Project report

1 Overview

The goal of this project was to develop an extension for the drawing application Ipe that allows to draw the Farthest Line Segment Voronoi Diagram (FSVD); such an extension is called an Ipelet.
The Computational Geometry Algorithms Library (CGAL) for `C++` offers a good support to develop Ipelets.
This project's repository is available on Github at https://github.com/Spyridox/UROP2017-FSVD.

1.1 FSVD

The Farthest Line Segment Voronoi Diagram is a planar arrangement constructed for a set of line segments. It has faces, edges and vertices and it represents the area of points that are farthest from a given segment, for all segments (for some, this area might be empty); for some segments, this area could also be disjoint, for example constituted by two separate parts. The edges of a FSVD always form a tree, and are therefore never disconnected (unlike the Nearest Line Segment Voronoi Diagram, where the edges can be in up to $n-1$ disconnected parts (for a diagram for n segments).
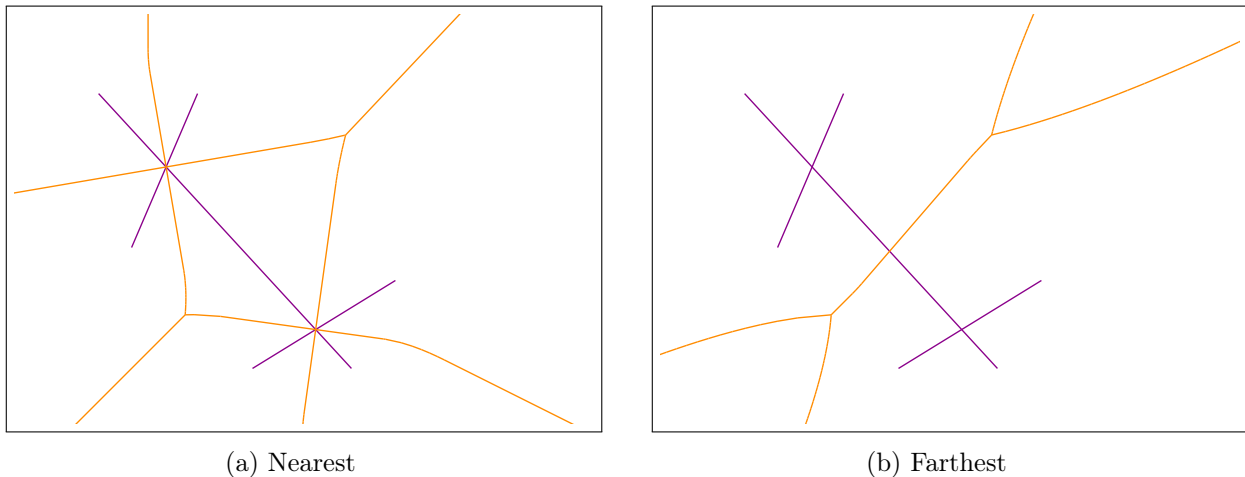See examples below, with segments in dark and the diagrams in orange (figure 1).

(a) Nearest

(b) Farthest

Figure 1: Voronoi diagrams with line segments

1.2 CGAL and Ipelets

In this project two new functions were added to an Ipelet that already featured functions to construct Voronoi diagrams of points, Voronoi diagrams for segments, points and polygons using the L_∞ metric, farthest color and Hausdorff Voronoi diagrams.
The two functions added are to construct, under L_2 metric, the nearest or the farthest Voronoi diagram for line segments.
For both cases the strategy used is to get the distance function of each segment as a surface, then take either the upper or the lower envelope of these surfaces (for FSVD and SVD respectively).

1.3   The envelopes strategy

For a single segment on the xy-plane, for every point plot on the z axis the distance of that point to the segment. This creates a 3D surface. For a point, this surface is simply a cone; for a segment, it's two haflcones originating at the endpoints of the segment and two planes originating from the inner part of the segment (figure 2).
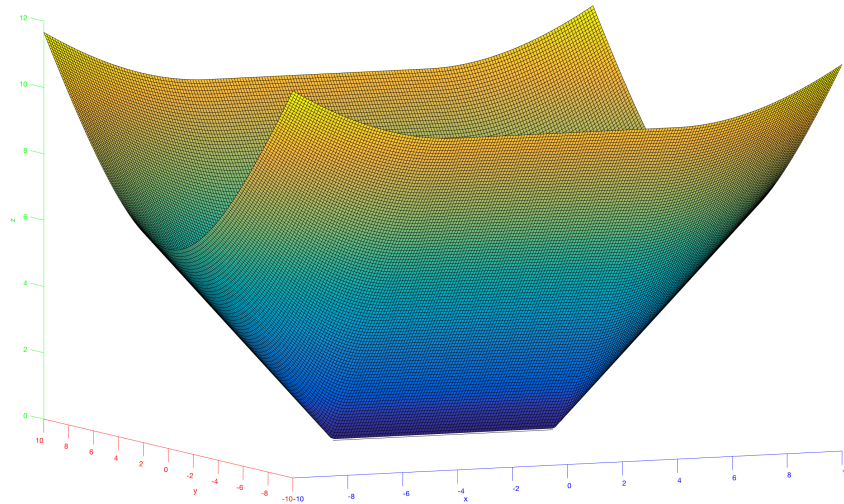

Figure 2: Distance function surface for a segment[1]

For two distinct segments, these distance function surfaces intersect, and their intersection projected back onto the xy-plane is simply the planar euclidean bisector of the two segments.

For many segments, given all their distance function surfaces, if the lower envelope is taken then the xy-projection of this envelope is the nearest Voronoi diagram, whereas the xy-projection of the upper envelope is the farthest Voronoi diagram.

2   Envelope package and `Envelope_diagram_2<EnvTraits>`

CGAL features a package to compute projections of envelopes. The methods `upper_envelope_3` and `lower_envelope_3` output their resulting xy-projection diagram in a `Envelope_diagram_2` object.

This diagram class is parametrised by a traits class that must define the geometry it handles; to do so, it needs to support computation of the projected boundary of a surface, the projection of the intersection of two surfaces, as well as other functions to determine the z-order of two surfaces in specific cases (more specifically, the traits class must be a model of the concept `EnvelopeTraits_3`, specified on the CGAL documentation).

There are already traits classes to support construction of envelopes of spheres, triangles and planes, but these did not include what we needed for FSVD, so a new class needed to be developed.

---

[1]created using MATLAB script by Martin Suderland

2.1   Traits class `L2_segment_voronoi_traits_2<ArrTraits_2>`

The `EnvelopeTraits_3` concept allows the surfaces to be any kind of object, as long as the required functions do their task. Because of this, for FSVD (and SVD too, they can use the same traits class) the surfaces are simply the segments themselves, since they contain all the information needed for the computations.

In the function that requires to compute the projected intersection of two surfaces, we can in fact simply compute the planar euclidean bisector of the two segments.

The bisector is formed, in the general case, by unbounded rays, segments and parabolic arcs (figure 3).
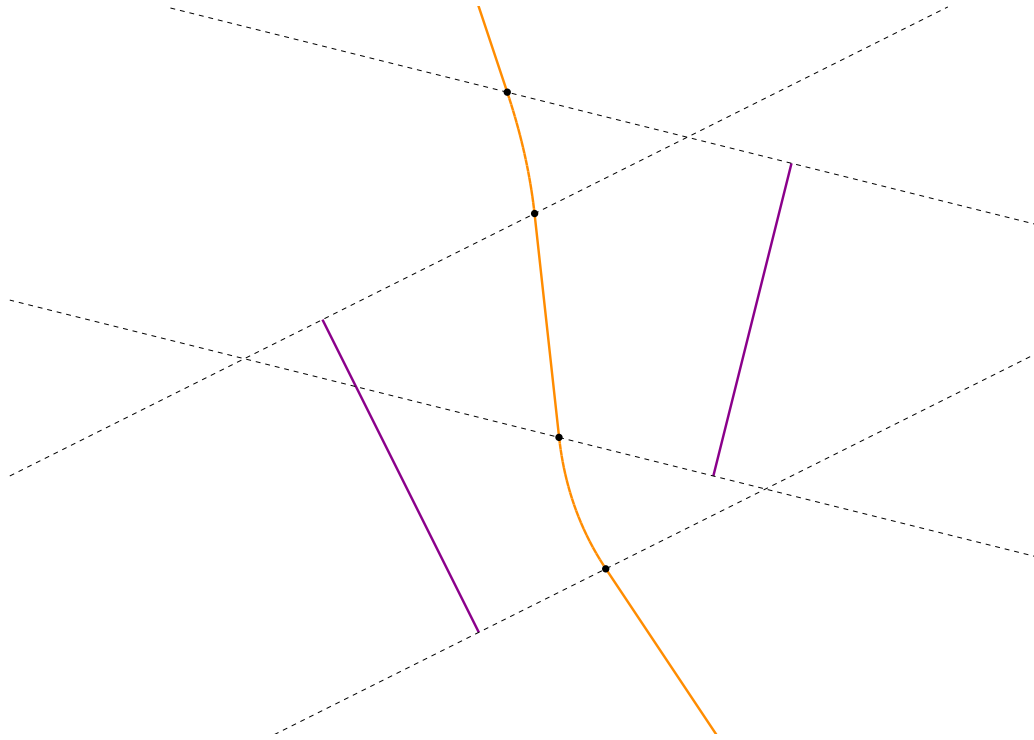

Figure 3: Different parts of the bisector

Notice how the parts of the bisector are delimited by the lines orthogonal to the segments at their endpoints. For a single segment, these two lines divide the plane in three parts: the points closest to the source of the segment, the points closest to the inner part of the segment, the points closest to the target.

The model of the concept `EnvelopeTraits_3` that was developed is `L2_segment_voronoi_traits_2` and is parametrised by an Arrangements class from which it inherits, similarly to how the already available model `Env_sphere_traits_3` for envelopes of spheres does; again similarly, the Arrangements class used is `Arr_conic_traits2`.

This Arrangements class defines the geometry of the edges of the diagram, as well as methods for computation of intersections and predicates.

2.2   Arrangement traits class `Arr_conic_traits2`

This class was chosen because it supports construction and computation using conic curves, needed because of the parabolic arcs in the planar euclidean bisector of two segments; such arcs are present in

the part of the bisector that is closest to one of the segments' inner part and to one the other segment's endpoints (see figure 3).

The curves need to be closed conic arcs lying on a supporting conic curve defined by the equation:

$$rx^2 + sy^2 + txy + ux + vy + w = 0$$

According to the specification (available on the CGAL documentation), the coefficients (r, s, t, u, v, w) need to be all rational numbers. This is because by enforcing this, all intersections between such conic arcs are guaranteed to be algebraic numbers of degree at most 4, and the traits class is optimised for operations with these numbers.

If the supporting conic is not a closed curve such as a circle or an ellipse, and it our case it never is (because it's either a parabola or a line for the straight parts of the bisector), then two endpoints have to be provided. This poses a problem with the unbounded rays (the first and last parts of the bisector, again see figure 3), because they only have one endpoint.

To obviate this, the rays were cut off at an arbitrary distance: imagine this as if the distance surfaces were not unbounded, but were all limited inside the same huge square. Because of this, the traits class `L2_segment_voronoi_traits_2` also constructs a projected boundary for all surfaces (the same for all of them).

2.3 `Parabola` class

A `Parabola` class was implemented to support construction of parabolas, computation of intersection with lines, construction of arcs on the parabolas and construction of tangent lines at given points on the parabola. Other predicates such as `has_on` were also implemented.

The only constructor for a `Parabola` object takes a focus point and a directrix line. These are defined as the endpoint of one segment and the supporting line of the other segment accordingly (see figure 4, parabola in green).

3   Bisector computation

The main part of the project was to implement the functor `Construct_projected_intersections_2` which, as the name suggests, must construct all parts of the xy-projection of the intersection of two given surfaces. As said before, for FSVD this consists of constructing the bisector of two segments.

The approach used is to first find the unbounded rays and their endpoints.

Then, iteratively construct the rest of the bisector, the internal parts, starting from one ray's endpoint until the other ray's endpoint. This is true for the case in which the segments are disjoint; if the segments are intersecting, then there are four unbounded rays, and the bisector is constructed in two parts that "touch" at the intersection of the two segments.

For both segments, the lines orthogonal to each endpoint are kept saved for the whole computation, since they determine all separations between the bisector parts. In the code, these lines are called `Delimiter_lines`.

3.1   Unbounded rays

To find the unbounded rays we have to find the bisector of all pairs of endpoints of the two segments that are extremal in the set of the four endpoints of the segments. That is, we find the convex hull of the set of endpoints of the segments, then we construct a polygon with these points. Walking through the edges of this polygon, if the edge is not one of the two segments, trace the orthogonal line through
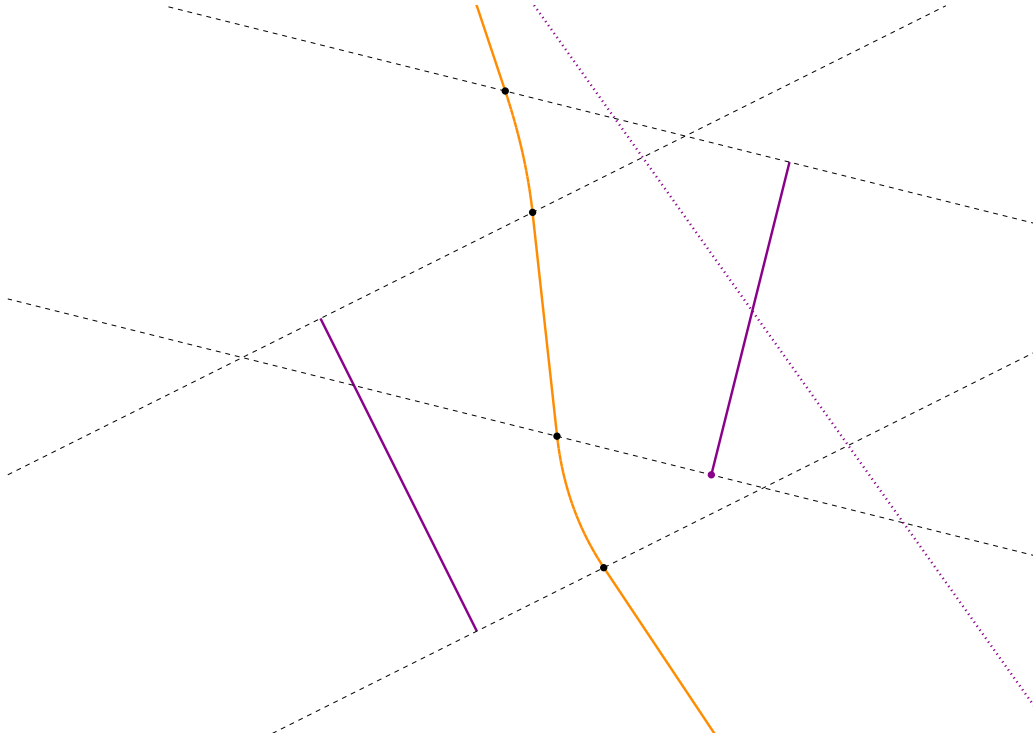
Figure 4: A parabola defined by a supporting line (dotted) and the highlighted endpoint

the midpoint of this edge (which is the bisector of the endpoints of the edge) directed towards the outside of the polyogn.

To find the endpoints of a ray, take the farthest intersection of the line (in the direction of the line) with the four `Delimiter_lines`.

3.2   Internal parts

To find the internal parts we implemented the function `construct_bisector_from_point_to_point`; it takes the two segments, a start and an end point (they are the endpoints of two rays).

The computation is iterative, and keeps as status a current point (`curr_pt`) and a current direction (`curr_direction`); it stops when `curr_pt` is the same as the end point. At each loop, the algorithm has to decide what kind of segment the next internal part of the bisector is (as an `enum Bisector_type`):

    ▷ a parabolic arc   (PARABOLIC_ARC)

    ▷ a segment part of the bisector of the two supporting lines   (SUPP_LINE_BISECTOR)

    ▷ a segment part of the bisector of two endpoints   (ENDPOINT_BISECTOR)

To do this, based on the `curr_direction` and starting from `curr_point`, shoot a straight line until the next intersection with the `Delimiter_lines` is found. Then take the midpoint between `curr_pt` and the next intersection found. Determine where this midpoint is, based on the orientation of the `Delimiter_lines` (they always have their segment on the negative side) and on the predicates `has_on_positive_side` and `has_on_negative_side`.
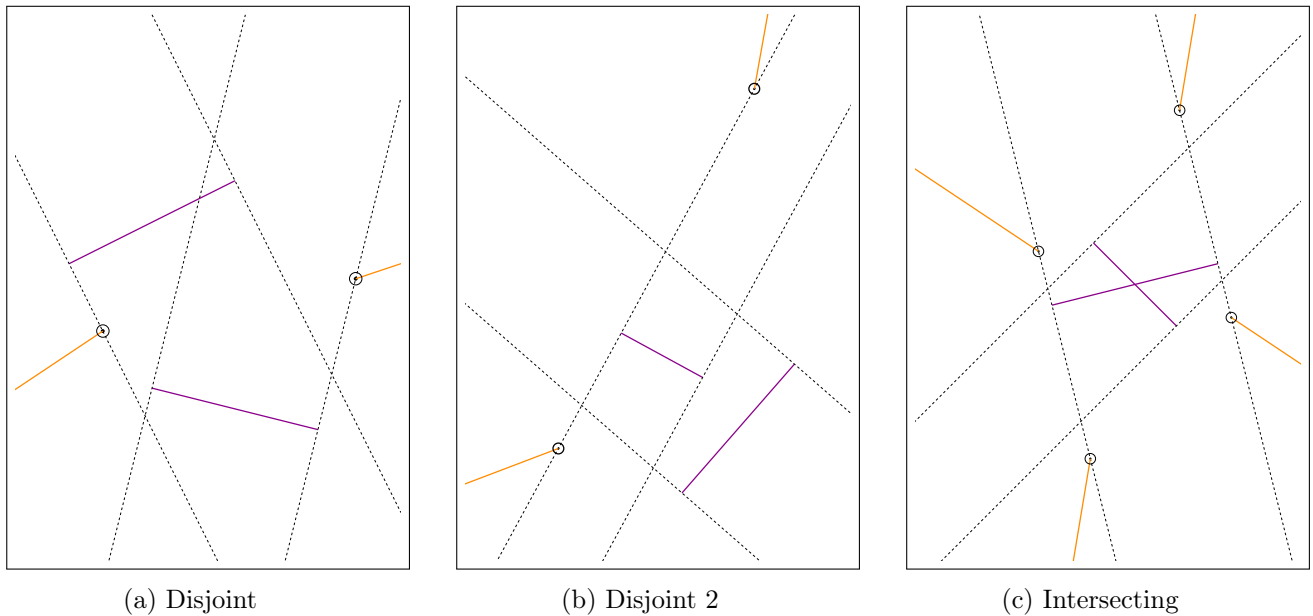
(a) Disjoint   (b) Disjoint 2   (c) Intersecting

Figure 5: The unbounded rays of the bisector in two cases (in orange)

3.3 Challenges and solutions

The main problem with the computation of the bisector were the parts that lie on the bisector of the supporting lines of the two segments. This is because even if the two segments lie on lines with rational coefficients, the bisector might potentially have algebraic coefficients, as shown in figure 6:

As said before, the class `Arr_conic_traits2` allows to construct only `Curve_2` objects that lie on curves that have rational coefficients. An algebraic segment has as supporting curve a line that has algebraic coefficients.

To save it, it needs to be slightly approximated. To do it, the two endpoints of the algebraic segment are moved sligtly so that the supporting line is either translated or rotated, but it still intersects the previous and the next parts of the bisector (in the general case they are parabolic arcs). This is done in such a way that the approximated supporting line has rational coefficients.

To know in which direction to translate/rotate the segment, it suffices to look at the direction of the segment and the orientation of the previous arc and the next arc.

Fortunately then, the `Arr_conic_traits2` class provides a constructor for a `Curve_2` object that takes:

  ▷ the coefficients of the supporting conic of the curve (so the ones of the approximated line)
  ▷ the orientation of the curve (collinear, since it's a segment)
  ▷ the approximations of two endpoints of the curve (the endpoints of the original algebraic segment are given, which are exact)
  ▷ the coefficients of the supporting conic of the previous arc
  ▷ the coefficients of the supporting conic of the next arc

3.4 Drawing

The `CGAL_Ipelets` package has many methods to draw geometric figures; it does not support, however, the drawing of conic arcs. Because of this, the arcs are converted to sets of very small segments. This is done just at the end, when the diagram is already ready to be drawn: the Ipelet asks user input for
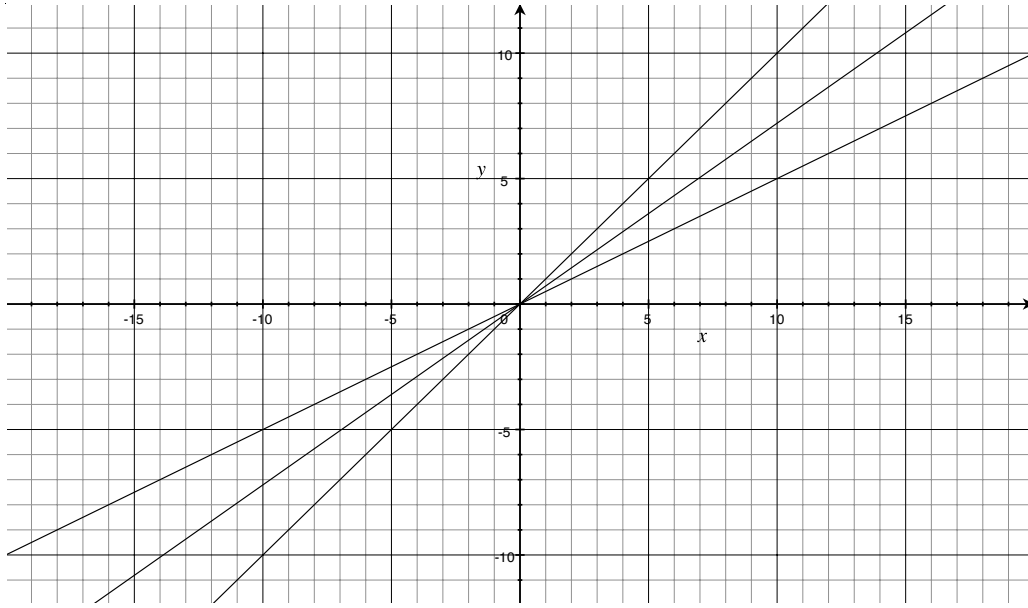
Figure 6: The two lines are $\{x - y = 0\}$, $\{\frac{1}{2}x - y = 0\}$, the bisector is $\left\{ \frac{2\sqrt{5}+5\sqrt{2}}{4\sqrt{5}+5\sqrt{2}} - y = 0 \right\}$

a precision value for the conversion of the arcs to sets of small segments. Lower values take a really long time.

4 Results and future work

4.1 Ipelet functionality

The Ipelet correctly computes the bisector of two segments in general cases.
For three segments, the FSVD and SVD diagrams are usually correct. For intersecting segments, the Ipelet works less well. By checking the console output it is possible to see all checks performed, and everything seems to be working correctly; the graphical output is, however, wrong.

To see the output, simply run the Ipe app from command line; the executable is found at: `/Applications /Ipe.app/Contents/MacOS/ipe`.

4.2 Known limitations

1. The Ipelet does not work if there is any pair of segments such that there are three points that are collinear: this is because the computation of the convex hull to find the unbounded rays would leave the middle collinear point out of the hull, but it should be included for the idea used.

2. If any two segments are touching, the Ipelet also does not work correctly, because one part of the bisector is not included (the one shooting orthogonally to the segment at the contact point with the other segment).

3. The ipelet does not work if there are degenerate segments (points) in the input set. If all segments are degenerate, though, it works fine, since the problem degenerates to a Voronoi diagram for points.

7

4.3 Issues

4. For two segments that are *horizontal*, parallel and of the same length, the Ipelet crashes because of an error in the `Envelopes` package, a violated precondition. Before the crash there is also a warning: `"Explanation: Found two vertices with the same geometric point."`.

5. The Ipelet also draws the fake boundary of the diagram (the one used to bound the unbounded rays). It is only visible by zooming out to the maximum and by panning the canvas a bit towards the sides (in any direction). This is a minor issue.

4.4 Alternatives and possible solutions

First of there are some solutions for the listed limitations and problems (from 1 to 4):

1. Two disjoint segments with three endpoints collinear: this is simply a problem with the computation of the convex hull of the four endpoints. The middle collinear point is left out, therefore one of the unbounded rays is constructed wrongly. To solve this, there are two ways that were only though of but not implemented:

   ▷ create/modify a convex hull algorithm for the set of four points that keeps the middle of three collinear points if it would be on the edge of the hull. With this, the algorithm should work correctly.
   ▷ create a special case (at the beginning of the method, where other special cases are handled) to handle this. This case would only need to get the unbounded rays correctly, and the internal parts could still be computed using the iterative method (by calling the function `construct_bisector_from_point_to_point`).

2. Touching segments: this case should be dealt with as a special case. There was little time to implement this because already the general case had many hidden difficulties and still does not work properly.

3. Mixed input (segments and points): to implement this, it is sufficient to implement the case at line `1377`, with the comment `//TODO bisector of point and segment`. This case was just ignored because it was easy to do and not really important.

4. Weird crash for horizontal parallel and same length segments: this case needs to be debugged by looking into the `Envelopes` package.

5. Drawing of fake boundary: to avoid drawing this boundary one needs to correct the method `on_boundary` at line 230 in the file `bisectors.cpp`. This function simply checks if the edge of the diagram to be drawn is a piece of the drawing itself, and if it returns true the edge is not drawn.

   The current implementation was written really fast and fails in a special case, when the diagram contains a single line that starts and ends on the boundary itself.

4.4.1 `Arr_linear_segment_traits_2`

As said, the `Arr_conic_traits_2` class only supports bouded curves, and the unbounded rays needed therefore to be converted into long segments by introducing an arbitrary boundary for all distance surfaces.

To avoid this, it would be possible to use the more recent class `Arr_linear_segment_traits_2`. This class allows the curves to be any kind of curves of the form:

$$\sum_{i+j \leq n} a_{ij} x^i y^j = 0$$

This of course includes conic arcs. The coefficients a_{ij} need to be integer numbers (so also rational, by just multiplying all of them to make them into integers), so the approximation of the algebraic parts of the bisector would still be necessary.

This, however, would allow to use unbounded rays; it is unknown if the current implementation with the fake boundary actually causes problems.

This approach was at first discarded because of the complexity of the `Arr_linear_segment_traits_2` class; it could, however, even be more efficient.

4.4.2 Own Arrangements class

Another possibility would be to implement an own model of `EnvTraits` containing also an implementation of the Arrangements class methods. Doing so, it might be possible to remove the approximation by allowing segments lying on algebraic curves. This could however increase the mathematical complexity of the computation of intersections.