Project report

List

1. A timeline of all steps and approaches done.
   (a) You should mention all issues faced one by one.
   (b) How you attempted to solve them initially and why it didn't work out.
   (c) How you solved them, finally.
   (d) For all issues not solved, mention them clearly.
   (e) Why you didn't manage to solve them, what were the main difficulties.
   (f) How would you have solved them if you had more time (drop ideas).
   (g) Special cases you didn't have time to tackle and ideas for dealing with them.
   (h) What would further work on this topic include.

   In general keep in mind that someone would ideally read your report and clearly understand what your problem was, how you dealt with it, which difficulties you faced, how did you tackle them (or not) and how your work could be extended.

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).

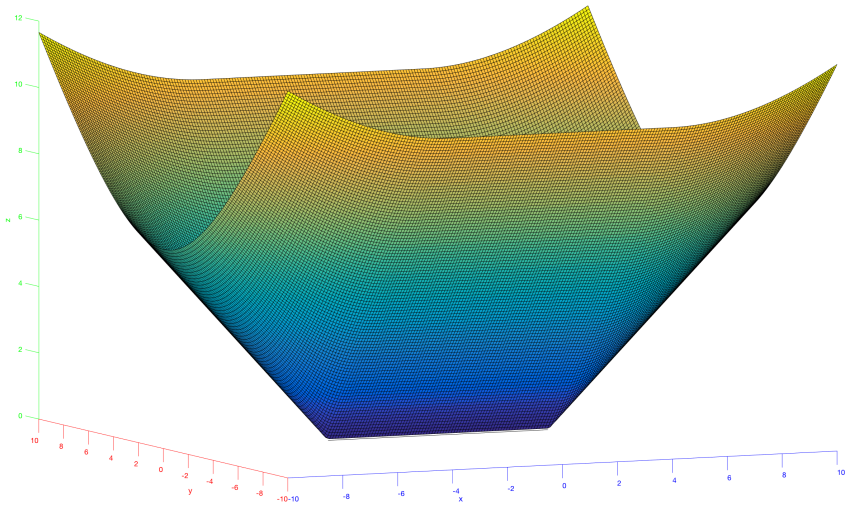


Figure 2: Distance function surface for a segment[1]

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.

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.

[1] created using MATLAB script by Martin Suderland

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).

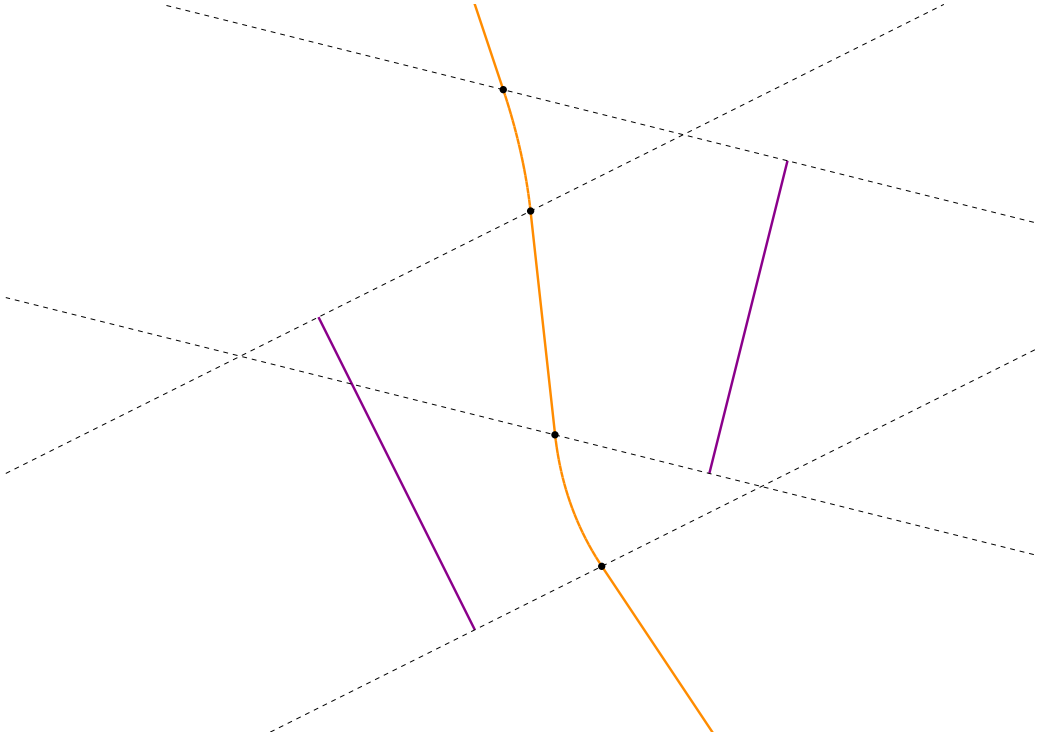


Figure 3: Different parts of the bisector

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 and construction of arcs on the parabolas. 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).

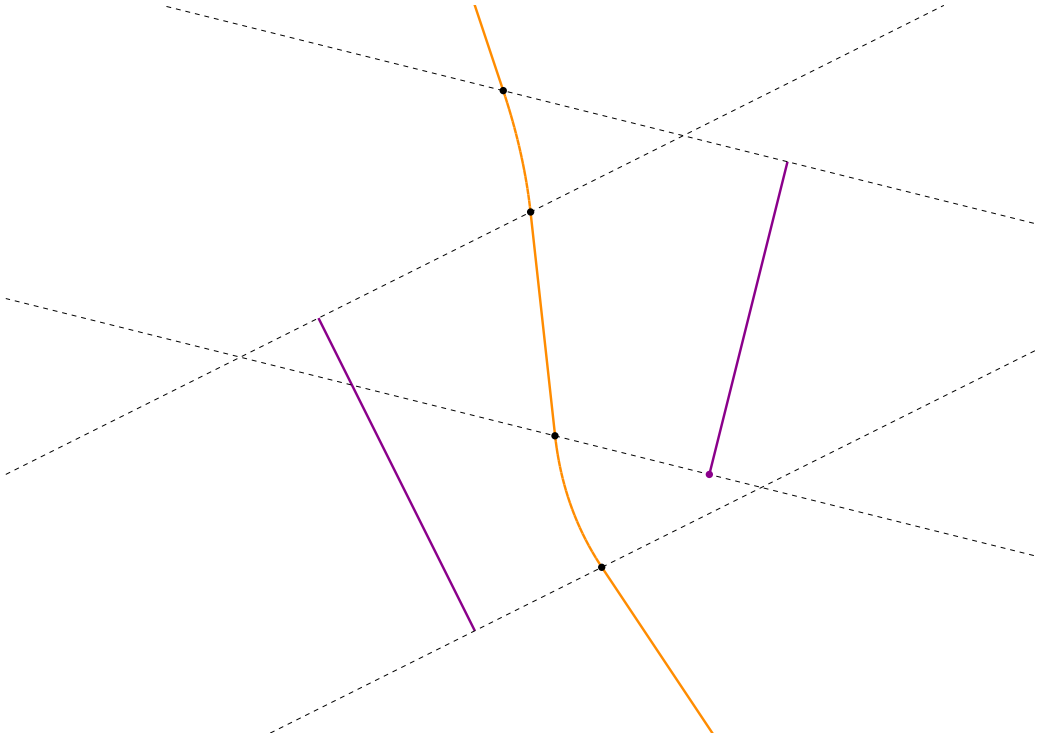


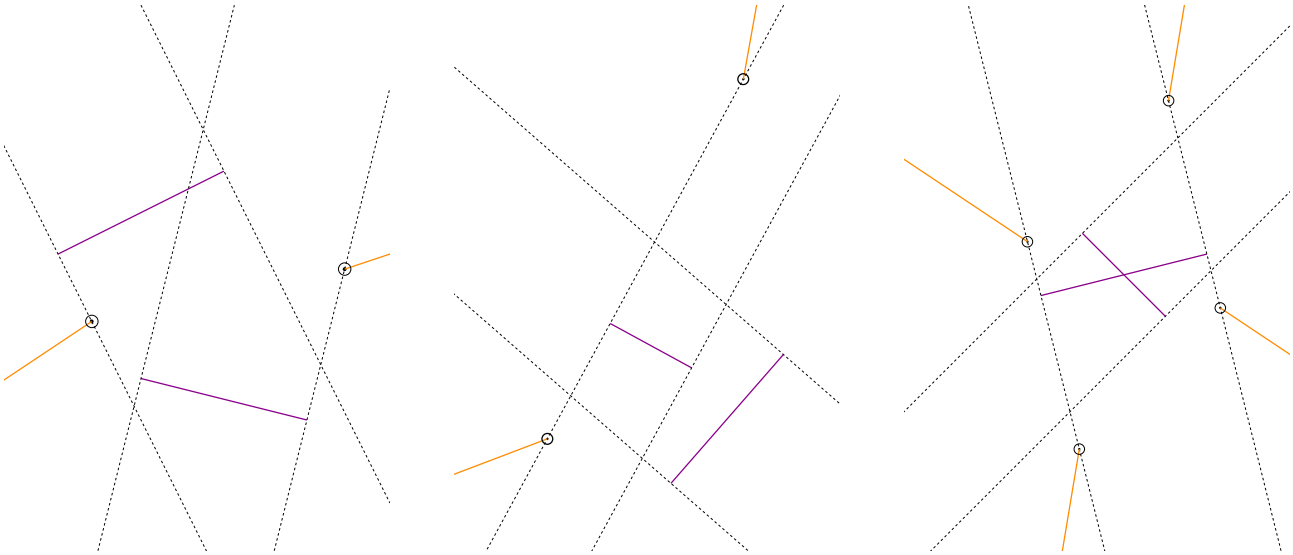
Figure 4: A parabola defined by a supporting line (dotted) and the highlighted endpoint

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



(a) Disjoint (b) Disjoint 2 (c) Intersecting

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

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 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.

3.3 Challenges and solutions

4 Results and future work

4.1 Ipelet issues

4.2 Known limitations

4.3 Alternatives

4.3.1 `Arr_linear_segment_traits_2`

4.3.2 Own Arrangements class