Trajectory Visualization

2IL76 - Algorithms for Geographic Data

J.H.T. Aben (0674244)
T. Hurks (0828691)
W.W.T Reddingius (0775368)
J. van Wijgerden (0775460)

April 2015

Contents

1 Introduction

This report describes and motivates the techniques used to create a visualization tool for animal trajectory data, and contains an evaluation of the tool in use. This animal trajectory data is retrieved from Movebank (`http://movebank.org`), an online sharing platform of animal tracking data.

We considered various visualization goals paired with algorithmic solutions. Ultimately we have realized a prototype implementation for one specific goal loosely based on the characteristics of a specific animal trajectory data set. To be more precise, the prototype gives an indication of the popularity of flight routes taken by albatrosses near the West coast of South-America.

The approaches that were not implemented are presented in the appendices. The approach, reason for abandonment and possible improvements are described.

2 Input data

We retrieved our data set from Movebank (*movebank.org*). From *movebank.org*: "Movebank is a free, online database of animal tracking data hosted by the Max Planck Institute for Ornithology. We help animal tracking researchers to manage, share, protect, analyze, and archive their data. Movebank is an international project with over four thousand users, including people from research and conservation groups around the world. [...] The animal tracking data accessible through Movebank belongs to researchers all over the world. These researchers can choose to make part or all of their study information and animal tracks visible to other registered users, or to the public."

All data available from Movebank comes in a default format with possibly some minor additional data. By default, a data set is a list of items containing coordinates (latitude-longitude pairs) with an id, a time stamp, a heading and a ground speed, along with additional research data that is not interesting for the purposes of our tool (like what type of trackig device was used). Additionally, an item is tagged to a 'local identifier', which we assume to be an identifier for the tracking sessions of a specific animal. We make this assumption because the data is presented in such a way that all items tagged to the same local identifier appear subsequently, and according to the timestamps of the items, this sequence is chronological. Making this assumption, we see such a Movebank data set as a collection of multiple animal trajectories.

2.1 Selected data set

Movebank provides a data set selection tool on their website that gives a basic visualization of the selected data set: on a map an arrow is drawn from one item to the next item. Being able to see the basic characteristics of data sets, we shortlisted several data sets. These sets were picked such that the data, in the representation offered by Movebank at least, looks messy and not too sparse, so we have a good opportunity to create a nice visualization.
The data set of our choosing ended up being that of Galapagos Albatrosses, consisting of 28 trajectories which combined have just over 16,000 line segments. The Movebank basic visualization can be seen in Figure 2.1.

Visible properties of this data set are the diagonal movement patterns across the ocean and very dense movevent along the shores. An unintentional beneficial property of this data is that it lies relatively close to the equator, which means the latitude and longitude pairs do not suffer that much from the spherical distortion of planet Earth; when considered as points in the plane (which simplifies distance calculations) the introduced error appears to be negligible.

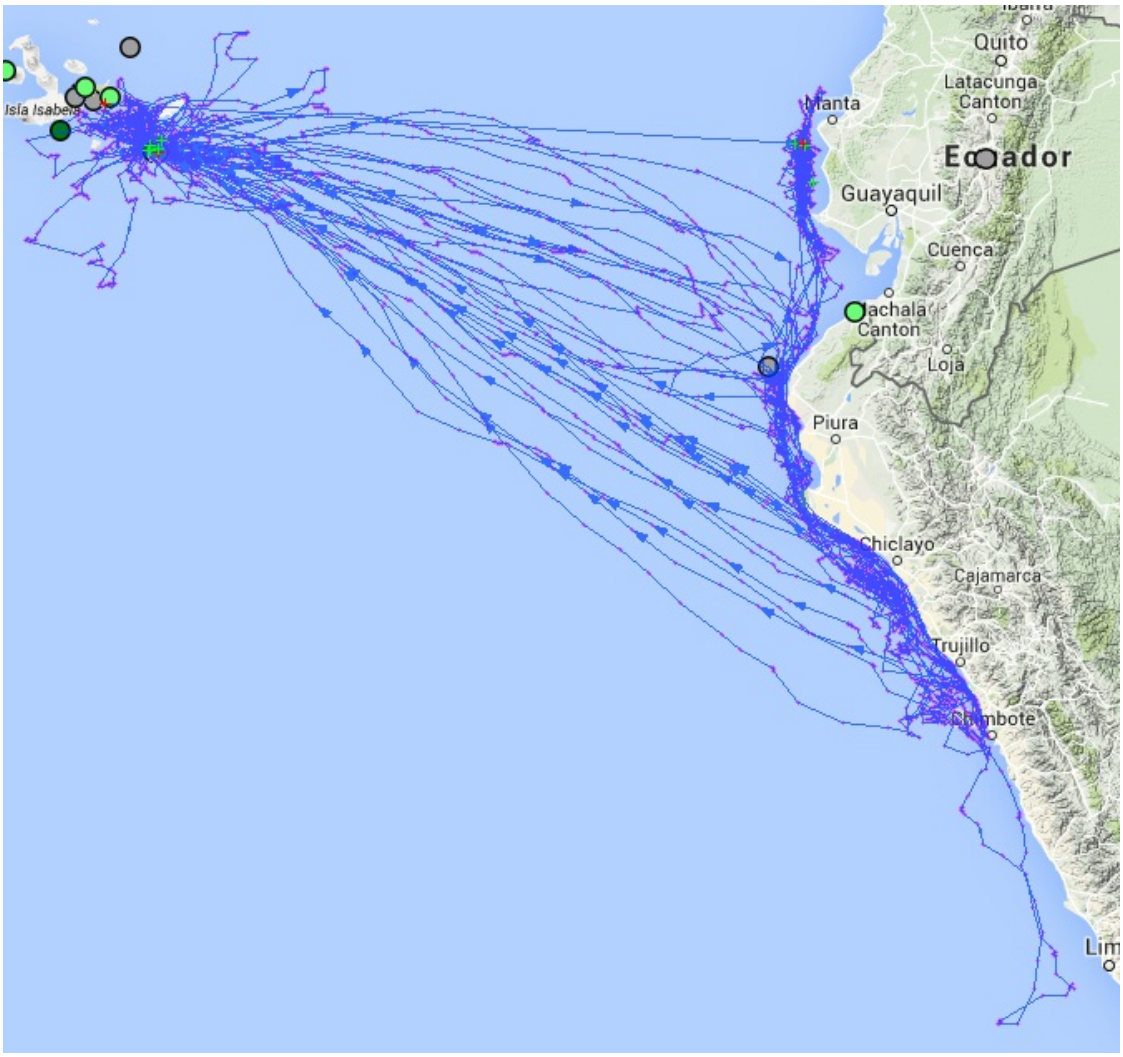

Figure 1: The basic visualisation Movebank provides for the dataset of our choice.

3 Goal of the visualization tool

For our specific data, we are interested in displaying popular flight routes. We want to achieve this by grouping pieces of trajectories that have the same heading and lie close together. This should reduce the amount of segments on screen, making the visualization more clear, while maintaining the characteristics of the data. The popularity of a piece of trajectory is then determined by the amount of trajectories that were grouped to create it. For a more exact definition, please see the next section. Doing so, instead of a wiry mess we obtain a more clean visualization where popular routes can be seen and popular routes on different parts on the map can be compared more precisely. The data as it is presented by Movebank also allows the user to observe different intensities of movement, but with a very coarse granularity; a whole bunch of lines versus a few lines clearly indicates a different intensity, but two 'whole bunches' are much less easily compared than two colors.

For this purpose we do not really need any notion of separate trajectories and are only interested in the segments. This also means that we are not interested in movement, but rather only in presence; the direction of a segment is not relevant. Rather than an arrow, we will regard a segment

is as just a line.

For the algorithms in the appendices the goal was different, namely, we wanted to show densely visited areas, while maintaining a notion of spatial information. The initial goal was to give a visualisation which was spatially informative, i.e. where there was never a trajectory, there would be nothing now, as well as color coded depending on how well visited the area in question was. It would be comparable to a very precise and sharp heatmap. More details are given in the appendices themselves.

4 Algorithmic problem description

This section contains various definitions which capture (or help to do so) the goals set in the previous section. How these definitions are used can be found in Section 5, where the actual algorithm is described.

Our input is a set of trajectory segments, where a segment is a pair of points in the plane, where each point is in fact a latitude-longitude pairs.

We define any line segment a to be ε-close to line segment b if the distance of all points on a to at least one point of b is less than or equal to ε. As can be seen in Figure 2, of segments A, B and C only the latter is ε-close to segment L.

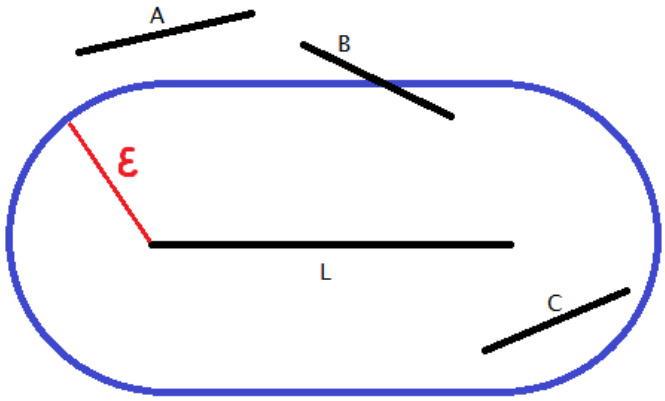

Figure 2: A line segment L with its ε-close zone indicated in blue.

The output of the algorithm (which will be visualized) is a set of segments such that each segment in the input is ε-close to a segment in the output. Ideally, this output set is minimal as to obtain a visualization with the least amount of visible components that still adheres to the goal of the tool. In fact, the amount of ε-close (partial) segments of the input to a segment in the output determines the popularity of the flight route that this segment in the output represents.

The popularity of a segment is encoded by a weight. This weight is used to give the segment a color and a thickness. The color ranges from white to red, where a higher weight means the segment is more red and thicker. The thickness of the segment is determined linearly by the weight, whereas the color follows a square-root scale, as to more quickly transfer from white to red than a linear scale would. We chose a square-root scale because when compared to a linear scale we felt the visualization looks more like what we would expect the output to be.

5 Algorithm

When trying to tackle the problem of visualizing the trajectories in the 'nicest' possible way, we came up with many different possible approaches using very different perspectives on the problem. Most of these attempts sadly failed due to problems with either time complexity, or simply because the algorithm would be too complex to actually implement. More specifically, we ended up making either complicated case distinctions, or developing a complicated list of geometric constructions that eventually was far too slow for our purposes. These failed approaches are described in the Appendix, as they do lead to interesting insights with regards to the problem. It should be noted that only the algorithm in appendix D has opportunities to be completed at some point. The other 3 have inherent design flaws.

Eventually we settled for an algorithm focused on visualizing popular flight routes of the albatrosses. This algorithm was actually implemented and will be explained more in depth in this section.

The main idea of the algorithm is to iteratively choose a segment s from the data (longest first) and divide this segment in weighted intervals I, corresponding to the amount of (parts of) segments that are ε-close to this interval on this segment. To illustrate, consider Figure 3.

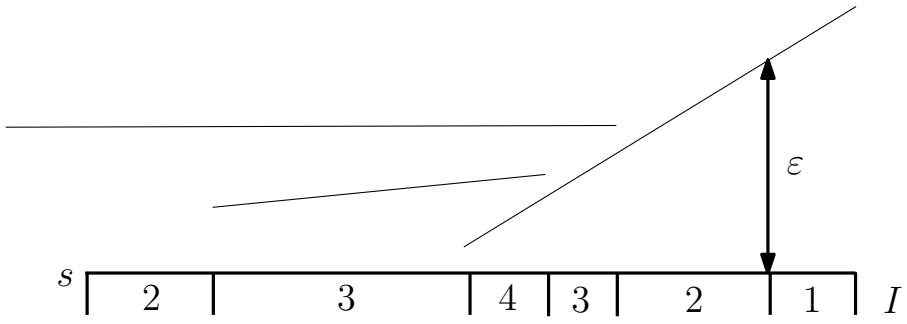

Figure 3: Illustration of how I is determined. Note that the last interval in I has weight 1, because the rightmost part of the rightmost segment is more than distance ε away.

The idea is then for each of these ε-close segments s', to cut off the subsegment s'' of s', which can be projected on s. If s'' is simply the same as s', then we simply completely remove s' from the data. If s'' is only a part of s', then we carefully attach the remaining part of s' to s. More specifically, we attach this remaining part to the projection of s'' on s. Finally, we update the intervals of s accordingly. To illustrate, consider Figure 4.

Finally, after all segments are handled, we use the intervals on each segment to draw lines with different thickness and colors, depending on the weights we have for each interval.

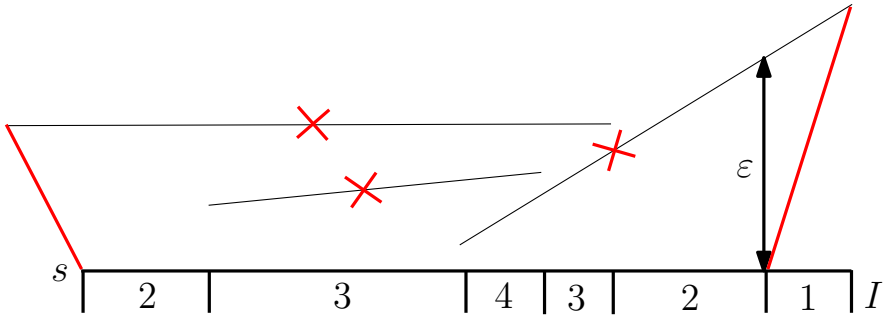

Figure 4: Illustration of how all ε-close segments to s are manipulated. All original segments are removed and the red segments are added.

6 Implementation

6.1 Dealing with a geographic coordinate system

During implementation, we simplified our computations by ignoring the fact that the input segments are specified as longitude-latitude pairs. We simply treat these as x, y pairs in 2D euclidean space. It would appear that the specific data set we chose happens to be located between +0.1821983 and -12.794643 degrees latitude. This means that the distortion we introduce by treating them as x, y coordinates in 2D euclidean space is at most 3%, which we deem acceptable.

6.2 Preprocessing the data

Because we have a rather large dataset and a rather slow algorithm which runs inside a web browser, we simplify the data before running the algorithm on it. This simplification is done in two steps: first, we run the simplification algorithm by Driemel et al, using some ε_s. Next, we perform a similar algorithm, except that we now look at the angle between the segments. This algorithms makes sure that all bends in our data are at least α.

Furthermore, as an optional preprocessing step we have a way of making sure that also all segments are at most 2ϵ in length, so that any bias based on length would be kept in check. In the tool, this option is called 'Equalize'.

As can be observed by comparing Figure 5 and Figure 6, the simplification does not visibly change the shape of the input. In fact, the simplification reduces the amount of segments from about 16,000 segments to about 3,200 segments, which is a great improvement.

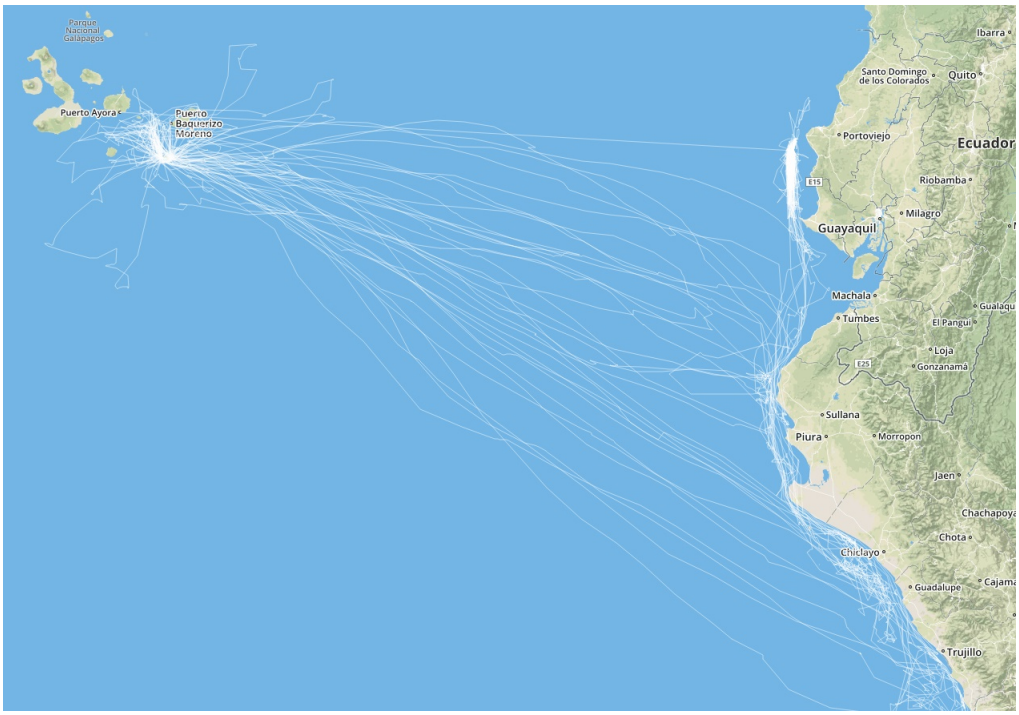

Figure 5: Raw input data

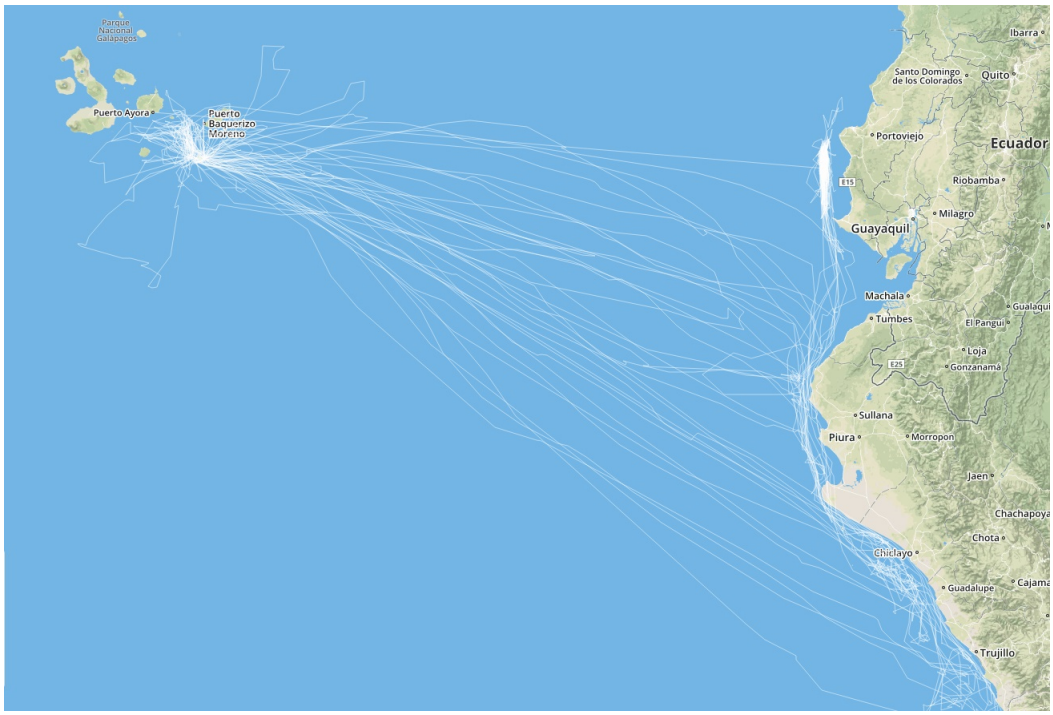

Figure 6: Simplified input data

6.3 Running time

The algorithm has to basically compare each segment with every other segment, and for each comparison we have to perform at most two insertions into a list of intervals with a possibly linear length. These insertions/searches are linear in the implementation, and hence, in the very worst case, the algorithm runs in $O(n^3)$ time. However, this is a very pessimistic upper bound and we expect that on most real life input sets, the running time can be expected to be $O(n^2)$.

7 Results

Below are given three visualizations:

1. Figure 7: a visualization that does not run the algorithm and simply shows all trajectories in the data.
2. Figure 8: a visualization that resulted of running the algorithm with $\varepsilon = 0.15$.
3. Figure 9: a visualization that resulted of running the algorithm with $\varepsilon = 0.25$.

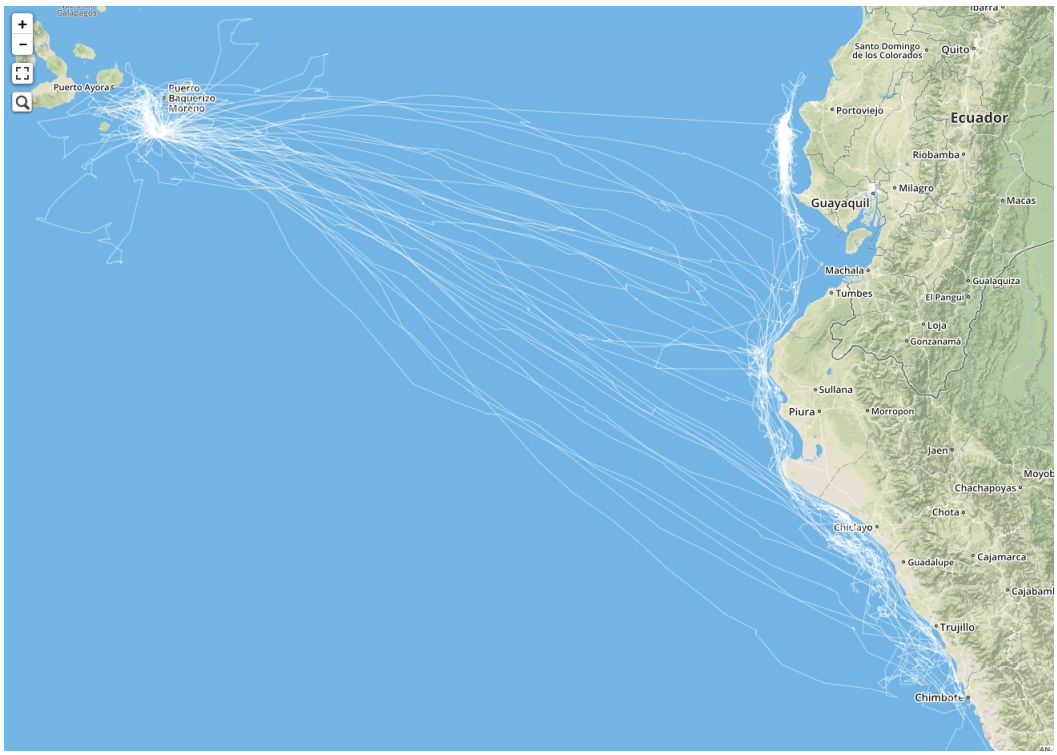

Figure 7: Raw visualizing of all trajectories

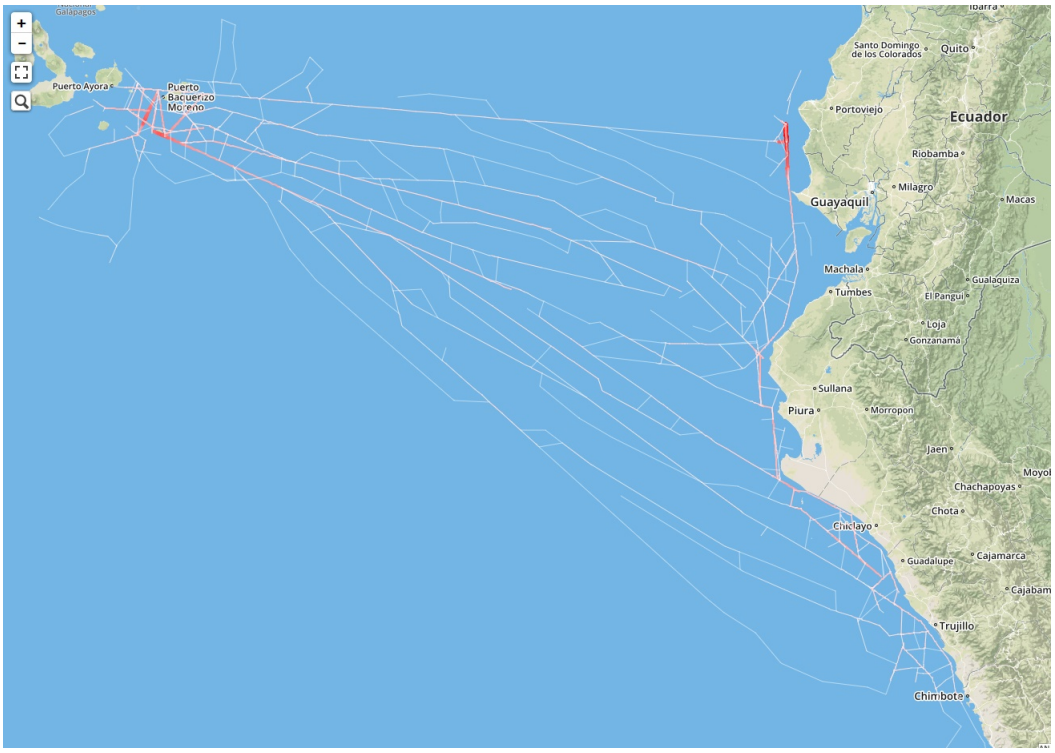

Figure 8: Running the algorithm with $\varepsilon = 0.15$

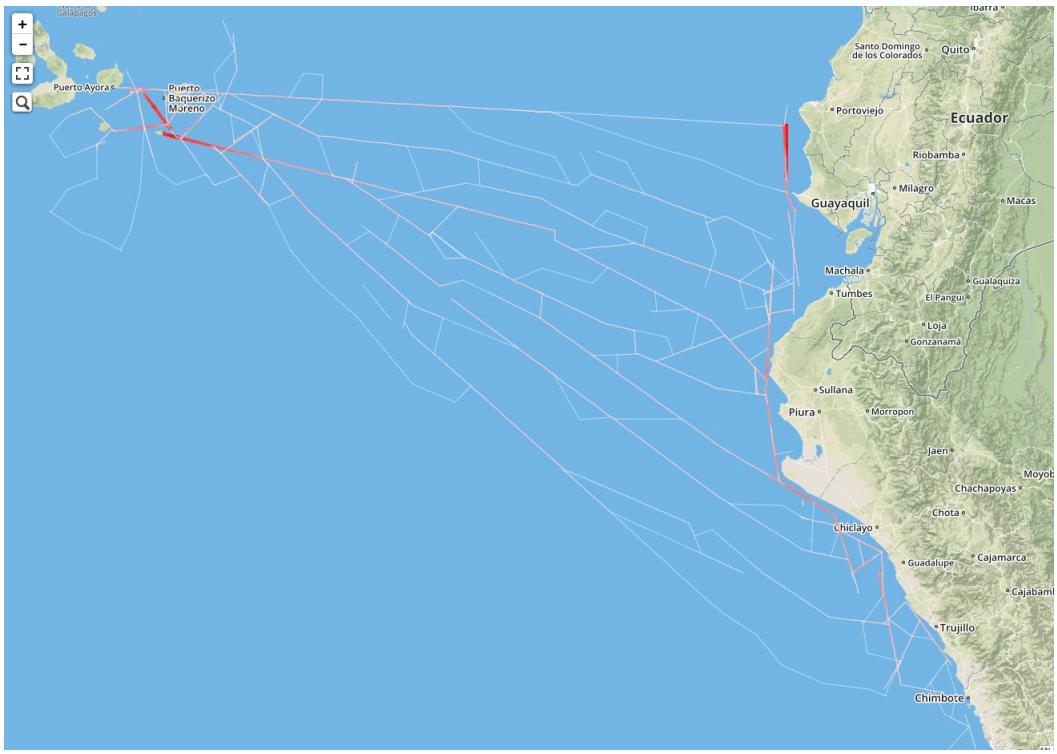

Figure 9: Running the algorithm with $\varepsilon = 0.25$

What is immediately clear from these figures is that the higher ε, the clearer the figure becomes with less and less lines. However, as you would expect, this does lose a lot of information and making meaningful observations using the visualization becomes harder and harder. Furthermore, even in the figure with ε 0.15, we can see how sometimes segments can be observed, that do not seem to match with any of the segments in the original data. This is caused by the incremental construction of the full set of segments: every time we handle a segment and we redirect one or two other segments (red lines in Figure 4), we create a small error by adjusting the angle of these two segments slightly. These segments can then be slightly adjusted again when handling a different segment, and therefore, the error can grow rather large.

Having noted these artifacts, the figure does seem to achieve what we set out to do, which is a visualization of popular flight routes. The colors along with a subtle increase in line width are a nice and effective depiction of the degree of popularity.

8 Future work

As noted before, all insertions and searches on lists of size n are implemented in a linear fashion and thus take $O(n)$. By using binary search trees, it is possible to reduce these terms to $O(\log n)$.

Furthermore, a more advanced weighting system with interpolation and smarter choices for the colors and line thickness could improve the visual quality.

Finally, as was described in the previous sections, the incremental construction of the segments can cause large errors in some places. If we would have had more time, we would have sought an alternative for this.

9 Conclusion

As intended, the visualization preserves the global characteristics of the input data (diagonal paths across the ocean and much movement near the shores), while reducing the amount of segments on screen. The color and thickness of the segments in the visualization do give a representative image of how many segments in the input data are similar to such a segment.

We have come to realize that is fairly easy spiral to a grand, complex visualization algorithm, which then becomes impossible to implement within the allotted time frame. The relatively simple approach we were forced to implement, however, turned out to perform pretty well, aside from occasional artifacts. However, using the tool, we feel the data itself does not really allow for non-trivial observations to be made. It was hard to come up with a meaningful goal for the tool, which could have possibly led to the lack of power the tool has, in terms of giving new insights due to our specific visualization. To obtain the goal, the tool is not a necessity, but only helpful; the information we want to obtain as specified in the goal can, to a lesser (but some!) extent already be obtained from the raw input, which steals the thunder from the tool. That being said, the tool does produce appealing visualizations that are easy and comprehensive, and to some extent, it does manage to accurately aggregate (parts of) trajectories.

The simplification of the input turned out to not only be fairly easy to implement, but also a great performance increase with little to no change of the shape of the input.

A Algorithm attempt 1 - insertion based

A.1 Desired output definition

The initial goal was to convert the input set of segments to a weighted undirected graph, where each edge would have both a weight and a width. The vertices would denote locations in the plane, while the weight of the edges would be the amount of segments were represented by that edge. The locations of the vertices would be chosen in such a way that the width of the edge (which could be defined at multiple points) would lead to the boundaries of the represented segments.

Stated more formally, every vertex in the graph would be either a start vertex, with degree 1, a normal vertex with degree 2, or a split/merge vertex with degree 3. This would mean that it would be similar to the Reeb graph, except that in our case we disregard time, and allow trajectories to form a group with themselves.

In the ideal case, we would have a continuous width function on every edge. We however went with a simpler approach of only assigning 2 widths to an edge, namely at both endpoints. This would lead to trapezoid-shaped polygons around each edge.
Our mathematical model would have the following restrictions: The weight of 2 edges meeting in a vertex can differ at most ϵ, while at a split/merge vertex would have the restriction that the weight of the 2 merging vertices combined can differ at most $2*\epsilon$ from the width of the merged edge.

As a final note, without going into the details, this graph could be easily visualised, as we require it to be planar, and the width of the edges lead to a trivial mesh of polygons.

A.2 Algorithm description

As the graph will not be created very often we do not invest too much time in coming up with advanced geometric algorithms to get a decently fast solution. We leave that to future work.

The algorithm described informally will look as follows:

1. Start with an empty graph $G = (\{\}, \{\})$
2. add the segments of the input to G in arbitrary order
3. handle the segment s - repeat from step 2.

A.3 Abandonment

As we see below, even when handling the simple cases the case distinction becomes rather big. At some point we decided that this case distinction became too large, and would not yield the desired result. Later on, we also discovered that some of the ways we handle the cases do not work out in every scenario.

A.4 Cases

For handling the segment we have a lot cases to cover. In the cases below we assume that the s only intersects the edges described in the case. We also leave the updating of the width functions implicit, as their shape follows directly from the other parameters.
In the drawings to come, we use the following guideline: Thick black lines are edges, already present in the graph. The thick red line is the segment to be inserted in the graph, with the thin red box being the ϵ box of the inserted segment. The normal blue lines define the graph after adding this segment. For simplicty we omit widths from the illustrations.

A.4.1 No intersections

Let us start by how to handle a segment s whose ϵ-rectangle does not intersect any of the edge-trapezoids. This is the easiest case, as we add a new edge e completely representing s. That is, $e[u] = s[p_1], e[v] = s[p_2], e[n] = 1, e[w(x)] = 0$, where $e[u], e[v]$ are new vertices in the graph.

A.4.2 Edge without trapezoid intersects the parallel lines of ϵ box

The next case we handle is where our segment s intersects an edge e, with for now $e[n] = 1$. We illustrate this case in figure A.4.2

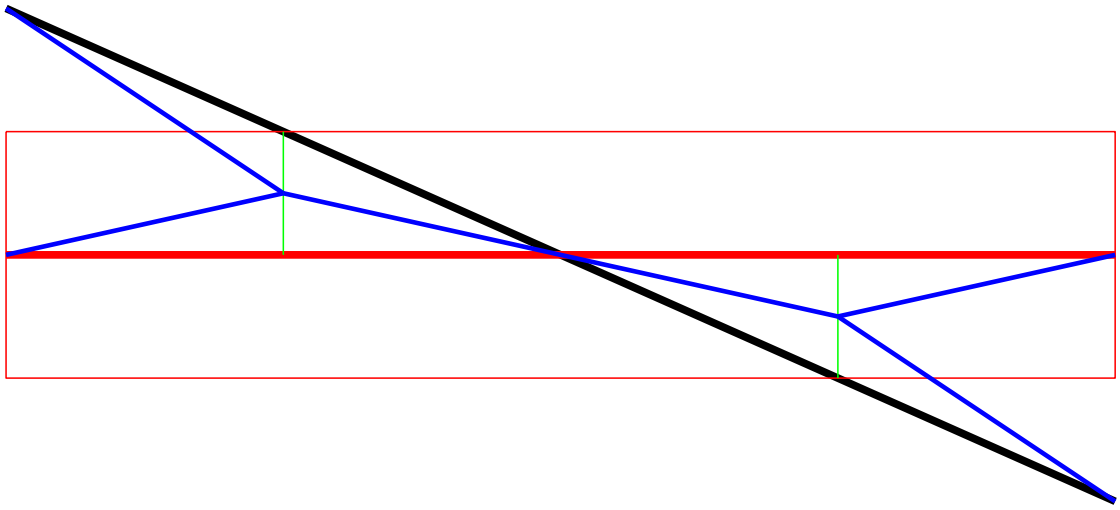

Figure 10: Case 1

As a first step, we find the intersections between the line and the box. (for now let us indeed have 2 intersections) Next, we project these points onto s, thus forming the green lines. We then take the average of the projected points and the intersection points. These define the endpoints of the edge that will represent the part where the 2 segments are grouped, shown as the middle blue line. we refer to these points as p_1 and p_2. We now replace e in the graph by the following set of edges, including all newly defined vertices as vertices in the graph:

$$
\begin{array}{llll}
e_1: & e_1[u] = e[u] & e_1[v] = p_1 & e_1[n] = 1 \\
e_2: & e_1[u] = p_1 & e_2[v] = p_2 & e_2[n] = 2 \\
e_3: & e_3[u] = p_2 & e_3[v] = e[v] & e_3[n] = 1 \\
e_4: & e_4[u] = s[p_1] & e_4[v] = p_1 & e_4[n] = 1 \\
e_5: & e_5[u] = p_2 & e_5[v] = s[p_2] & e_5[n] = 1
\end{array}
$$

In this case, however, we also have a special case, namely that there is only 1 intersection between the ϵ box and e. This means that $s[p_2]$ ends 'inside' e_2. In that case we drop e_3 from the set of new edges, and adjust e_2 accordingly.
Also note that the perpendiculars on e_2 need not to intersect s, as they define a bigger subsegment than the original intersections. In that case, we take the corrosponding endpoints of s, and omit the appropriate edge(s) (e_4 and/or e_5) from the set above.

A.4.3 Edge without trapezoid inside the ϵ box, no intersections

This case is described in figure A.4.3. It may also occur that an edge e with $e[n] = 1$ is contained within the ϵ box of s. As e is within the ϵ box of s, the goal is to solve this with 3 edges.
We obtain our edges by means of the following construction.

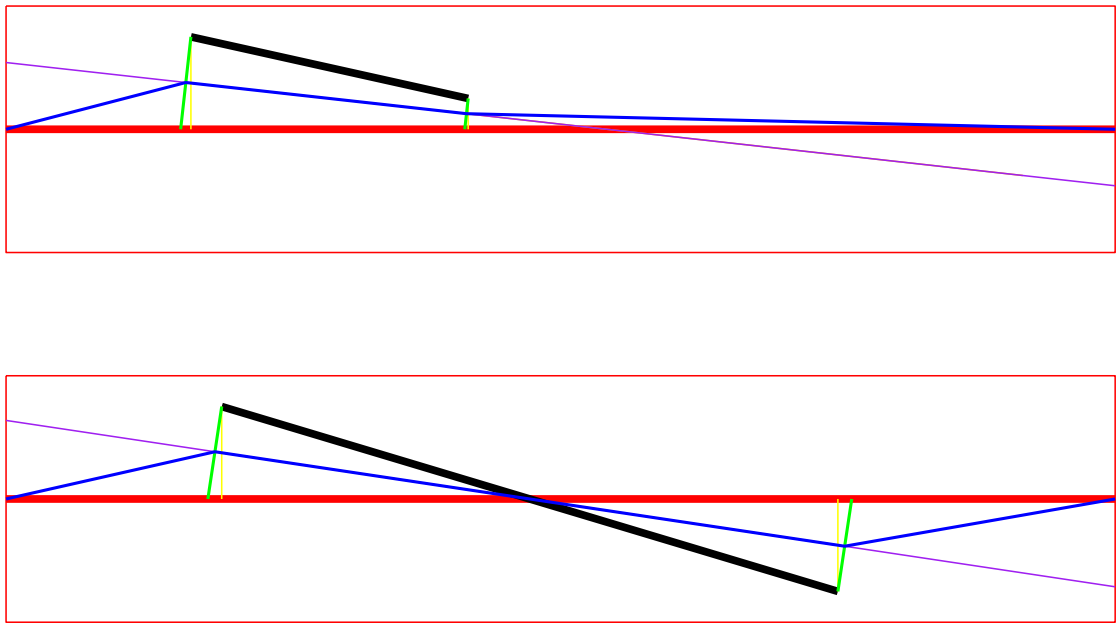

Figure 11: Case 2

Let p_1 be the point on s that is on the perpendicular line to s that intersects $e[u]$.
Let p_2 be the point on s that is on the perpendicular line to s that intersects $e[v]$. These are the yellow lines
Let p_3 be the average between p_1 and $e[u]$.
Let p_4 be the average between p_2 and $e[v]$.
Let l_1 be the line containing p_3 and p_4. This is the purple line
Let p_5 be the point on l_1 where the perpendicular on l_1 goes through $e[u]$.
Let p_6 be the point on l_1 where the perpendicular on l_1 goes through $e[v]$. These are the 2 endpoints of the middle blue section.
p_5 and p_6 are the locations of the new vertices. We now get our 3 new edges, defined as follows:

$$\begin{array}{llll} e_1: & e_1[u] = e[u] & e_1[v] = p_5 & e_1[n] = 1 \\ e_2: & e_1[u] = p_5 & e_2[v] = p_6 & e_2[n] = 2 \\ e_3: & e_3[u] = p_8 & e_3[v] = e[v] & e_3[n] = 1 \end{array}$$

Using this construction we get an e_2 whose perpendiculars contain $e[u]$ and $e[v]$.

A.4.4 Edge without trapezoid intersects the perpendicular lines of ϵ box

Let s' be the segment represented by e.
Remove e. Insert s, giving 1 edge: e'. Insert s'.
e' will now be inside the ϵ box of s'. We already handled this case.
Note that this is possible only because e only represents a single segment.

A.4.5 Edge without trapezoid intersects perpendicular and parallel lines of ϵ box

Not worked out.

A.4.6 Edge intersecting multiple edges or trapezoids

We can reason that for any point on the to be inserted segment s, we can reason that the perpendicular on s through any point will intersect at most 2 edges/trapezoids. Thus, by handling these cases, we have covered all cases. For we reason that we could split the segments so all occurring cases are covered above.

A.5 Running time analysis

The running time analysis is relatively straightforward. We know that the computed graph gets of complexity $O(n^2)$, and we check every element in that graph for every segment we add, which we have $O(n)$, and we thus arrive at a complexity of $O(n^3)$. The practical running time may be improved by adding the edges in order of maximal x coordinate, such that at some point we do not have to check the edges on the left of the graph any more. Other optimizations may be possible.

A.6 Proof of correctness

As we did not finish the algorithm, we do not prove its partial correctness.

B Algorithm attempt 2 - Sweepline

The following algorithm is a sweepline based approach, with a vertical sweepline, from left to right. First we need to process our input to something usable for a sweepline algorithm.
To this end we first walk through the following steps, without going into detail.

B.1 Preprocesing

1. We consider the input as a set of segments $S = \{s_1, s_2, s...n\}$. Each segment s_i is defined by it's 2 points, q_i and p_i, such that $q_i.x \leq p_i.x$.
2. Take the $\epsilon/2$ rectangle around each segment giving us a rectangle set $R = \{r_1, r_2, ..., r_n\}$.
3. For each rectangle we have 4 sides, contained in the set $A_i = \{a_{i1}, a_{i2}, a_{i3}, a_{i4}\}$.
4. Let $A = \cup_i A_i$ be the union of of the sides.
5. Let A^* be the set of lines that are induced by the linesegments of A.
6. Next, we compute the planar subdivision $P = (V, E, F)$ of A^*, with V denoting the vertices (intersections), E the edges (subsegments), and F the faces of P.
7. For each of the $f_i \in F$ we maintain the set $f_i.S \subseteq S$ of segments that have led to the creation of that face.
8. V are the events for our sweepline algorithm. Note that $B \subseteq V$
9. Sort V on the x value of its elements.

During the algorithm we assume that all x coordinates are unique, which corresponds with there being no vertical segments.

B.2 Status structure

Next we define the status structure. The status structure will in some sense maintain the faces intersected by the sweepline, sorted by y value. This basic approach would give us a list of faces, representing some number of edges. If we were to split the list by the faces that represent 0 segments, we would get connected components, whose $\epsilon/2$ boxes (transitively) intersect. These sets thus implicitly represent the data we want to capture as an edge in our graph.
Initially, we want to define our status structure as an interval tree T on the y-value of the connected components C, with each connected component $c_i \in C$, and thus interval $i_i \in T$, have a set of segment that define its existence.

Of course, as the sweepline progresses, the intervals also change, so we need to adept the nature of our interval tree. An important observation here is that each interval's bounds only move linearly between 2 events. Now we could of course update all the intervals at every event. This would however take $O(T)$ time at every event, and we would lose all the benefits the interval tree gives us. Instead we use a ???, which can cope with moving intervals, whilst maintaining good performance.

B.3 Event handling

In order to do meaningful steps whilst handling our events we first need some theory, on how our desired graph $G = (V', E')$ will look like.
The idea is that every vertex of our desired graph is located on one of the edges in our planar decomposition.

B.4 Abandonment

When reasoning and drawing about handling events we came to the conclusion that a sweepline algorithm is not possible, as we need information from future events. The following 2 subsections were reasoned about before reaching this conclusion. Also the claim that every desired vertex of the output graph being on the edges of our planar decomposition was wrong. As well as the reasoning behind intersecting $\epsilon/2$ boxes, which in itself lead to the choice for a planar decomposition.

B.5 Assigning widths

Nowhere in the algorithm so far have we had to discuss the widths of the edges, and they are so far undefined. As we already have our complete graph structure by the point we arrive at this point, and we chose our widths as as trapezoids, we only need to assign 2 widths for each edge e, namely $e[w(0)]$, which is at $e[u]$, and $e[w(1)]$, which is at $e[v]$. These widths would follow from these notions:
Let l_u be the perpendicular to e that goes through $e[u]_\delta$.
Let I be the intersections between l and $e[S]$.
the width is the maximal distance between 2 points in I.
The width at $e[v]$ is computed similarly, only using $e[v]_\delta$.
The notions of $e[u]_\delta$ and $e[v]_\delta$ are the positions on the edge ever so slightly away from $e[u]$ and $e[v]$ themselves.

B.6 Running time analysis

The running time analysis of a sweepline algorithm is relatively straightforward, as it is simply the amount of events times the time it takes to handle an event. The deciding factor in the amount of events is the planar subdivsion, which generates $O(n^2)$ events.
The time it takes to handle an event is dependant on the size of the status structure, how much elements from the status structure we need, and the time it takes to retrieve these elements. The size of the status structure can become $O(n)$, and we find the elements we need, and update them in $O(\log n)$ time. This leads to a total complexity of $O(n^2 \log n)$.

C Algorithm attempt 3 - Geometric construction 1

C.1 Altering the desired outcome

During the the construction of the following algorithm we came to the conclusion that we would get a list of polygons as an intermediate result. We thought it would be silly to try and transform this list to the graph described in section A.1, and later on, during the visualisation step, compute new polygons, containing less information than these intermediate ones.

C.2 Informal description

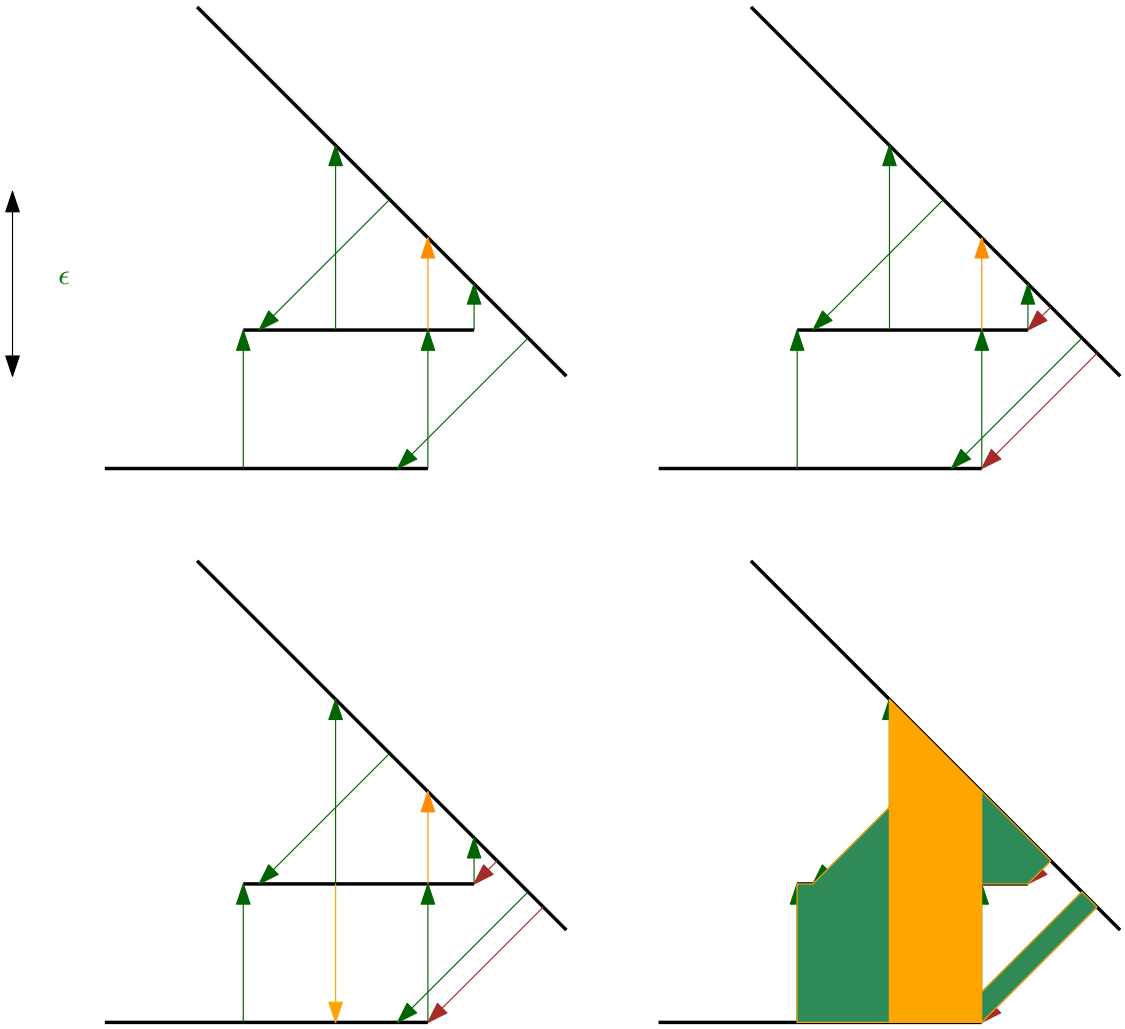

Figure 12: A depiction of how this algorithm would work. Images should be read topleft, topright, bottomleft, bottomright. The top left depicts the first step of castinig the first rays. Step 2 already depicts a hotfix, that ensures that the rays always come in pairs. The third step depicts the ray casting in the reverse direction. The last panel depicts the output that we would have expected from this algorithm, had we finished it. The wierd gap arrises because we use boxes, and do not use ϵ semi-circles to round these off.

The idea behind this algorithm is to find the linesegments which intersect the ϵ box of every

segment. Subsequently, one cast a ray through the found segments, and continue that ray until no more segments are found within ϵ of the last segment. An illustration can be found in figure C.2. Subsequently one would also cast the ray in the backwards direction, to incorporate any segments that would contribute.

C.3 Abandonment

During the construction of this algorithm we found that the entire basis of only casting these rays as a basis to be wrong. Once we discovered this we abandoned this algorithm, and continued with the next algorithm. As we abandoned this algorithm, we discourage the reader from reading the cumbersome list of definitions below, and instead skip to the fourth algorithm. That algorithm follows the same line of thinking, and will be explained more clearly.

C.4 Algorithm description - formal

Let the input be $S = \{s_1, s_2, ..., s_m\}$ be the input set.
Let r_i denote the ϵ-box of s_i.
Let every segment be defined by the points q_i and p_i, such that $s_i = (q_i, p_i)$, and $q_i.x \leq p_i.x$.
Let A_i denote the intersections of s_i and r_i. This will simply be the set of q_i and p_i.
Let $B_{i,j}$ denote the intersection points between s_i and r_j
If $\|B_{i,j}\| = 1$ either q_i or p_i inside r_j. let x be the endpoint of s_i for which this is the case. We now define $C_{i,j}$ as $\{x\}$. If $\|B_{i,j}\| \neq 1$ we define $C_{i,j} = \phi$.
Let $D_{i,j} = B_{i,j} \cup C_{i,j}$. Observe $\|D_{i,j}\| \in \{0, 2\}$.
Let $C'_{i,j}$ denote a set of linesegments (a, b) such that:

- $\|a - b\| < \epsilon$
- a, b are endpoints of segments s_i and s_j respectively
- $\|D_{i,j}\| = 2$

This implies that $\|C'_{i,j}\| \in \{0, 1\}$.
Let $D'_{i,j} = D \cup C'i, j$ Let $E_{i,j}$ be the set of linesegments e_k such that $e_k = (d_k, x)$, where x denotes the intersection of the perpendicular on s_j through d_k, with $d_k \in D'_{i,j}$
Let $F_{i,j}$ denote the endpoints on s_j of $E_{i,j}$.
Let $E = \cup_{i,j} E_{i,j}$ be the union of the linesegments defined above.
For every linesegment $e_k \in E$, let us denote $e_k^0 = e_k$ as the initial linesegment. Also, let δ_k^ϵ denote the vector in the direction long e_k, of length ϵ.
If E contains linesegments that are defined on the same 2 endpoints, toss either one, for efficiency.
Let E^c be the set of e_k^c.
Let e_k^c be a linesement defined as follows, for $c \neq 0$

- Let s denote the sign of c, thus $s = -1$ iff $c < 0$ and $s = 1$ iff $c > 0$.
- If e_k^{c-s} is not defined, neither is e_k^c.
- Let q denote the second point of e_k^c, and let $p = q + s * \delta_k^\epsilon$.
- Let s_l denote the first segment intersected by (q, p), with the intersection point being x.
- If such an x exists, $e_k^c = (q, x)$, if no such x exists, e_k^c is not defined.

Let $e_k^{\min}$ be the smallest c such that e_k^c is defined
Let $e_k^{\max}$ be the largest c such that e_k^c is defined
Let F_k be a list of linesegments, of size $e_k^{\max} - e_k^{\min}$. We define $f_i \in F_k$ as $f_i = e_k^{i-1+e_k^{\min}}$, for $i \geq 1$
Let $F = \cup_k F_k$.

In case there are pairs f_i and f_j of elements of F that coincide, toss the one with the smallest amount of elements.
Let G denote the set of endpoints of S.
Let H denote the set of linesegments defining the convex hull of G.
For every $h \in H$ label it with 0.
Let I_i denote the points on s_i that are intersected by any F_j.
Let J_i denote the linesegments we obtain from splitting s_i at the points I_i.
Each of j_k originated from 2 $F_{k'}$ and $F_{k''}$, or one of the endpoints of the original segment. in case one of the points was an endpoint, label j_k with 1. Otherwise with $\min(\|F_{k'}\|, \|F_{k''}\|)$. Label all of e_k^c with $\|F_k\|$.
Let $J = \cup_i J_i$.
Let K be the set of intersections among F. Let $L = J \cup (\cup_c E^c) \cup H \cup K$.
Let $M = G \cup (\cup_i I_i)$
Let $N = (M, L)$ be a planar graph
Let $O = (M, P, Q)$ be the DCEL induces by this planar graph.
Let every half edge $p_i \in P$ have the label of it's original counterpart stored in $p_i[l]$.
...Here we skip to the next algorithm

D Algorithm attempt 4 - Geometric construction, v2

D.1 Abandonment

We abandoned this algorithm during the implementation and analysis phase. During early attempts to implement this algorithm we found out that the running time would get drastically out of hand, making the practical use of this version rather small. We thus chose for a simpler algorithm, as described in the main sections of this document.

D.2 Algorithm description - informal

The algorithm we have come up with to generate a list of polygons gets rather complicated when doing a formal description. We will start with an informal description, describing the intuition behind the algorithm. The formal, rather cumbersome description follows in Section D.3.

We will do a top down description of the algorithm; the idea behind the algorithm is to generate a list of polygons, that can later be easily verified. In order to achieve this, we wish to create a planar graph, or rather, a DCEL, whose faces will be the polygons we want to output. Each face of the DCEL would have a weight between 0 and 1, which would be an indication of how densely that polygon is populated. This weight can later easily be converted to a color.

To see how we construct the DCEL, we do a procedure for all segments, or rather, for every side of each segment. So consider our segment s, and the perpendicular unit vector on the side of our choosing. We now draw the half-*epsilon* box on our side, and find all other subsegments that are covered by this box. We now pick all these subsegments, and apply the same procedure, using the perpendicular vector from our original segment to draw half-ϵ boxes on these segments. We do this recursively, until we can no longer find any segments.
Next, we cast the rays back to the original segment. Each ray carries the information of the segments it has past. Finally, we take all rays cast in this manner, and compute the DCEL that they induce. At a lot of places we will have 2 overlapping rays. We can there use both rays to put a weight on the half edges of the DCEL.

The output now would become the faces of the dcel, weighted according to the information carried by the half-edges that define the face.
If we were to compare it to algorithm 3, and process the example in figure C.2, we would find the same output, except that the extra step in the second picture would not be necessary, as it is already covered in step 1. That is because this algorithm does not operate on the basis of intersections of lines, but on the notion of overlapping areas.
In figure D.2 we depict a scenario where algorithm 3 would fail, yet algorithm 4 succeeds in producing a nice output.

D.3 Algorithm description - formal

In this section we describe the algorithm in a formal, step-by-step manner. As it is a geometric, and complicated algorithm at that, this description will be hard to follow. It is advised to just stick to the informal description, which conveys the workings of the algorithm rather clearly. The following wall of text is here for completeness, and because it helped during the design of the algorithm.

Let A be our input set of segments.
A segment a is defined by 3 values: $a[p]$; its starting point, $a[d]$; its direction, and $a[l]$; its length. We choose this representation so we can have a segment with length 0, but with a defined direction. Notice that $a[d]$ is a vector of unit length. For shorthand, let $a[q] = a[p] + a[d] * a[l]$ denote

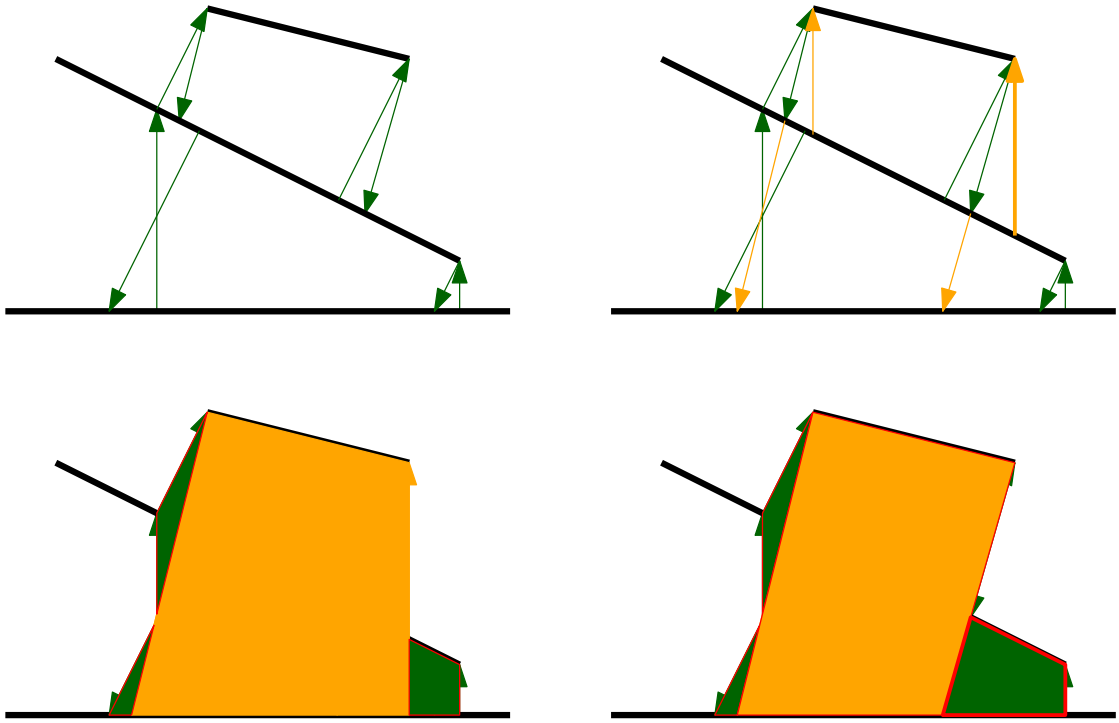

Figure 13: A depiction of how this algorithm would work. Images should be read topleft, topright, bottomleft, bottomright. The first panel shows the first level of arrows. The second panel shows the second level of arrows. The third panel shows the expected output from algorithm 4. The fourth panel shows the output algorithm 3 would give, with that input. The diffrence is due to the fat vertical orange arrow in panel 2.

the other endpoint of a.
We assume our input set is in general position, as well as that no two line segments are perpendicular to eachother.
We now first pre process A to not contain any intersections. At an intersection point we simply slice the 2 intersecting segments into 2 parts, creating 4 new line segments, each sharing 1 endpoint.
Let $B_{i,k}$ be a set of segments, with $B_{i,0} = \{a_i\}$. The endpoints of any element of this set will be contained by some segment in A. Let us denote this segment by $b_{i,k,j}[o]$.
Let for any $b_{i,0,0} \in B_{i,0}$, $c_{i,0,0}$ be the ϵ box around $b_{i,0,0}$.
Let $D_{i,k,j,l}$ denote the endpoints of the subsegment of a_l that is contained in $c_{i,k,j}$.
Notice that $\|D_{i,k,j,l}\| \in \{0, 2\}$. If $\|D_{i,k,j,l}\| = 2$ Then let d_1, d_2 be the elements. Then let s denote the line segment defined by d_1 and d_2. We now have that $s \in B_{i,k+1}$.
For $k > 0$ let us have the line segment $b_{i,k,j} \in B_{i,k}$. Let p be the unit vector perpendicular to a_i that points in the direction of $b_{i,k,j}$. We now define $c_{i,k,j}$ as begin the 'half-ϵ-box, as it is the rectangle defined by $b_{i,k,j}[p]$, $b_{i,k,j}[p] + \epsilon * p$, $b_{i,k,j}[q] + \epsilon * p$ and $b_{i,k,j}[q]$.
Let $e^1_{i,k,j}$ and $e^2_{i,k,j}$ denote the 2 lines that are perpendicular to a_i and go through the points $b_{i,k,j}[p]$ and $b_{i,k,j}[q]$ respectively. Including a_i and $b_{i,k,j}$ this line will intersect k line segments of A. More specifically, it will intersect 1 line segment $s \in B_{i,m}$ for $0 \leq m \leq k$.
Let $f^l_{i,k,j}$ be the intersection points of a_i and $e^l_{i,k,j}$.
Let $g_{i,j,k}$ be the line segment defined by $f^1_{i,k,j}$ and $f^2_{i,k,j}$.
From here on, working with variable names from z downwards, for clarity
Let $Z^1_{i,k}$ and $z^2_{i,k}$ be the sets containing all the linesegments $h_{i,k,j}$, such that for all the elements $z_{i,k,j} \in Z^1_{i,k}$ we have that $b_{i,k,j}$ are on 1 side of a_i, and $Z^2_{i,k}$ be the remainder.

Let $Z^s_{i,k}[d]$ be the unit length vector perpendicular to a_i such that for all $z^s_{i,k,j} \in Z^s_{i,k}$ we have that $Z^s_{i,k}[d]$ points in the direction of $b_{i,k,j}$.
Let $Y^s_{i,k,m,n}$ be a set of segments, with $Y^s_{i,k,m,0} = \{z^s_{i,k,m}\}$, where $z^s_{i,k,m} \in Z^s_{i,k}$.
Let for for all linesegments $y^s_{i,k,m,n,j} \in Y^s_{i,k,m,n}$, $x^s_{i,k,m,n,j}$ be the 'half-ϵ-box' in the direction of $-Z^s_{i,k}[d]$. Note the minus sign.
Let $W^s_{i,k,m,n,j,l}$ denote the endpoints of the subsegment of a_l that is contained in $x^s_{i,k,m,n,j}$.
Notice that $\|W^s_{i,k,m,n,j,l}\| \in \{0, 2\}$. If $\|W^s_{i,k,m,n,j,l}\| = 2$ Then let w_1, w_2 be the elements. Then let v denote the line segment defined by w_1 and w_2. We now have that $v \in Y^s_{i,k,m,n+1}$.
Let $v^{s,1}_{i,k,m,n,j}$ and $v^{s,2}_{i,k,m,n,j}$ denote the 2 lines that are perpendicular to a_i and go through the points $y^s_{i,k,m,n,j}[p]$ and $y^s_{i,k,m,n,j}[q]$ respectively. Including a_i and $y^s_{i,k,m,n,j}$ this line will intersect n line segments of A. More specifically, it will intersect 1 line segment $s \in Y^s_{i,k,m,o}$ for $0 \leq o \leq n$.
Let $u^{s,r}_{i,k,m,n,j}$ be the intersection points of a_i and $v^{s,r}_{i,k,m,n,j}$.
Let $t^s_{i,k,m,n,j}$ be the line segment defined by $u^{s,1}_{i,k,m,n,j}$ and $u^{s,2}_{i,k,m,n,j}$.
subsequent variable names will start from A again, with a hat.
The local goal at the moment is to construct a DCEL.
Let $\hat{A}$ denote the line segments $t^1_{i,k,m,n,j}$ and $t^2_{i,k,m,n,j}$.
Let $\hat{a}_{i,k,j,m,n} \in \hat{A}$ have a weight $\hat{a}_{i,k,j,m,n}[w] = k + n + 1$.
Let $\hat{B}$ denote the set containing all the elements of $\hat{A}$, only dropping the indices k, m, n, j, and re-using the index j to identify its own elements. The important part is that we now have for every line segment a_i, a set of weighted line segments $\hat{B}_i$, where all elements of $\hat{B}_i$ are contained in a_i.
Let $\hat{C}$ denote a set of new line segments, being the line segments you get when slicing a_i on all the endpoints of the segments in $\hat{B}_i$. The elements of $\hat{C}_i$ thus do not overlap. The weight of a line segment $\hat{c}_{i,j} \in \hat{C}_i$, is defined as $\hat{c}_{i,j}[w] = \max_{\hat{b}_{i,k} \text{ overlapping } \hat{c}_{i,j}} \hat{b}_{i,k}[w]$.
Let $\hat{D}_{i,j}$ denote set of the 2 lines perpendicular to a_i through the 2 endpoints of $\hat{c}_{i,j}$. Each $\hat{d}_{i,j,k} \in \hat{D}_{i,j}$ gets the weight $\hat{d}_{i,j,k}[w] = \hat{c}_{i,j}[w]$.
Let $\hat{E}^1_{i,j}$ and $\hat{E}^2_{i,j}$ denote the 2 rays one gets when splitting the elements of $\hat{D}_{i,j}$ along a_i. These rays get the weights of their line counterparts.
Let $\hat{F}^s_{i,j}$ be set of line segments $\hat{f}^s_{i,j,k,l} = a_l$ such that it is the k^{th} line segment intersected by both rays in $\hat{E}^s_{i,j}$. Furthermore, $\hat{f}^s_{i,j,0,l} = a_i$, and $\hat{f}^s_{i,j,k,l}$ only exists if both $\hat{f}^s_{i,j,k-1,l}$ exists, and the distance from $\hat{f}^s_{i,j,k-1,l}$ to $\hat{f}_{i,j,k,l}$ is smaller than ϵ along both rays.
Let $\hat{G}^s_{i,j}$ be the set of existing $\hat{f}^s_{i,j,k,l}$, so we can identify the elements of $\hat{G}^s_{i,j}$ as $\hat{g}^s_{i,j,l}$. (We disregard k, as it was merely a counter.)
Let $\hat{G}_{i,j} = \hat{G}^1_{i,j} \cup \hat{G}^2_{i,j}$.
Let $\hat{J}^s_{i,j,n,m}$ be the linesegments along the rays of $\hat{E}^s_{i,j}$, such that a_m and a_n are 2 consecutive elements in $\hat{f}^s$. We give the these linesegments weights equal to $\hat{d}_{i,j}$.
Let $\hat{j}_1$ and $\hat{j}_2$ temporarily denote the 2 elements of $\hat{J}^s_{i,j,m,n}$. We now set the values for $\hat{j}_1[z] = \hat{j}_2$ and $\hat{j}_2[z] = \hat{j}_1$.
Let $\hat{K}$ denote the set of linesegments in all of $\hat{J}^s_{i,j,m,n}$.
Let $\hat{k}_1, \hat{k}_2 \in \hat{K}$ be 2 linesegments with both segments having the same endpoints. We claim that for any pair of endpoints, at most 2 such elements exist. Furthermore, we claim that their values in $[z]$ lie on opposite sides of the segment. Let $\hat{m}$ be an edge, or combination of 2 half edges rather, such that the weight of the appropriate $\hat{k}$ is on the correct half-edge.
For all elements $\hat{k} \in \hat{K}$ that do not fit the above statement, we introduce the edge $\hat{m}$ with on 1 side the weight of k, and 0 on the other side.
Let for all $\hat{m}$ us have the set of segments $\hat{m^1}[Y], \hat{m^2}[Y] \subset A$ be the set of original segments, that together made that half-edge get its weight. That is, $\hat{m^s}[w] = \|\hat{m^s}[Y]\|$.
Let us from this point on disregard the weight of the half edges, as it is inferred directly from the set of segments that have lead to its definition.
Next we find the intersections between the elements of $\hat{M}$, and split the edges where they intersect

other edges. Each part that an edge is split in maintains the values for the segment sets that define it. The set of non-intersecting half edges this induces is called $\hat{N}$. Notice that $\hat{N}$ is planar. Now we need to format the original segments of A into half edges with associated segment sets, and we will have a complete set of edges to be the definition of our DCEL.
Let the set $\hat{O}_i$ denote the edges one gets by intersecting the edges of $\hat{M}$ with a_i. The edge gets the set of segments that are on the $\hat{m}$ on one side, and the empty set on the other side.
We now split every a_i at the points endpoints of $\hat{O}_i$, and assign the union of the segments sets of $\hat{O}_i$ as segment set to the edges this generates. We call this set of planar edges $\hat{P}$.
Let $\hat{Q} = \hat{N} \cup \hat{P}$.
Let $\hat{R} = (\hat{S}, \hat{Q}, \hat{T})$ be the DCEL that is induced by the edge set $\hat{Q}$.
Let for every face $\hat{t} \in \hat{T}$ the segment set be defined as the union of the segment sets of the half-edges that surround the face.
We now have a set of polygons/faces $\hat{T}$, each with a set of segments assigned to it, that define the weight of the polygon. All that is left to do is to assign a color (gradient) to the polygons, such that it is a nice mesh together, and the color is informative in terms of density of line segments in that area.
Let the set $\hat{u}_{i,j}$ denote the shortest distance between $\hat{t}_i$ and a_j, if $a_j \in \hat{t}_i[Y]$, where $\hat{t}_i[Y]$ denotes the segment set of $\hat{t}_i$.
Let $\hat{t}_i[w] = \|\hat{t}_i[Y]\|$.
Let $\hat{v}_i = \sum_j \hat{u}_{i,j}$.
Let $\hat{t}_i[c] = \frac{2*\hat{v}_i}{\epsilon*\hat{t}_i[w]*(\hat{t}_i[w]+1)}$.
We now have $\hat{t}_i$, which we claim to be between 0 and 1. This value represents the color of the polygon. All there is to it now is to draw the polygons, and put a color scale on the color value, and we're done.

D.4 Running time analysis

As the algorithm is described above, doing pairwise checks every time, not using efficient data structures, and making a naive implementation, the running time is unfortunately $O(n!)$. We can see this by observing the following example of n input segments, where $s_i = ((-i, i), (i, i))$ for $i = 1...n$, with $\epsilon = n + 1$. This would cause the first segment's box to intersect the other $n - 1$ segments. Each of the $n - 1$ segments above s_1 would then cast a box, and the box of s_i would find $n - i$ intersections. Continue this recursion and one finds $O(n * n!)$ as total running time, if one also counts that this is done for every segment.

D.5 Future improvement

The current description realises a very bad running time. However, this running time is only realised when the data and the chosen ϵ generate the right conditions, which are a lot of segments contained in an ϵ box.

We believe that with more work, such as proper preprocessing, simplification, clever sorting of the elements, and timely pruning, there may be a polynomial time algorithm that in the end realises the same results.
A very big improvement would likely be to use a sweepline- like algorithm, that instead of finding all segments in the half-ϵ box, casts a shadow, only keeping the first found segment for any point on the casting segment. (In the current algorithm, for any point p on a segment, there can be $O(n)$ segments in the half-ϵ that intersect the ray perpendicular to s through x. The shadow casting method would reduce that to 1). In the case described in the running time analysis, this would reduce the running time to $O(n^3 \log n)$ (For that part of the algorithm), however, more research would need to be done.