.

4.5. Crystal growing strategy

It is important to understand that we did not mention the crystal growth behavior because it is actually naturally handled by our algorithm. We recall that the crystal growth comes out by walking around obstacles encountered. In the algorithm, this behavior is initiated by an *EDGEEDGECRASH* event. Such an event merges 2 sweepedges into an other one if needed. This new edge is computed by the cross product of normals of both external faces. Nevertheless, the newly created edge has to be contained into the angular wedge $b_0\widehat{(SF_1)b_1(SF_0)}$. In Fig. 13(a), we can see that the new edge result of $SF_1 \wedge SF_0$ is outside this angular wedge. In this case, as many as necessary new facets and new edges are created in order to fulfill the angular requirement. The result is illustrated in Fig. 10(c). The induced rotation can be seen on a real example in Fig. 13(b) making the spiral path whose analytic solution has been given in section 3.1 to appear clearly.



(a) New edge outside angular wedge.

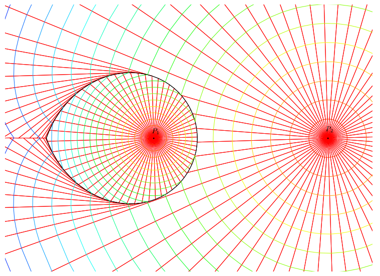

(b) Rotations occurring along the spiral path.

Figure 13: Crystal growth strategy results.

4.6. Main routine

For each of these events corresponds one handling function. It comes that the algorithm can be summarized by Algorithm 1. Init stage consists only in populating the main event queue with *NEWGENERATOR* events and then the *Run* routine is called. New events will be automatically generated by handlers, populating the event queue. The main routine ends when the main event queue is empty.

4.7. Output

The *Vorosweep* outputs a topological datastructure composed of the adjacency between cells, polylines representing the bisectors as well as a triangulation of the diagram making the diagram generated easy to use for any path planning, closest neighbor or any other application.

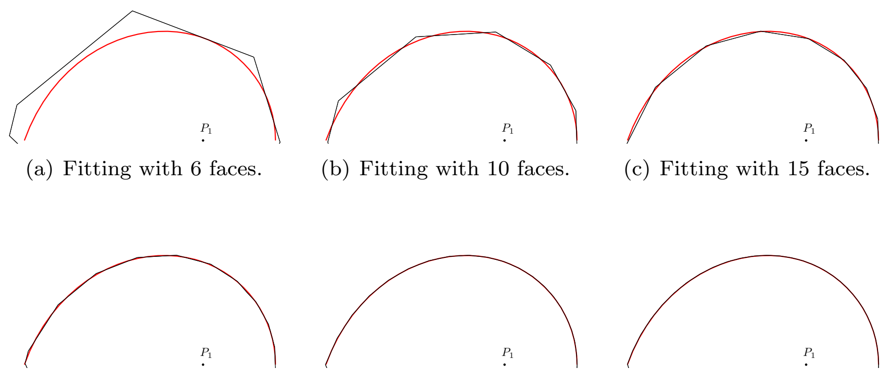
Algorithm 1 Main routine

```
procedure Run(QUEUE<Event> Ev)
    while Ev $\neq$ empty do
        e $\leftarrow$ earliest(Ev)
        so $\leftarrow$ get_sweepobject(e)
        remove_earliest_event(so)
        if is_active(so) then
            if type(e) == NEWGENERATOR then
                handle_newgenerator(e)
            else if type(e) == NEWEDGESWITCH then
                handle_newegdgeswitch(e)
                ...
            end if
        end if
    end while
end procedure
```

5. Results

5.1. Accuracy

Since we do not have the possibility to compute the analytic solution of most of the bisectors, we compared the results with the only analytic solution we have, i.e. the crystal growing multiplicatively weighted Voronoi diagram. Example presented in Fig. 14 is the solution we obtained with our algorithm for generators with different number of faces. The corresponding layout is the one of Fig. 1 with parameters $a = 0.5$, $k = 3$.



(a) Fitting with 6 faces. (b) Fitting with 10 faces. (c) Fitting with 15 faces.

(d) Fitting with 20 faces. (e) Fitting with 40 faces. (f) Fitting with 60 faces.

Figure 14: Error between analytic solution (in red) and our discrete solution (in black).

As we can see, precision is already quite good with 20 faces and it quickly converges to the analytic solution so that no differences can be seen with more than 40 facets per generator.

5.2. *Statistics on academic test cases*

Our code is a C++ code making highly use of standard library and its datastrutures. It has been developed and tested on x86_64 Linux operating system and compiled using gcc compiler version 4.6 using standard optimization 02 flags. The performance benchmarks have been obtained on an Intel Xeon X5690 CPU running at 3.47GHz using 24GB of RAM. While this CPU is a multicore one, the *Vorosweep* algorithm does not take advantage of this.

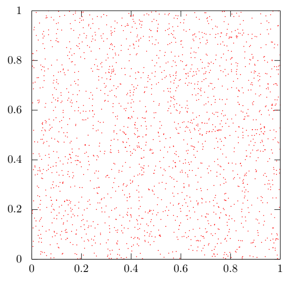


Figure 15: Random dataset.

Datasets have been automatically generated with n generators by randomly choosing the starting points uniformly in the unit square $[0,1]\times[0,1]$ (see Fig. 15). All other parameters are also randomly generated, including the direction angle of the axis of the ellipses in $[0,\pi/2]$, as well as their lengths. Fig. 17 gives running times of datasets for generators with 40 faces and 60 faces. As expected, using 60 faces takes roughly 50 % more time than with 40 faces. It is interesting to note the linear dependency of the running time with the number of generators, exhibiting the $n \log n$ average behavior of the algorithm. We know that a very bad distribution of the points over the domain would lead to a very bad worst case complexity. But is this likely to happen in real life ?

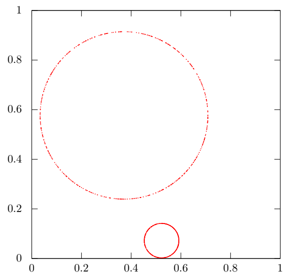


Figure 16: Circle dataset.

In order to give an idea of the running time for badly distributed generators, we ran the algorithm on non uniform datasets. The datasets are generated from two circles with center and radius randomly chosen in $[0,1]\times[0,1]$ and $[0,1]$ (see Fig. 16). Then all points are generated on this circles in order to create very badly distributed points. Results are given in Fig. 18. What can be seen is that in some rare situation, the runtime can reach $4\times$ the average runtime for the same number of generators. To precise the overhead cost, histograms of 4 intervals have been produced, showing the runtime divided by $n \log n$. It looks obvious that the majority of the time, the runtime will be less than $1.2\times10^{-3}\ n\log n$ seconds. Given that the average uniform time is about $0.9\times10^{-3}\ n\log n$ seconds, around 10% of the cases will take more than 2 times the average runtime.

Let us mention that an example with 10000 generators made of 60 facets consists in $10000 \times (\text{ 60 facets} + \text{60 edges }) = 1.2\times10^{6}$ objects and is expected

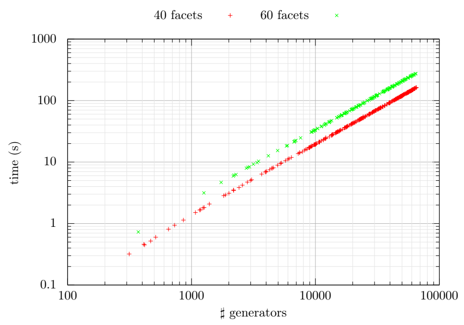


Figure 17: Runtime (y-axis) for n generators (x-axis). Datasets are randomly generated on a unit square with generators made of 40 and 60 facets.

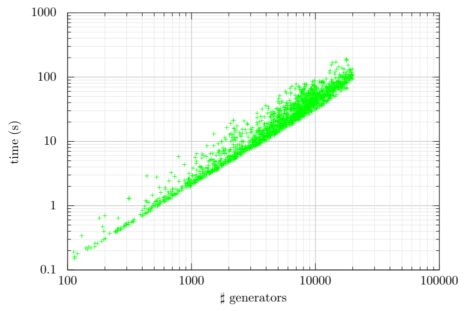
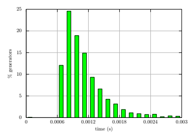
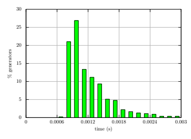


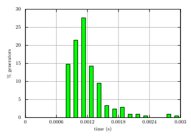
Figure 18: Runtime for datasets which are randomly generated on 2 circles with generators made of 60 facets.



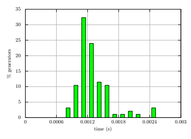
(a) $0 < \sharp \leq 5000$



(b) $5000 < \sharp \leq 10000$



(c) $10000 < \sharp \leq 15000$



(d) $15000 < \sharp \leq 20000$

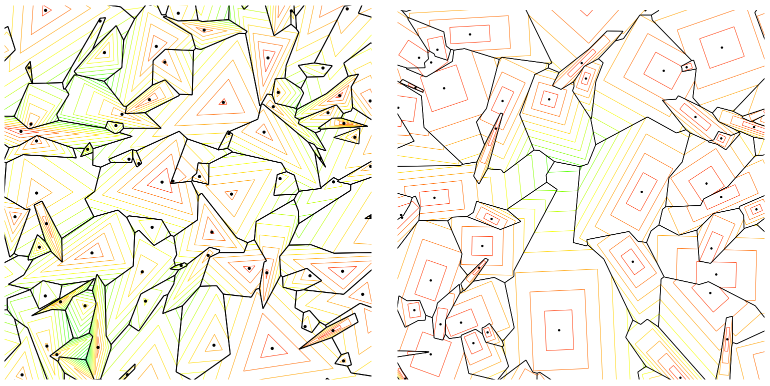
Figure 19: Running time distribution for the non uniform datasets with generators made of 60 facets.

to run within 35 seconds if data are randomly enough distributed, processing around 15×10^6 events.

The only algorithm to be known being comparable of ours is the one of Emiris et al. [12] which computes an exact solution. It gives 19.4 seconds for 800 sites in the L_8 metric. Nevertheless, it is more limited in the range of possibility it offers. In comparison, our algorithm takes around 3 seconds for generating the diagram of 1000 sites with a reasonably good precision.

5.3. Graphical examples

In this section we give several examples of diagrams of various settings that can be obtained with *Vorosweep*. All examples are generated from random sets of parameters. Since they contain less than 500 generators, they are processed under one second.
We start by showing examples of polygonal metric in Fig. 20. These are generators with only 3 and 4 faces. The last example can be seen as a diagram in the oriented L_∞ metric and an additional anisotropy factor.



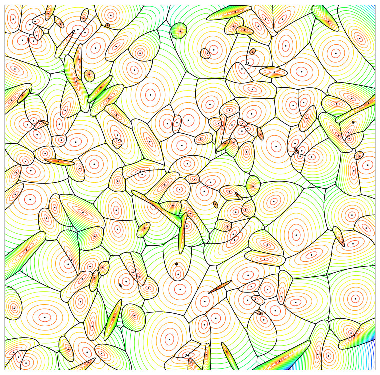
(a) With generators made of 3 faces. (b) With generators made of 4 faces generators. Also known as L_∞ metric with an additional orientation angle.

Figure 20: Polygonal metric examples.

The following examples show approximate anisotropic diagrams. Fig. 21(a) shows a simple example of what produces the algorithm. Fig. 21(b) is generated by setting the speed of one of the generators 20× greater than the second fastest generator. It is interesting to see how the isovalues are increasing over the domain, enclosing all other cells. All the generators are made of 60 facets in order to ensure a good precision.

Fig. 22(a) is used to demonstrate the versatility of our approach. In this example, the generators are associated to an anisotropic metric under an arbitrary L_p norm, as explained in section 4.3. They also feature a randomly chosen starting time which affects the shape of the cells.

Finally, Fig. 22(b) shows a diagram built inside an arbitrary non convex polygonal domain. Isolines demonstrate that the diagram follows the underlying metric induced by the boundaries of the domain.


(a) Anisotropic 60 facets simple example.

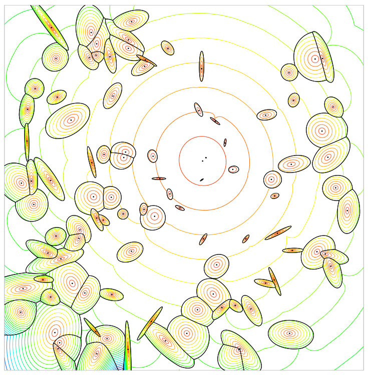
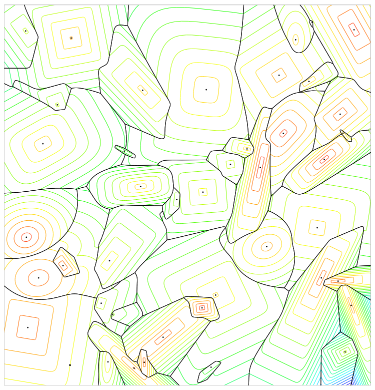

(b) Anisotropic 60 facets example with high velocity difference.

Figure 21: Anisotropic examples.


(a) L_p with $2 \leq p \leq 10$ oriented metric 60 facets example with random starting time.

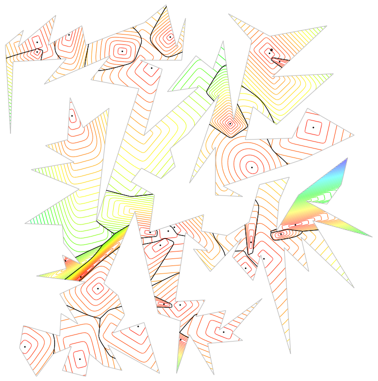

(b) L_p with $2 \leq p \leq 10$ oriented metric 60 facets example constrained in a domain.

Figure 22: Advanced examples of diagrams using L_p metric.

5.4. Robustness issues

It is obvious that like for any computational geometry algorithm, robustness is one of the hardest problem to address. Vorosweep does not escape to the statement and it is still an issue to make the implementation a robust one. If general position inputs can be efficiently handled, degenerated ones are quite complicated to handle. Fig. 23 shows a completely degenerated input which is challenging to handle for the algorithm. The reason can be explained by the fact that all events occur exactly at the same time and it is still an open problem for us to produce a scheduling algorithm efficient enough not to raise the processing time to an impractical level.

The solution we adopted temporarly is to add a small perturbation to the angle of the generator, so that all events are not scheduled at the same time, allowing the algorithm to run in a coherent way. This makes the algorithm not deterministic but few perturbations do not lead to dramatic changes in the Voronoi diagram structure anyway.

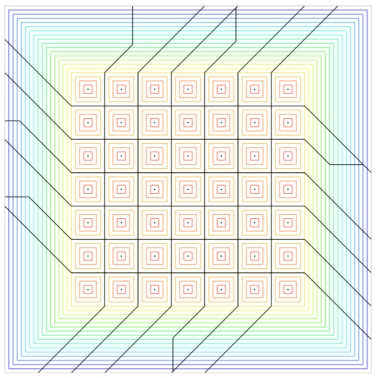


Figure 23: Degenerated input : all points are positioned on a grid with the same angle.

6. Conclusion and future work

6.1. Robustness

A future extension of this work would consist in an improvement of the robustness of our proof of concept *Vorosweep*. Indeed, robustness has not been our first priority. It is clear that a good implementation should follow the so-called topology oriented implementation proposed in Sugihara et al. [28] and, to some extent, the current implementation already take this approach into account. But an important work remains to be done in order to analyze weaknesses of our algorithm and fix them. Our code makes also heavy use of orientation tests and since few of them are crucial in the run of the algorithm, these predicates should be replaced by more robust versions like for instance the ones from Shewchuk [26].

6.2. Performance improvements

In order to make the runtime performance of our algorithm less dependent to irregularly distributed generators, we have planned to replace the rectangular regular grid by quad trees, allowing a geometric hashing of the plane following clusters of points.

Since the algorithm is processing on a grid, a multithreaded implementation would also be possible. This assume that an independent processing can be done on each bucket of the grid. Since this condition is only fulfilled at the beginning of the processing, such an implementation would potentially only speedup the beginning of the algorithm and thus may not be as interesting as expected.

6.3. Higher order metric

Our framework only handle linear metric, i.e. not squared distances, like power diagrams or even logarithmic metric and so on. Implementing them would raise the complexity of the implementation. Indeed, it would involve parabolic or logarithmic cylinders and their intersections instead of planes and straight lines.

6.4. Generator types

It is obvious that this approach is not restricted to point sites, and more complex sites are can be considered, like segments (See Fig. 24), arbitrary curves or even free forms, by still using our discretization method. All of these kind of diagrams can be easily implemented thanks to the generality of our approach. It is also easy to include features like "*wind field*" by modifying the angle of the axis of the generators between the *xy* plane.

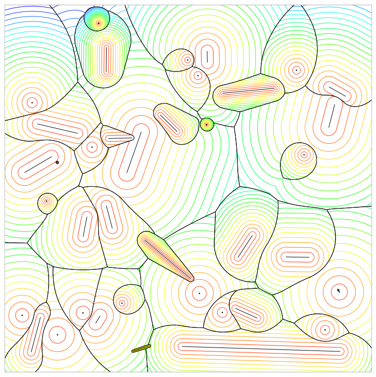
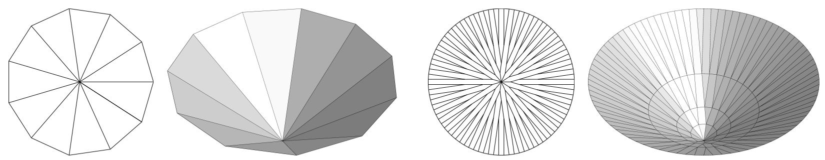


Figure 24: Point and segment sites with random speeds.

6.5. Accuracy improvements

Accuracy improvements should also be done, especially for points which are far from their generator. If the relative error remains the same, the absolute error increases. In Fig. 25 we show the kind of pattern that we are planning to bring into our framework. Given that faces are not intended to go for long distances, this increase in the number of sweepobjects should be limited in average. Experiments are still to be done in this direction.



(a) Actual lack of accuracy. (b) Refinement strategy that should be adopted.

Figure 25: Refinement strategy keeping the approximation of the wavefront.

6.6. Extension to 3D

It looks like the extension to 3D is possible, and a step in this direction has already been made by Held [16]. A significant work is nevertheless to be done in order to obtain results in 3D, since it raises the complexity of the whole algorithm to a much higher level.

To conclude, we want to ensure that the reader understands that this method is very different from others that could look very close like the one by Hoff III et al. [17] and by Emiris et al. [12]. If they are the only ones to give a practical framework able to generate generalized Voronoi diagrams, they do not propose a method that produces orphan free diagrams, which is the major strength of ours.

We believe that this implementation is very promising since it is closely related to different problems, from the shortest path in a domain with polygonal obstacles to the fast marching method of Sethian [24]. The *Vorosweep* package can be found at `http://www.cadxfem.org/vorosweep` and is distributed under the GPL license for non commercial use.

Acknowledgements

This research project was funded in part by the Walloon Region under WIST 3 grant 1017074 DOMHEX (Dominant Hexahedral Mesh Generation).

References

[1] Oswin Aichholzer, Franz Aurenhammer, Danny Z. Chen, D. T. Lee, and Evanthia Papadopoulou. Skew voronoi diagrams. *International Journal of Computational Geometry & Applications*, 9(03):235247, 1999. URL `http://www.worldscientific.com/doi/abs/10.1142/S0218195999000169`.

[2] Boris Aronov. On the geodesic voronoi diagram of point sites in a simple polygon. *Algorithmica*, 4(1-4):109–140, June 1989. ISSN 0178-4617, 1432-0541. doi: 10.1007/BF01553882. URL `http://link.springer.com/article/10.1007/BF01553882`.

[3] Jean-Daniel Boissonnat and Christophe Delage. Convex hull and voronoi diagram of additively weighted points. In Gerth Stlting Brodal and Stefano Leonardi, editors, *Algorithms ESA 2005*, number 3669 in Lecture Notes in Computer Science, pages 367–378. Springer Berlin Heidelberg, January 2005. ISBN 978-3-540-29118-3, 978-3-540-31951-1. URL `http://link.springer.com/chapter/10.1007/11561071_34`.

[4] Jean-Daniel Boissonnat, Camille Wormser, and Mariette Yvinec. Curved voronoi diagrams. In Jean-Daniel Boissonnat and Monique Teillaud, editors, *Effective Computational Geometry for Curves and Surfaces*, pages 67–116. Springer Berlin Heidelberg, January 2006. ISBN 978-3-540-33258-9, 978-3-540-33259-6. URL `http://link.springer.com/chapter/10.1007/978-3-540-33259-6_2`.

[5] Jean-Daniel Boissonnat, Camille Wormser, and Mariette Yvinec. Anisotropic diagrams: Labelle shewchuk approach revisited. *Theoretical Computer Science*, 408(23):163–173, November 2008. ISSN 0304-3975. doi: 10.1016/j.tcs.2008.08.006. URL `http://www.sciencedirect.com/science/article/pii/S0304397508005598`.

[6] A. Bowyer. Computing dirichlet tessellations. *The Computer Journal*, 24 (2):162–166, January 1981. ISSN 0010-4620, 1460-2067. doi: 10.1093/comjnl/24.2.162. URL `http://comjnl.oxfordjournals.org/content/24/2/162`.

[7] Siu-Wing Cheng and Antoine Vigneron. Motorcycle graphs and straight skeletons. *Algorithmica*, 47(2):159–182, February 2007. ISSN 0178-4617, 1432-0541. doi: 10.1007/s00453-006-1229-7. URL `http://link.springer.com/article/10.1007/s00453-006-1229-7`.

[8] Chistophe Delage. *CGAL-based First Prototype Implementation of Moebius Diagram in 2D.* 2003.

[9] Elena Deza and Michel-Marie Deza. Chapter 20 - voronoi diagram distances. In Elena Deza and Michel-Marie Deza, editors, *Dictionary of Distances*, pages 253–261. Elsevier, Amsterdam, 2006. ISBN 978-0-444-52087-6. URL `http://www.sciencedirect.com/science/article/pii/B9780444520876500202`.

[10] Qiang Du and Desheng Wang. Anisotropic centroidal voronoi tessellations and their applications. *SIAM Journal on Scientific Computing*, 26(3):737761, 2005. URL `http://epubs.siam.org/doi/abs/10.1137/S1064827503428527`.

[11] Ioannis Z. Emiris and Menelaos I. Karavelas. The predicates of the apollonius diagram: Algorithmic analysis and implementation. *Computational Geometry*, 33(1-2):18–57, January 2006. ISSN 09257721. doi: 10.1016/j.comgeo.2004.02.006. URL `http://linkinghub.elsevier.com/retrieve/pii/S0925772105000623`.

[12] Ioannis Z. Emiris, Angelos Mantzaflaris, and Bernard Mourrain. Voronoi diagrams of algebraic distance fields. *Computer-Aided Design*, 45(2):511–516, February 2013. ISSN 00104485. doi: 10.1016/j.cad.2012.10.043. URL `http://linkinghub.elsevier.com/retrieve/pii/S0010448512002485`.

[13] David Eppstein and Jeff Erickson. Raising roofs, crashing cycles, and playing pool: Applications of a data structure for finding pairwise interactions. *Discrete & Computational Geometry*, 22(4):569592, 1999. URL `http://link.springer.com/article/10.1007/PL00009479`.

[14] Steven Fortune. A sweepline algorithm for voronoi diagrams. *Algorithmica*, 2(1-4):153174, 1987. URL `http://link.springer.com/article/10.1007/BF01840357`.

[15] Robert Gorke. Constructing the city voronoi diagram faster. In *in Proc. 2nd Int. Symp. on Voronoi Diagrams in Science and Engineering (VD05*, page 162172, 2005.

[16] Martin Held. On computing voronoi diagrams of convex polyhedra by means of wavefront propagation. In *CCCG*, page 128133, 1994. URL `ftp://129.49.108.37/geometry/cccg94.ps.gz`.

[17] Kenneth E. Hoff III, John Keyser, Ming Lin, Dinesh Manocha, and Tim Culver. Fast computation of generalized voronoi diagrams using graphics hardware. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, page 277286. ACM Press/Addison-Wesley Publishing Co., 1999. URL `http://dl.acm.org/citation.cfm?id=311567`.

[18] Stefan Huber and Martin Held. Motorcycle graphs: Stochastic properties motivate an efficient yet simple implementation. *Journal of Experimental Algorithmics*, 16:1.1, May 2011. ISSN 10846654. doi: 10.1145/1963190.2019578. URL `http://dl.acm.org/citation.cfm?doid=1963190.2019578`.

[19] Ales Jaklic, Ales Leonardis, and Franc Solina. *Segmentation and Recovery of Superquadrics*. Springer, September 2000. ISBN 9780792366010.

[20] Menelaos Karavelas and Mariette Yvinec. 2d apollonius graphs (delaunay graphs of disks). In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.3 edition, 2013. URL `http://doc.cgal.org/4.3/Manual/packages.html#PkgApolloniusGraph2Summary`.

[21] Kei Kobayashi and Kokichi Sugihara. Crystal voronoi diagram and its applications. *Future Generation Computer Systems*, 18(5):681–692, April 2002. ISSN 0167-739X. doi: 10.1016/S0167-739X(02)00033-X. URL `http://www.sciencedirect.com/science/article/pii/S0167739X0200033X`.

[22] Francois Labelle and Jonathan Richard Shewchuk. Anisotropic voronoi diagrams and guaranteed-quality anisotropic mesh generation. In *Proceedings of the nineteenth annual symposium on Computational geometry*, page 191200, 2003. URL `http://dl.acm.org/citation.cfm?id=777822`.

[23] Barry F. Schaudt and R. L. Drysdale. Multiplicatively weighted crystal growth voronoi diagrams. In *Proceedings of the seventh annual symposium on Computational geometry*, page 214223, 1991. URL `http://dl.acm.org/citation.cfm?id=109672`.

[24] James A. Sethian. Fast marching methods. *SIAM review*, 41 (2):199235, 1999. URL `http://epubs.siam.org/doi/abs/10.1137/S0036144598347059`.

[25] Michael Ian Shamos and Dan Hoey. Closest-point problems. In *Foundations of Computer Science, 1975., 16th Annual Symposium on*, page 151162. IEEE, 1975. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4567872`.

[26] Jonathan Richard Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, 18(3):305363, 1997. URL `http://link.springer.com/article/10.1007/PL00009321`.

[27] Gary M. Shute, Linda L. Deneen, and Clark D. Thomborson. AnO (n logn) plane-sweep algorithm forL 1 andL delaunay triangulations. *Algorithmica*, 6 (1-6):207221, 1991. URL `http://link.springer.com/article/10.1007/BF01759042`.

[28] K. Sugihara, M. Iri, H. Inagaki, and T. Imai. Topology-oriented implementationan approach to robust geometric algorithms. *Algorithmica*, 27(1): 5–20, May 2000. ISSN 0178-4617, 1432-0541. doi: 10.1007/s004530010002. URL `http://link.springer.com/article/10.1007/s004530010002`.

[29] Antoine Vigneron and Lie Yan. A faster algorithm for computing motorcycle graphs. In *Proceedings of the 29th annual symposium on Symposuim on computational geometry*, page 1726, 2013. URL `http://dl.acm.org/citation.cfm?id=2462396`.

Appendix A. Multiplicatively weighted Voronoi diagrams.

The Apollonius circle equation is obtained by the identity:

$$\frac{\|P_1 - P\|}{v_1} = \frac{\|P_2 - P\|}{v_2}$$

This can be written as

$$\frac{x^2 + y^2}{v_1^2} = \frac{(x-a)^2 + y^2}{v_2^2}$$

$$k^2(x^2+y^2) = (x-a)^2 + y^2$$

$$x^2 + y^2 + \frac{2ax}{k^2-1} = \frac{a^2}{k^2-1}$$

$$x^2 + \frac{2ax}{k^2-1} + \frac{a^2}{(k^2-1)^2} + y^2 = \frac{a^2}{k^2-1} + \frac{a^2}{(k^2-1)^2}$$

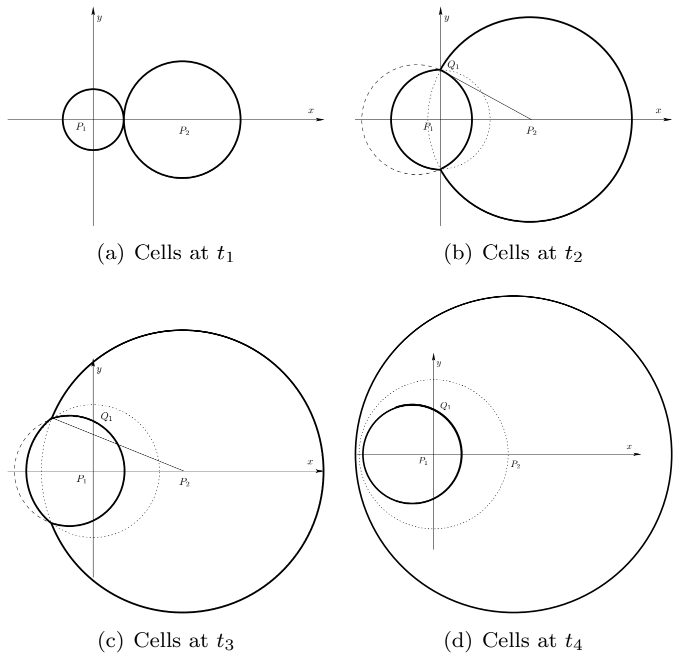
$$\left(x + \frac{a}{k^2-1}\right)^2 + y^2 = \frac{a^2k^2}{(k^2-1)^2}$$

This is the equation of a circle $(x-x_0)^2 + (y-y_0)^2 = R^2$ with

$$\begin{cases} x_0 = -\dfrac{a}{k^2-1} \\ y_0 = 0 \\ R = \dfrac{ak}{(k^2-1)} \end{cases} \tag{A.1}$$

also called the *Apollonius circle*.

Fig. A.26 shows the different step of the growth.



(a) Cells at t_1 (b) Cells at t_2

(c) Cells at t_3 (d) Cells at t_4

Figure A.26: Steps of multiplicatively weighted Voronoi cells growth. In this case, the distance is measured along a straight line thus creating the Apollonius circle.

We will now detail the computation of the curved portion of the boundary. At point Q_1, $x = 0$, so we can write:

$$\frac{\|P_1 - P\|}{v_1} = \frac{\|P_2 - P\|}{v_2}$$
$$y^2 = \frac{a^2k^2}{(k^2-1)^2} - \frac{a^2}{(k^2-1)^2} = \frac{a^2}{k^2-1}$$
$$y = \frac{a}{\sqrt{k^2-1}}$$

By consequence, $\|P_1 - P\| = \frac{a}{\sqrt{k^2-1}}$ at point Q_1, and $\|P_2 - P\| = k\|P_1 - P\|=\frac{ak}{\sqrt{k^2-1}}$

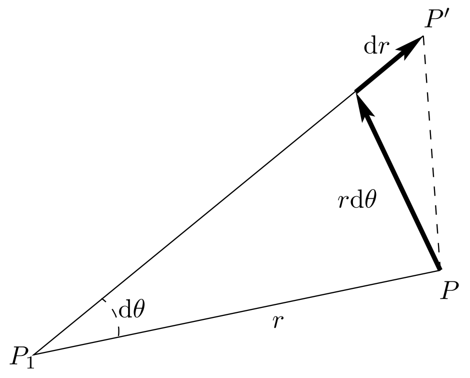


Figure A.27: Evolution of a point at boundary at an infinitesimal level.

Let have a look at Fig. A.27 which represents the polar frame of a point at the boundary. By carefully noting that $\mathrm{d}r$ and $[PP']$ are related to the speed of P1 and P2, we can write that $\mathrm{d}r = v_1\delta t$ and $[PP'] = v_2\delta t$ for a small increment δt. Since $r\mathrm{d}\theta^2 + \mathrm{d}r^2 = [PP']^2$:

$$\begin{aligned} r\mathrm{d}\theta &= \sqrt{[PP']^2 - \mathrm{d}r^2} \\ &= \sqrt{(v_2\delta t)^2 - (v_1\delta t)^2} \\ &= v_1\delta t\sqrt{k^2-1} \end{aligned} \tag{A.2}$$

We can finally write that

$$\frac{\mathrm{d}r}{r\mathrm{d}\theta} = \frac{1}{\sqrt{k^2-1}} = C^{te}$$

This is a constant coefficient first order equation of the kind

$$\frac{\mathrm{d}r}{\mathrm{d}\theta} + Br = 0$$

whose non trivial analytic solution is $r(\theta) = Ce^{-B\theta}$.
Since the curved section starts at Q_1, $\theta = \pi/2$, and from the *Apollonius circle* equation we got $r(\theta) = \frac{a}{\sqrt{k^2-1}}$, the final equation of the boundary for $x < 0$ is:

$$r(\theta) = \frac{a}{\sqrt{k^2-1}} e^{\frac{\theta-\pi/2}{\sqrt{k^2-1}}}, \quad \frac{\pi}{2} \leq \theta \leq \pi \tag{A.3}$$

After point Q_1, $\|P_2 - P\| = \|P_2 Q_1\| + \widehat{Q_1 P}$. Since the arc length of any differentiable curve s is defined by $\int_a^b \mathrm{d}s$. In the polar form $\mathrm{d}s = \sqrt{r^2 + \left(\frac{\mathrm{d}r}{\mathrm{d}\theta}\right)^2} \mathrm{d}\theta$, it finally comes that the path from P_2 can be computed by

$$\|P_2 - P\| = \int_{\frac{\pi}{2}}^{\theta} \sqrt{r(\phi)^2 + \left(\frac{\mathrm{d}r}{\mathrm{d}\theta}(\phi)\right)^2} \, \mathrm{d}\phi + \frac{ak}{\sqrt{k^2-1}}$$

Since

$$\frac{\mathrm{d}r}{\mathrm{d}\theta} = \frac{r}{\sqrt{k^2-1}}$$

we can write

$$\|P_2 - P\| = \frac{1}{\sqrt{k^2-1}} \int_{\frac{\pi}{2}}^{\theta} \sqrt{(k^2-1)r(\phi)^2 + (r(\phi))^2} \, \mathrm{d}\phi + \frac{ak}{\sqrt{k^2-1}}$$

$$\|P_2 - P\| = \frac{k}{\sqrt{k^2-1}} \int_{\frac{\pi}{2}}^{\theta} r(\phi) \, \mathrm{d}\phi + \frac{ak}{\sqrt{k^2-1}}$$

$$\|P_2 - P\| = \frac{ak}{\sqrt{k^2-1}} \left[e^{\frac{\theta-\pi/2}{\sqrt{k^2-1}}} \right]_{\frac{\pi}{2}}^{\theta} + \frac{ak}{\sqrt{k^2-1}}$$

$$\|P_2 - P\| = \frac{ak}{\sqrt{k^2-1}} e^{\frac{\theta-\pi/2}{\sqrt{k^2-1}}}$$

Appendix B. Extension to anisotropic diagrams.

The discrete drawing is obtained in following the path for which $|P_1 Q| = |P_2 Q|$ by small increment since we know at each hidden point that the increment is along the tangent of the path. At point Q, we are at a distance d^n from P_1 and from P_2. We assume that speed from P_2 is always greater than from P_1 otherwise the path is not a spiral and P_1 is not enclosed by P_2.
We use a constant distance increment dx to follow the bisector of P_1 and P_2. We assume that P_1 and P_2 are associated with metric $\mathcal{M}_1$ and $\mathcal{M}_2$ respectively. Fig. B.28 illustrates one step of this method.

1. the unit length in the direction of $\mathbf{v}^n$ is $l_2 = \sqrt{\mathbf{v}^t \mathcal{M}_2 \mathbf{v}}$

2. the corresponding distance increment is $dl = \frac{dx}{u^n}$
3. the temporary point $Q' = Q^n + \mathbf{v}^n dl$ is built by following the tangent vector.
4. the radial vector is generated from P_1: $\mathbf{r}^{n+1} = \overrightarrow{P_1Q'}$. $\mathbf{r}^{n+1}$ is normalized in the euclidean space.
5. we can set $d^{n+1} = d^n + dx$.
6. the unit length in the direction of $\mathbf{r}^{n+1}$ is $l_1 = \sqrt{\mathbf{r}^t \mathcal{M}_1 \mathbf{r}}$.
7. the final point is obtained by $Q^{n+1} = P_1 + \mathbf{r}^{n+1} \frac{d^{n+1}}{l_1}$.
8. finally, the tangent vector is updated by setting $\mathbf{v}^{n+1} = \frac{\overrightarrow{Q^n Q^{n+1}}}{\|\overrightarrow{Q^n Q^{n+1}}\|}$.

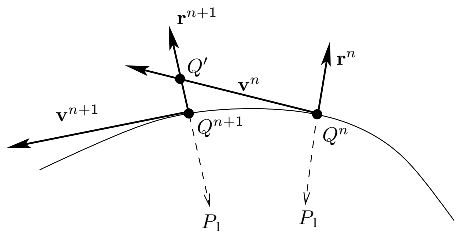


Figure B.28: Anisotropic scheme.