6-1-2007

# Closest and Farthest-Line Voronoi Diagrams in the Plane

Mark C. Henle
*Dartmouth College*

# Closest and Farthest-Line Voronoi Diagrams in the Plane

## Senior Honors Thesis

Mark C. Henle *
Advisor: Robert L. Drysdale

June 1, 2007

Dartmouth College Technical Report TR2007-593

**Abstract**

Voronoi diagrams are a geometric structure containing proximity information useful in efficiently answering a number of common geometric problems associated with a set of points in the plane.. They have applications in fields ranging from crystallography to biology. Diagrams of sites other than points and with different distance metrics have been studied. This paper examines the Voronoi diagram of a set of lines, which has escaped study in the computational geometry literature. The combinatorial and topological properties of the closest and farthest Voronoi diagrams are analyzed and $O(n^2)$ and $O(n \log n)$ algorithms are presented for their computation respectively.

*Undergrad in Department of Computer Science, Dartmouth College, Mark.C.Henle@dartmouth.org

# Contents

# 1 Introduction

The Voronoi diagram is a compact geometric structure which can be efficiently computed, and is useful for answering a number of questions related to proximity. When the sites of a Voronoi diagram are points, the diagram contains all the information necessary to quickly solve the all closest points problem, the Euclidean minimum spanning tree, the convex hull, the largest empty circle and the smallest enclosing circle problem [9].

In this paper we study the properties of the, previously unexamined, closest and farthest-line Voronoi diagram and give asymptotically optimal algorithms for computing each. While information like nearest neighbors doesn't make sense for lines, which in general position all intersect each other, the closest Voronoi diagram does contain all the information necessary for find a largest empty circle in $O(n^2)$ additional time. Additionally, the farthest Voronoi diagram contains all the information necessary to compute the smallest circle which touches every line, in $O(n)$ additional time.

## 1.1 Related Work

While the closest and farthest Voronoi diagrams of a set of lines in the plane has escaped study, two related Voronoi diagrams have receive some attention. There has been some work done in analyzing the complexity of the closest-line Voronoi diagram in three dimensions [10]. This work has found an $\Omega(n^2)$ lower bound and an $O(n^{3+\epsilon})$ upper bound on the complexity of this problem. In two dimensions, however, the problem becomes significantly more tractable and we can give not only asymptotically tight complexities for the closest and farthest Voronoi diagrams but also give relatively straightforward algorithms for their computation.

The farthest-segment Voronoi diagram has also recently received attention from Aurenhammer, Drysdale, and Krasser [2]. In their paper, they give an $O(n \log n)$ for computing the farthest segment Voronoi diagram under the Euclidean distance function. The closest and farthest Voronoi diagrams for every pair of input points has also received attention in the literature by Barequet, Dickerson and Drysdale [4]. The consider a variety of distance functions such as sum of distances and area of the triangle defined by the point under consideration and the two points defining the segment. They ultimately reduce the problem of computing the distance to the line defined by two points, to the problem of computing area where every pair of points is equidistant. Modifying this algorithm so that it operated only on specified pairs of points, rather than all pairs, offers a way of computing closest and farthest-line Voronoi diagrams in an asymptotically optimal way, via reduction to computing the lower and upper envelopes in 3D. While asymptotically optimal, it is likely that an algorithm implemented this way would have unattractive constants, and so it makes sense to compute the diagram in a more direct way.

# 2 Properties of Closest-Line Voronoi Diagrams

**Theorem 2.1** *The complexity of the closest-line Voronoi diagram, $CVD(S)$, where $S$ is a set of lines in general position with $n = |S|$, is $O(n^2)$.*

**Proof.** We begin by taking the arrangement of $S$, $\mathcal{A}(S)$. It is well known that the number of vertices, edges, and faces of $\mathcal{A}(S)$ are $O(n^2)$. Thus on average each face in $\mathcal{A}(S)$ has a constant number of edges and vertices. We now notice that since each face is the intersection of a series of half-planes, it is a convex polygon. Finding which edge is closest to each point in the face is exactly the medial axis problem of a convex polygon which has complexity $O(m)$ where $m$ is the number of edges [1]. Thus, the complexity of all medial axes together is $O(k)$ where $k$ is the number of edges in the entire diagram. We showed in Theorem 2.1 that the number of edges is $O(n^2)$. The overall complexity of the diagram is then $O(n^2)$. □

**Theorem 2.2** *The number of regions associated with a line $\ell \in S$ for a set of lines $S$, in general position, where $n = |S|$, is exactly $n$.*

**Proof.** Each intersection of $\ell$ with some $s \in S \setminus \ell$ produces a new region since a region of $\ell$ can't cross any other line by definition. Because our lines are in general position we have $n - 1$ intersections and thus $n$ regions. □

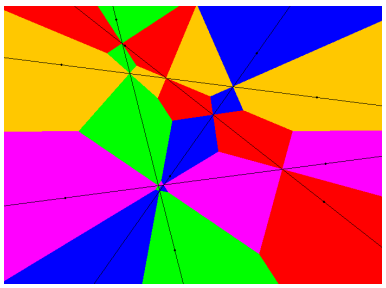

Figure 1: An example of a closest-line Voronoi diagram of 5 lines

## 3 An $O(n^2)$ Algorithm for Computing Closest-Line Voronoi Diagrams

The computation of the closest line Voronoi Diagram is a fairly straightforward and closely follows the proof of Theorem 2.1. We begin with a set of lines $S$ in general position, where $n = |S|$. We compute their arrangement $\mathcal{A}(S)$ and then compute the medial axis of each face in time proportional the number of edges on the face. Thus the algorithm runs in $O(k)$ where $k$ is the number of edges. We showed in Theorem 2.1 that the number of edges is $O(n^2)$. Thus it takes $O(n^2)$ time to compute the medial axis of all faces.

We now have $O(n^2)$ disjoint doubly-connected edge lists for our medial axes, which we need to link up. We need only destroy the boundaries of the medial axes by changing pointers in each DCEL. Changing the pointers takes constant time on average and must be done once for each DCEL. Since we have $O(n^2)$ disjoint DCELs putting our medial axes together takes $O(n^2)$ time.

Computing our medial axes takes $O(n^2)$ time and linking them together in a single diagram also takes $O(n^2)$ times, so our algorithm is $O(n^2)$ on the whole.

Given that Theorem 2.1 showed the complexity of the diagram to be $O(n^2)$ it must be the case that our $O(n^2)$ algorithm for computing it is optimal.

**Algorithm 1**: An algorithm for computing the closest-line Voronoi diagram

**Input**: $A$ = Set $S$ of $n$ lines
**Output**: An edge in the DCEL of the closest Voronoi diagram of the input lines
$\mathcal{A}(S) \leftarrow computeArrangement(S)$;
$B \leftarrow$ Empty set of medial axes of faces;
$C \leftarrow$ Empty hash table (backed by an array) from an undirected edge to two directed edges;
**foreach** *Face* $f \in \mathcal{A}(S)$ **do**
    /* $medialAxis$ produces a DCEL including the edges of $f$ */;
    $m \leftarrow medialAxis(f)$;
    **foreach** *UndirectedEdge* $e \in f$ **do**
        /* $m(e)$ is $m$'s directed copy of $e$ */;
        $C.put(e, m(e))$;
    **end**
**end**
**foreach** *UndirectedEdge* $e \in C.keys$ **do**
    DirectedEdge $d1, d2 \leftarrow C.get(e)$;
    $d1.prev.next \leftarrow d2.next$;
    $d2.prev.next \leftarrow d1.next$;
    $d1.next.prev \leftarrow d2.prev$;
    $d2.next.prev \leftarrow d1.prev$;
**end**
return $C.keys[0]$;

# 4 Properties of Farthest Line Voronoi Diagrams

**Theorem 4.1** *All regions in the farthest Voronoi diagram, $FVD(S)$, where $S$ is a set of lines, are unbounded.*

**Proof.** Let $x$ be a point in one of the regions in $FVD(S)$ belonging to some $\ell \in S$. This means that $\forall s \in S \setminus \ell \ : \ d(\ell, x) > d(s, x)$. We can then draw a disk $D(x)$, centered at $x$ and tangent to $\ell$ as shown in Figure 2. Because $\ell$ is the farthest line from $x$, $D(x)$ must intersect every $s \in S \setminus \ell$. We can also extend a ray $R$ from $p$, the point of tangency of $D(x)$ on $\ell$, and passing through $x$. It should be noted that the the line segment $\overline{px}$ is orthogonal to $\ell$ because it represents a shortest distance from a point to a line, and thus $R$ is also orthogonal to $\ell$. Let $y$ be some point along $R$, which is further from $p$ than is $x$. Because $y$ is further from $p$ we have $D(x) \subset D(y)$, where $D(y)$ is centered at $y$ and tangent to $\ell$. Thus, $D(y)$ must also intersect all $s \in S \setminus \ell$. This in turn implies that $\ell$ is the furthest line in $S$ from $y$. Since $y$ was any point along an infinite ray $R$, it must be the case that every region in $FVD(S)$ is unbounded. □

**Theorem 4.2** *The farthest Voronoi diagram of $n$ lines in general position (i.e. no parallel lines and $n \geq 2$) consists of $2n$ regions.*

**Proof.** Let $S$ be the set of lines and $FVD(S)$ be their farthest Voronoi diagram (a set of unbounded polygons). Also, let $|FVD(S)|$ denote the number of regions in $FVD(S)$.

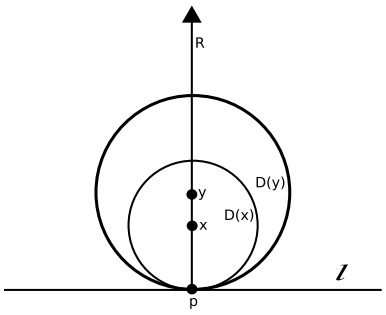


Figure 2: All points $y$ must also belong to $\ell$

We begin by showing that $|FVD(S)| \geq 2n$. Again let $\ell \in S$. Again draw a ray $R$ orthogonal to $\ell$. Now let $x$ be a point along this ray and $D(x)$ be a disk centered at $x$ and tangent to $\ell$. As we move $x$ further and further from $R$, $D(x)$ grows larger and larger. Since, no two lines are parallel, every $s \in S \setminus \ell$ must intersect $\ell$. As we move $x$ further from $\ell$ it becomes a good approximation of $\ell$ for a larger section of $\ell$ as seen in Figure 3. Thus, as $d(x, \ell) \to \infty$, $D(x)$ intersects every line $s \in S \setminus \ell$, and thus $\ell$ eventually becomes the furthest line from $x$. From Theorem 1, this $x$ must represent an entire unbounded region. We can do this for rays orthogonal in each direction and so have at least one region on either side of $\ell$. Further, these regions can't be connected because any point $y$ on $\ell$ has $d(y, \ell) = 0$ so another line in $S$ will own $y$ (we specified that $n \geq 2$). Thus, $|FVD(S)| \geq 2n$.

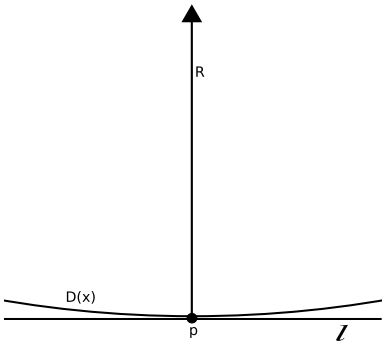


Figure 3: Every line intersecting $\ell$ also intersects $D(x)$ as it grows sufficiently large

Now we show that $|FVD(S)| \leq 2n$. We prove this by contradiction. Suppose $|FVD(S)| \geq 2n$. A line's region may not pass through the line because of our discussion in Theorem 2.1. Thus, in order for $|FVD(S)| > 2n$, it must be the case that some line $\ell \in S$ owns two distinct regions $r_1, r_2$, both on one side of it. Now let $z_1, z_2$ be points in $r_1, r_2$ respectively, and let $d(z_1, \ell) = d(z_2, \ell)$. These points must exist since by our earlier discussion each region contains an infinite portion of a ray emanating from $\ell$ and orthogonal to $\ell$. Thus, if we walk an equal distance along these rays until we've reached points along both that are in $r_1, r_2$ respectively, we have found $z_1, z_2$.

Once again, let $D(z_1), D(z_2)$ be disks centered at $z_1, z_2$ respectively and tangent to $\ell$. Since $d(z_1, \ell) = d(z_2, \ell)$, $D(z_1)$ and $D(z_2)$ have equal radii. Now let $z_3 \in \overline{z_1 z_2}$. Since $z_1, z_2$ are equidistant from $\ell$, $\overline{z_1 z_2}$ is parallel to $\ell$, and thus $d(z_3, \ell) = d(z_1, \ell) = d(z_2, \ell)$. Since $z_1, z_2$ are farther from $\ell$ than any line $s \in S \setminus \ell$, it must be the case that every $s$ intersects $D(z_1)$ and $D(z_2)$.

Without loss of generality, take $\ell$ to represent $y = 0$ in our coordinate system. Again without loss of generality, let $z_{1x} < z_{2x}$ where $z_{1x}, z_{2x}$ are the x-coordinates of $z_1, z_2$ respectively. Let $s_1$ be the leftmost intersection of $s$ and $D(z_1)$ and $s_2$ be the rightmost intersection of $s$ and $D(z_2)$. Then, $0 < s_{1y}, s_{2y} < 2 * d(z_1, \ell)$ where $s_{1y}, s_{2y}$ are the y-coordinates of $s_1, s_2$ respectively. Let $D(z_3)$ be the disk centered at $z_3$ and tangent to $\ell$. $s_1$ must be left of $z_3$ and $s_2$ must be right of it. Since the segment between the points $(z_{3x}, 0)$ and $(z_{3x}, 2 * d(z_1, \ell))$, is contained within $D(z_3)$ and $\overline{s_1 s_2}$ passes through it $\overline{s_1 s_2}$ must also pass through $D(z_3)$. Thus, $s$ must pass through $D(z_3)$.

Since $s$ represents every line other than $\ell$, $D(z_3)$ is only not intersected by $\ell$, and thus belongs to $\ell$ in $FVD(S)$. Since $z_3$ represents every point point between $z_1$ and $z_2$, then $r_1$ and $r_2$ are not two disjoint regions - a contradiction. Thus, it must be the case that $|FVD(S)| \leq 2n$.

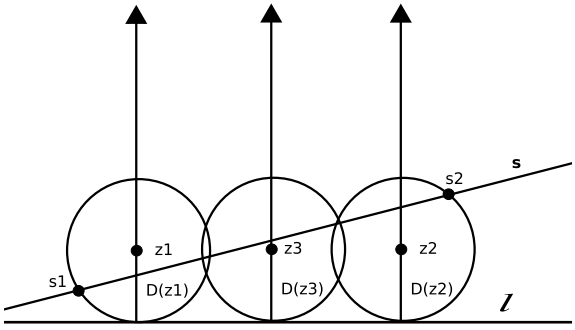


Figure 4: Any point $z3$ lying between $z1$ and $z2$ must also be farthest from $\ell$.

Since both $|FVD(S)| \geq 2n$ and $|FVD(S)| \leq 2n$, it must be the case that $|FVD(S)| = 2n$. □

**Theorem 4.3** *The overall complexity of the farthest-line Voronoi Diagram of a set of $n$ lines in general position is $O(n)$.*

**Proof.** Given Theorem 4.2 we know that the number of faces in the farthest-lines Voronoi diagram is $2n$. Because our farthest-line Voronoi diagram is a planar graph, its set of edges ($E$) and vertices ($V$) must also be linear in size. In a planar graph $|E| \leq 3|V| - 6$. Substituting into Euler's formula ($|V| - |E| + |F| = 2$), we get:

$$|V| - 3|V| - 6 + |F| \geq 2$$

which reduces to:

$$|F| > 2|V| + 8$$

This inequality implies that $|V|$ is $O(n)$. Rearranging Euler's formula we get $|E| = |V| + |F| - 2$, which implies that $|E|$ is also $O(n)$ since both $|V|$ and $|F|$ are $O(n)$. It is worth noting that since all our faces are unbounded, our graph is a tree, which would allow us to further lower our bounds on $|E|$ and $|V|$ although not asymptotically. Thus, the overall complexity of our Voronoi diagram will be $O(n)$. □

It is worth noting that adding in parallel lines doesn't change the asymptotic complexity of our diagram. This is because the number of faces for each possible orientation is two, so adding lines to parallel to lines already in $S$ doesn't increase the number of faces. For the case of two parallel lines, each line has one associated face. For more than two lines sharing an orientation, the middle lines are always closer to any point than are the extreme (by y-intercept) lines. Thus, we need only concern ourselves with the two extreme parallel lines.

**Theorem 4.4** *For a set $S$ of lines in general position, the regions associated with every line $\ell \in S$ are exactly the lower and upper envelopes of the two angle bisectors between $\ell$ and each $s \in S \setminus \ell$ where $\ell$ is taken to be x-axis.*

**Proof.** We begin by observing that the two angle bisectors represent all points which are equidistant from from $\ell$ and $s$. For any point $p$ along $a$, an angle bisector of $\ell$ and some $s \in S \setminus \ell$, we can draw orthogonal segments from $\ell$ and $s$ to $p$. Because these segments create congruent triangles, they must be equal. It then follows that any point lying to the $\ell$ side of $a$ is farther from $s$ and vice versa. Because the two angle bisectors determine the regions associated with $\ell$ and $s$, and algebraically they must lie orthogonal to one another, they form 4 regions, each representing a $\frac{\pi}{2}$ slice of the plane.

The angle bisectors of $\ell$ and $s$ produce a $\frac{\pi}{2}$ slice above and below $\ell$, which represent the regions belonging to $\ell$. Thus, the intersection of the $n-1$ regions above $\ell$ and the intersection of the $n-1$ regions below $\ell$ represent the regions that are farther from $\ell$ than any other line. By definition these are the regions belonging to $\ell$.

Taking these intersections of $\frac{\pi}{2}$ slices of the plane is equivalent to taking the lower and upper envelopes, so we're done. □

**Theorem 4.5** *Every region in $FVD(S)$ is a convex unbounded polygon.*

**Proof.** This follows directly from Theorem 4.4, and the definition of lower and upper envelopes of an arrangement of lines. □

# 5 An $O(n \log n)$ Divide-and-Conquer Algorithm for Computing Farthest-Line Voronoi Diagrams

This section presents an $O(n \log n)$ algorithm for computing the farthest Voronoi diagram of a set of $n$ lines. The algorithm's general approach is to first sort the entire list of lines based on their angle and then apply a divide-and-conquer algorithm to the sorted list.

**Input** The algorithm takes as input a set $S$ of lines where $n = |S|$.

**Output** The algorithm outputs 2 hash tables. One where lines are the keys, and their two (in general) associated faces, the values. The other where the faces are the keys and the lines, the values. The faces are sections of the underlying doubly-connected edge list which completely partitions the bounding box. We use hash tables for simplicity of notation but in practice these hash tables could be arrays indexed by line and face numbers and thus have O(1) performance for assignment and retrieval operations.

**A Note on Notation:** Throughout the rest of this paper we represent a line as the x and y-coordinates of a point intersected by the line, and the line's angle. For some $s \in S$ we refer to these as $s.x$, $s.y$ and $s.\theta$ respectively.

### 5.1 Optimality of $O(n \log n)$

A lower bound of $\Omega(n \log n)$ can be established by reduction from the problem of sorting $n$ real numbers, which is know to be $\Omega(n \log n)$:

The algorithm's basic approach is to normalize the numbers to between $0$ and $\frac{\pi}{2}$. It then computes the FVD of the lines created by using the normalized number as an angle and $(0, 0)$ as its intersection point. The resulting FVD looks like a wheel around the point $(0, 0)$, where the slices correspond to the lines in order. Given that each line own an infinite portion of the orthogonal ray emanating from it, we would expect this correspondence of orderings to hold. By walking from face to face counter-clockwise we can then print each number in order.

**Algorithm 2**: Reduction from sorting to computing a FVD

**Input**: $A$ = List of $n$ real numbers, none of which are equal
**Output**: List of $n$ real numbers sorted in non-descending order
$B \leftarrow$ Empty set of lines;
**foreach** *Real* $a \in A$ **do**
  $angle \leftarrow \frac{\pi}{2} * \frac{a - min(A)}{max(A) - min(A)}$;
  $b.x, b.y, b.\theta, b.num \leftarrow 0, 0, angle, a$;
  $B.add(b)$;
**end**
$lineToFaces \leftarrow FVD(B).lineToFacesHashTable$;
$faceToLine \leftarrow FVD(B).faceToLineHashTable$;
$start \leftarrow (b \in B : b.\theta = 0)$;
$curEdge \leftarrow lineToFaces.get(start)[0].edge$;
**repeat**
  **while** *curEdge.next.origin* $\neq$ *(0,0)* **do**
    $curEdge \leftarrow curEdge.next$;
  **end**
  Print $faceToLine.get(curEdge.face).num$;
  $curEdge \leftarrow curEdge.twin$;
**until** *faceToLine.get(curEdge.face)* = *start* ;

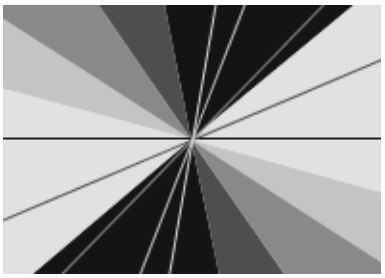

Figure 5: We would start from the black region (whose associated line has $\theta = 0$) and walk counter-clockwise the the increasing lighter regions, printing out the numbers associated with each of them.

Thus, we can reduce the problem of sorting $n$ real numbers to the problem of computing the farthest Voronoi diagram of $n$ lines. Since we know sorting to be an $\Omega(n \log n)$ problem, computing $FVD(S)$, the farthest Voronoi diagram of a set $S$ of $n$ lines, must also be an $\Omega(n \log n)$ problem.

### 5.2 Computing the bounding box

Before we can begin computing the farthest Voronoi diagram of the lines in general position contained in the set $S$, we need a sufficiently large bounding box such that no point in the Voronoi diagram lies outside the bounding box.

We begin by taking some $\ell \in T$, where $T$ is a list of our lines sorted by slope, and set it to be the x-axis without loss of generality. The infinite pieces of the lower envelope lie along the angle bisectors between $\ell$ and $\ell_{-1}$, and $\ell$ and $\ell_{+1}$, where $\ell_{-1}$ and $\ell_{+1}$ are the lines just before and just after $\ell$ in $T$ respectively. Take $a$ to be the angle bisector of $\ell$ and $\ell_{+1}$ where $\frac{\pi}{2} \leq a.\theta < \pi$. The infinite piece of the lower envelope of $\ell$ which extends infinitely down and right must lie along $a$. Any line $s \in S \setminus \{\ell, \ell_{+1}\}$, will form an angle bisector with $\ell$ that has a greater(shallower) angle than $a$. Thus, either the intersection of $s$ and $\ell$ lies to the left of the intersection of $\ell_{+1}$ and $\ell$ and $a$ will intersect the bisector between $\ell$ and $s$ and eventually become the lowest line and thus on the lower envelope. Else, $s$ and $\ell$ intersect farther right than do $\ell$ and $\ell{+}1$ and so the bisector between $s$ and $\ell$ is never below $a$ and so doesn't appear in the lower envelope.

A similar argument can be made for why the infinite piece extending downward on the left of the lower envelope lies along the bisector between $\ell$ and $\ell_{-1}$. Similar arguments hold for the upper envelope. Using the four points where each of the infinite pieces end on the lower and upper envelopes, gives the four most extreme points and thus suffice for our bounding box.

It's however unclear which bisectors first intersect the bisectors between the angle-adjacent lines described above. To avoid this issue altogether, we first calculate the bounding box of $\mathcal{A}(T)$. This problem is well understood [5]. We can then then take the length of its diagonal $|d|$ as an upper bound on the farthest distance between any pair of points.

As we move right along the lower envelope we continue to encounter points which are lower and farther right. While we don't know which angle bisector intersects $a$, we can translate $\ell_{+2}$ left until the intersection point of $\ell_{+2}$ and $\ell$ lies $|d|$ units to the left of the intersection of $\ell_{+1}$ and $\ell$. Then we claim that the angle

bisector $b$ of $\ell_{+2}$ and $\ell$ with $\frac{\pi}{2} \leq b.\theta < \pi$ intersects $a$, at least as low and at least as far right as any other bisector $c : \frac{\pi}{2} \leq c.\theta < \pi$ between $\ell$ and $s \in S \setminus \{\ell, \ell_{+1}\}$. Because $b.\theta \leq c.\theta$ and we shifted $b$ at least as far left as $c$, it intersects $a$ at least as low as $c$. Since $\frac{\pi}{2} \leq a.\theta < \pi$, this lowest intersection also implies a rightmost intersection.

Similar arguments can be made for the leftmost point on the lower envelope and the right and left points on the upper envelope. These four first intersections with the infinite portions of the lower and upper bounds form a convex hull of all the points on the lower and upper envelopes. Using them we can form a bounding box for all Voronoi points generated by $\ell$. Taking extreme x and y coordinates over bounding boxes constructed in this way for all $s \in S$ results in a bounding box for the entire diagram.

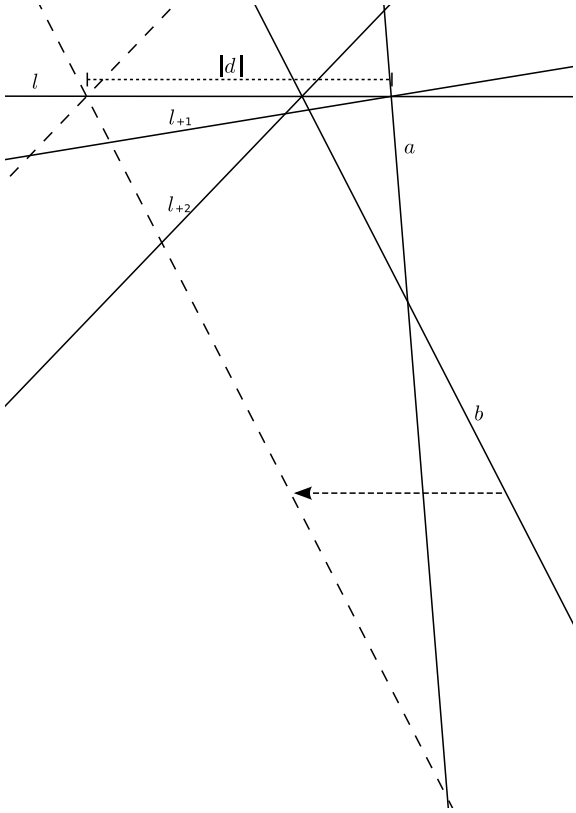


Figure 6: The intersection of the translated b and a gives a lower-right bound on the right half of the lower envelope

Let $m_1, b_1, m_2, b_2, m_3, b_3$ be the slope and intercepts of our three lines $\ell_{-1}, \ell, \ell{+}1$ respectively. Then we get the equation:

$$|d| = \sqrt{\left(\frac{b_2 - b_1}{m_1 - m_2} - \frac{b_3 - b_1}{m_1 - m_3}\right)^2 + \left(\frac{m1\,(b_2 - b_1)}{m_1 - m_2} - \frac{m1\,(b_3 - b_1)}{m_1 - m_3}\right)^2} \tag{1}$$

We then solve this for the two solutions to $b3$ presented in Appendix A and compute the intersections between the bisector of $\ell$ and $\ell_{+1}$ and the bisector between $\ell$ and $s_{+2}$, where $s_{+2}$ is $\ell_{+2}$ shifted so that its y-intercept is one of our solutions to $b3$. By using both values of $b3$ we can compute not only the bottom-right point of the lower-envelope but also the upper-left point of the upper envelope. This allows us to make a single pass through the lines since the lower-left point of the lower envelope and the upper-right point of the upper envelope, which we ignored, are the bottom-right and upper-left points on the previous three lines.

**Implementation Issues** When we implement this allowing for parallel lines or for a set $S$ with fewer than 3 orientations, we must modify the algorithm given above slightly. For parallel lines, we modify our definition of $\ell_{-1}$ and $\ell_{+1}$ to mean the lines before and after $\ell$ which are not parallel to it. For a single orientation, we need only include a point on each line in $S$ in our bounding box. For the two-orientation case, we include the intersection of all angle bisectors created by the 2 extreme lines for each orientation.

## 5.3 Merging Two Farthest Line Voronoi Diagrams

Our approach to merging two Voronoi diagrams mirrors the approach used by Shamos and Hoey [9] in their divide-and-conquer algorithm to merge two closest-point Voronoi diagrams. We merge two Voronoi diagrams by drawing two polygonal lines through the old diagrams in the manner described in Section 5.3.1. The claim is that, once this is done, the faces in the two old diagrams no longer overlap and we can simply combine the two to form the new diagram.

### 5.3.1 Drawing the Polygonal Line

In order to draw our polygonal lines we take the first and last lines, call them $\ell_1 = T[0]$ and $\ell_2 = T[|T|-1]$ from the sorted list of lines which will appear in our merged diagram. Let $\ell_3, \ell_4$ be the angle bisectors of $\ell_1$ and $\ell_2$. Further, let $\ell_4$ be the bisector which lies in the same angle created by $\ell_1$ and $\ell_2$ as a horizontal line which passes through the intersection of $\ell_1$ and $\ell_2$. We care only about $\ell_3$. Find it's intersections with the bounding box. Choose one intersection as the place to start the first polygonal line and the other as place to start the second polygonal line.

Starting from the bounding box intersection, find the first intersection of $\ell_3$ with one of our old Voronoi edges. Suppose without loss of generality that this edge is in the old Voronoi diagram which included $\ell_1$, call it $FVD_1$. Then find the intersected edge's twin. Look up this twin's face to find it's associated line $\ell_5$. This new line must lie farther from the region past the intersected edge, than does $\ell_1$. We now let $\ell_1 = \ell_5$.

Now recalculate the bisector $\ell_3$ such that it passes through our point of intersection with that last edge. Proceed along this bisector s.t. we stay across the last edge we intersected. Proceed until the first intersection in both Voronoi diagrams is on the bounding box. We're done.

### 5.3.2 Correctness

It is important to begin by noticing that our polygonal line maintains the invariant that at any point along it, it is equally far away from $\ell_1 \in L_1$, the farthest line in the sublist which generated $FVD_1$, and $\ell_2 \in L_2$, the farthest line in the sublist which generated $FVD_2$.

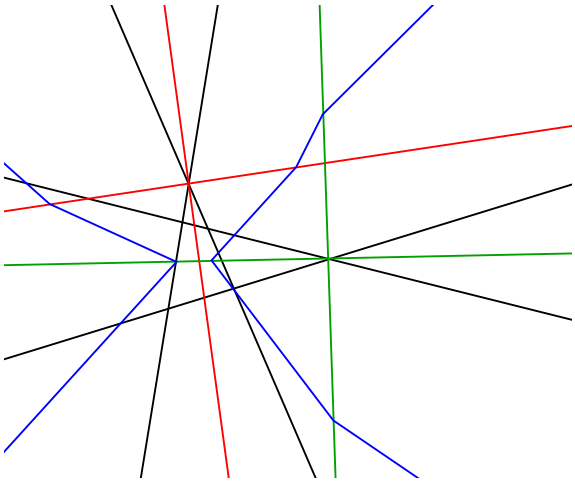

Figure 7: The blue lines represent polygonal lines in the merging of the red and green diagrams

**Lemma 5.1** *Let $T$ be the full sorted list of lines that are to appear in our new diagram, such that appending our sublists $L_1$ and $L_2$ gives $T$. At infinity, our polygonal line must begin and/or end from either side of the bisector between $L_1[0]$ and $L_2[|L_2|-1]$, which passes through the same angle as a vertical line passing through the intersection of $L_1[0]$ and $L_2[|L_2|-1]$, or it must begin and/or end from either side of the bisector between $L_1[|L_1|-1]$ and $L_2[0]$, which passes through the same angle as a horizontal line passing through the intersection of $L_1[|L_1|-1]$ and $L_2[0]$*

**Proof.** Recall our proof in Theorem 4.4 that the faces of a line $\ell$ are the upper and lower envelopes formed by the bisectors between it and every other line, where we take $\ell$ to be horizontal. The rightmost member of the lower envelope will be the bisector between $\ell$ and $\ell_{+1}$, the line with the next greatest slope, since it is steepest bisector. The same logic holds for why this bisector also the leftmost member of the lower envelope of $\ell{+}1$ when it is taken to be horizontal. Thus, as we move far enough outward, all face adjacencies are between angle-adjacent lines.

Since we already have valid Voronoi diagrams $FVD_1, FVD_2$ for $L_1, L_2$, these adjacencies must already exist between members of each sublist. However, the bisector (and resulting face adjacency at infinity) between $L_1[0]$ and $L_2[|L_2|-1]$, and the bisector between $L_1[|L_1|-1]$ and $L_2[0]$, do not yet exist. Because these bisectors did not exist in $FVD_1$ and $FVD_2$, the lower and upper envelopes of the lines at each end of the two sublists encompass the area where the bisectors will now lie. Thus, as we move outward, each pair of lines (i.e. $L_1[0]$ and $L_2[|L_2|-1]$, and $L_1[|L_1|-1]$ and $L_2[0]$) are the two farthest lines from their bisector in their respective diagrams.

Thus at infinity, the bisector represents points which are equidistant from each sublist of lines and thus divides the two old Voronoi diagrams. Further, every other angle-adjacent bisector is between lines in the same sublist and thus is not equidistant between the two sublists of lines. Thus, the four rays toward infinity created by these two bisectors, represent the only valid way to begin the polygonal line at infinity. □

**Lemma 5.2** *The two polygonal lines will never intersect so long as the two sublists don't contain lines with the same orientation and all lines don't intersect at a single point.*

**Proof.** An intersection implies that one line is moving into a region that the other line has already determined belongs solely to one of the old Voronoi diagrams. This violates the invariant that our polygonal line must always lie equidistant between the two farthest lines in each old diagram. Thus, it may never be the case that our polygonal lines intersect. □

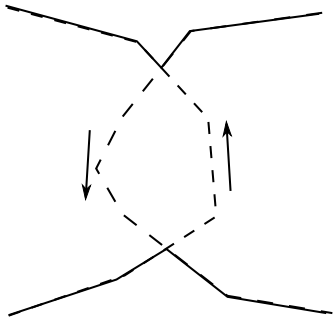

Figure 8: The fact that dashed polygonal lines intersect, implies that regions to their left intersect and so are one large region. The correct polygonal lines are the solid lines.

Lemma 5.2 implies that our two polygonal lines will never enter and leave along the same bisectors, since this would imply an intersection. Thus our choice in Section 5.3.1 to start the two polygonal lines from opposite ends of one of the bisectors guarantees that we will draw two distinct polygonal lines that do not intersect.

It stating which bisectors the polygonal line must enter and exit along we have implicitly assumed that once the polygonal line enters the old Voronoi diagrams that it will exit again and not simply enter a loop inside the old diagrams based on the rules we've set up for it. Lemma 5.3 shows that the polygonal line must exit the diagram, by which we mean it stops encountering new faces, in the old diagrams.

**Lemma 5.3** *Each polygonal line both enters and leaves each face in the old Voronoi diagrams at most once.*

**Proof.** Without loss of generality let $f_1 \in FVD_1$ be a face in the first (chosen arbitrarily) of our two old Voronoi diagrams. Suppose for the sake of contradiction that one of our polygonal lines $p_1$ intersects $f_1$ twice. If it does so in the way shown in Figure 9, then it doesn't matter whether the points to the left or right of $p_1$ are farthest from $FVD_1$ (and therefore $f_1$). In either case we end up either reducing $f_1$ to a finite region or splitting it into a two regions, one of which is finite.

This presents a contradiction, since we proved in Theorem 4.1 that a line can never be farthest from a finite region of points. Thus, it must be the case that polygonal line never intersects the same face twice.

Note that it is not possible for a polygonal line to enter and exit the same face, from and toward infinity. Suppose some polygonal line $p$ did leave and enter some face $f$ belonging to line $\ell$ from and toward infinity. Then, it must be the case that $p$ divides $f$ into two faces. If it didn't, then it would create finite regions in the

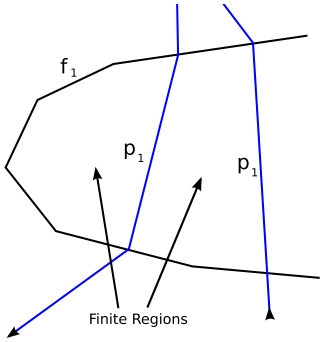


Figure 9: If a single polygonal line p1 enters and leaves a face twice, it implies a finite region. Yet we proved that all our regions are unbounded

old Voronoi diagram in which $f$ resides, which we know can't be the case by Theorem 4.1. We know, that $l$ does not intersect $f$ since $f$ represents all points farthest from $l$. Thus, we have two regions on one side of $l$. However, in Theorem 4.2 we showed that this may never be the case, a contradiction. □

Thus, while we won't make any statements about how the polygonal line moves through the diagram, we now know that if started on the bisector between $L_1[0]$ and $L - 2[|L - 2| - 1]$ at infinity, it will eventually exit along the bisector between $L_1[|L_1| - 1]$ and $L_2[0]$ toward infinity.

**Theorem 5.4** *Starting our polygonal lines on either side of the bisector between $T[0]$ and $T[|T| - 1]$ at infinity, results in complete partitioning of the old Voronoi diagrams $FVD_1$ and $FVD_2$ such that they no longer overlap.*

**Proof.** Suppose, for the sake of contradiction, that after the two polygonal lines are drawn, there are two faces $f_1 \in FVD_1$ and $f_2 \in FVD_2$ which overlap. This means that their associated lines $\ell_1 \in FVD_1$ and $\ell_2 \in FVD_2$ are farthest from the region of overlap. Thus, we may start a third polygonal line $p_3$ with the bisector between $\ell_1$ and $\ell_2$ and passing through this overlap region. Now let us continue down this bisector in the direction such that we exit $f_1$ first, entering $f_3$. We can now apply our rule of finding the bisector between $\ell_2$ and $\ell_3$, the line associated with the face $f_3$. This must be a valid part of our polygonal line since it is equidistant from the farthest line in each old diagram.

However, we know from Lemma 5.3 that a polygonal line can't touch the same face twice, and so must eventually exit along a bisector that doesn't intersect with any new faces. Yet, we know of only 4 valid entry and exit points which our two existing polygonal lines already occupy, so $p_3$ must join $p_1$ or $p_2$. Yet, this is clearly incorrect, since a polygonal line is splitting in two, meaning two different pair of lines in the two old diagrams are simultaneously farthest from a region. This has no valid geometric interpretation, and so we have reached a contradiction. Thus, it must the case that the two polygonal lines completely partition the two old diagrams, such that the two no longer overlap. □

Thus, after drawing our polygonal lines, no overlap exists between our two old diagrams. Further, because of the invariant maintained by our polygonal line, we know those regions covered by $FVD_1$ are farthest from some line in it and those covered by $FVD_2$ are farthest from some line in it.

Thus, simply combining the faces of the old diagrams (as reduced by our polygonal line) produces a correct Voronoi diagram for the full set of lines $T$.

### 5.3.3 Implementation

Our polygonal line consists of portions of the different choices of $\ell_3$ we make as we moved through the diagram. In order to implement this in our DCEL representation, we build queues of new edges for each of the two faces which we are currently passing through. When our polygonal line leaves a face, we connect the beginning and end of our queue to the points where we entered and left. Exactly how we connect our next and previous pointers depends on which face we are leaving. Because edges in a DCEL are always arranged counter-clockwise, we need to connect the edges in our queue in different orders depending on which face we are dealing with.

For the left face, our edges are linked in the same order as the direction we are moving with our polygonal line (i.e. when we add a new edge, it's previous edge is the last edge currently in the queue, and the next edge of the last edge currently in the queue is the new edge). When we leave a left face make the first edge in our queue the next edge of the edge we entered the face on, and we make the edge we leave on's previous edge the last edge in our queue. For the right face, it's edges are all linked in the opposite direction as our polygonal line is proceeding through the plane.

In our implementation we use the technique outlined in [8], progressing counter-clockwise around the left face and clockwise around the right face. Without this technique, we could potentially do $O(n)$ scans for a face with $O(n)$ edges, resulting in a $O(n^2)$ merge. Using it, however, means we traverse each edge in the diagram at most once. Since there are $O(n)$ edges in the diagram, our merge is $O(n)$ overall.

## 5.4 Base cases

We define two base cases.

### 5.4.1 One line

In the single-line case, we have two faces defined the be the two regions of the plane divided by the line. The line owns the whole plane but we use two faces for consistency.

### 5.4.2 Two lines

In the two-line case, we begin by drawing both angle bisectors, which partitions the plane into 4 faces. Then each line owns the two faces which it does not intersect.

The reason we need a base case for this rather than being able to merge two one-line Voronoi diagrams, is that the behavior of the polygonal line we use in our merge method below becomes undefined when it simultaneously hits two old Voronoi edges. In theory, we arbitrarily choose one edge to hit first. But we are now lying on the second edge, so we intersect it immediately either go back the way we came or end up with a straight polygonal line through the intersection point of our two lines. Either is incorrect.

In theory, with thoughtful coding, we could fix the errors described above. With double precision, however, things get worse and we have a "race condition" as to which Voronoi edge we hit first. Thus, no set of consistent rules help since the edge we hit first is effectively random. Because of these issues, dealing two lines as a base case is the most robust approach.

## 5.5 Special Cases

### 5.5.1 A set of parallel lines

In order to correctly compute this case, we convert each line to its slope-intercept representation and then take the two lines that have the most extreme intercepts, discarding the rest. Call these lines $\ell_1$ and $\ell_2$. For the case of vertical lines we take the most extreme x-coordinates. Paring down the list in this fashion is valid because the lines in the middle will be closer to every point in the plane than one of the extreme lines.

Now let $\ell_3$ be a line with the same slope as $\ell_1, \ell_2$ and whose intercept is the average of the intercepts of $\ell_1$ and $\ell_2$. All points on the $\ell_1$ side of $\ell_3$ are farther from $\ell_2$ and vice-versa. Thus, the face on the $\ell_1$ side of $\ell_3$ belongs $\ell_2$ and the face on the $\ell_2$ side of $\ell_3$ belongs to $\ell_1$, so we're done.

### 5.5.2 A set of lines all intersecting at a single point

This case represents a generalization of our approach to two-line base case. For $n$ lines, we draw all bisectors between angle-adjacent lines. This divides the plane into $2n$ faces where every line owns the two faces which the line orthogonal to it (and passing through the common intersection point) intersects.

### 5.5.3 Mid-lines of subsets of parallel lines intersect at a single point

This case arises in its simplest case when we have 4 lines, the first two which are parallel and $x$ units apart and the second two which are parallel and $x$ units apart. Thus, the mid-lines, which are edges in the Voronoi diagrams of each pair of lines, intersect. Unfortunately, when we attempt to merge the Voronoi diagrams of each pair, our polygonal line encounters the two mid-lines simultaneously. This creates the same inconsistent state in our polygonal line that caused issues in the special case presented in Section 5.5.2. This case is illustrated in Figure 10.

We can deal with this case by collecting the mid-lines, making a recursive call on them, which will be caught by either the two-line base case or the special case in Section 5.5.2. Then we need only replace each mid-line with it's two parallel lines and assign one face to each.

This case arises in it's full generality when the mid-lines of the two most extreme parallel lines for $n$ orientations intersect at a single point.

As with the two-line case, the polygonal line we are drawing in our merge method simultaneously encounters $n$ lines and it's undefined which it should intersect first. Again, in practice, double precision again creates a "race condition" as to which edge we encounter first.

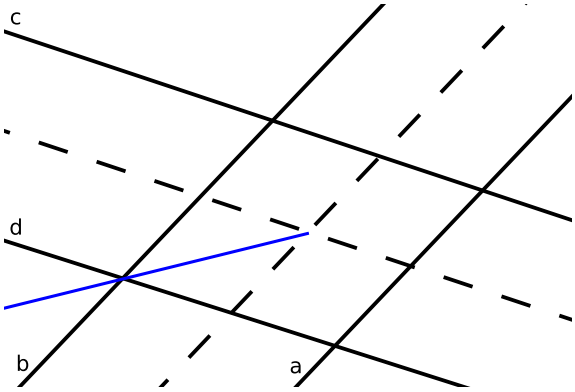


Figure 10: In a straightforward merge our polygonal line (represented in blue) encounters the mid-line of $a, b$ and the mid-line of $c, d$ simultaneously

#### 5.5.4 General Problem of a Polygonal Line Leaving Two Faces Simultaneously

The above represent easily identifiable and correctable cases of our polygonal line leaving two faces simultaneously. However, the lines may no be organized into an easily identifiable case as it is in the above three cases. In these cases, there is no easy way to improve the robustness of our algorithm with double precision as a confounding factor.

### 5.6 The Recursive Function

At the center of our divide-and-conquer is a recursive function. In the general case this function seeks to divide the set of lines given to it into two halves, call itself on these two halves, and then use the merge algorithm described in Section 5.3, to merge the two diagrams together.

However, it must also run tests on the list of lines given to it to check for the special cases described in Sections 5.5.1, 5.5.2, and 5.5.3, as well as the one and two-line base cases. In those cases it calls the appropriate method for that special case.

There were several ways we could have implemented handling parallel lines at a high level. One way would have been to simply keep dividing our set until we had a set which was composed entirely of parallel lines. While this seems like a reasonable approach, in reality we potentially end up dividing up our parallel lines among several cases. When we then go to merge them using the polygonal line method outlined in Section 5.3.1, we run into problems.

The primary one is that we are frequently drawing angle bisectors which pass through the intersection of the two lines whose angle they bisect. This intersection is obviously undefined when our two lines are parallel. Second, we would need some sort of special code to recognize when a face was completely removed, which can only happen with parallel lines.

Our approach is instead to do a simple check after splitting our list of sorted lines in two. If the angles of the last line in the first sublist and the first line in the second sublist are equal, then we enter a special high-level case. We search for the farthest left and farthest right indices of lines with this angle in the list.

Then call our recursive function on the sublist before the left index and the sublist after the right index. We also call our special parallel lines handling code on the sublist between the two indices. We then merge the FVDs of the left and middle (parallel) sublists. Then we merge this new Voronoi diagram with the Voronoi diagram produced by the right sublist. Since we never merge Voronoi diagrams of sublists, where a single orientation is divided among the two, we never run into the issues, which hamper the alternate approach detailed above.

Below is pseudocode for the algorithm described above, with the special case described in 5.5.3 ignored because of the extra complexity it introduces and its relative obscurity.

**Function** `RecursiveFVD(`$lines$`)`

**Input**: $T$ = a list of lines in sorted order
**Output**: The farthest Voronoi diagram of $T$
**if** $T[|T|/2-1].\theta = T[|T|/2].\theta$ **then**
  $i, j$ = indices of left and right-most lines parallel to $T[|T|/2]$;
  $vd2 = AllParallel(T[i, j+1])$;
  **if** $i > 0$ **then**
    $vd1 = RecursiveFVD(T[0, i])$;
    $tempvd = MergeFVD(vd1, vd2, T[0, j+1])$;
  **else**
    $tempvd = vd2$;
  **end**
  **if** $j < |T|$ **then**
    $vd3 = RecursiveFVD(T[j+1, |T|])$;
    $vd = MergeFVD(temp, vd3, T)$;
  **else**
    $vd = temp$;
  **end**
**else**
  **if** $|T| > 2$ **then**
    **if** $T$ *all intersect at a single point* **then**
      $vd = SingleIntersectionFVD(T)$;
    **else**
      $vd1 = RecursiveFVD(T[0, |T|/2])$;
      $vd2 = RecursiveFVD(T[|T|/2, |T|])$;
      $vd = MergeFVD(vd1, vd2, T)$;
    **end**
  **end**
  **if** $|T| = 2$ **then**
    $vd = baseFVD2(T)$;
  **end**
  **if** $|T| = 1$ **then**
    $vd = baseFVD1(T)$;
  **end**
**end**
return $vd$;

### 5.7 Complexity Analysis

We have argued above why our merge method runs in $O(n)$ time. Given our divide-and-conquer approach, the algorithm should run in $O(n \log n)$. The base cases are part of the normal divide-and-conquer analysis so we need only verify that our special cases don't raise the complexity of the algorithm. We handle the case where all lines intersect in a single point in $O(n)$, so this clearly remains within our bounds.

For parallel lines, we use $O(j - i)$ time where $i$ and $j$ are the left and right-most indices of the group of parallel lines we are examining. This is at most $O(n)$. We do two merges, which are each $O(n)$. Aggregating these steps, we still do only $O(n)$ work at each level. We always divide the problem into lists at most half the length of the list we were given, so the standard divide-and-conquer analysis still holds, and our algorithm is $O(n \log n)$ overall.

## 6 Implementation Issues

Below we discuss issues which are non-existent in theory but challenge our algorithm's robustness when implemented.

### 6.1 Double Precision

Because our code requires frequently finding intersection points and converting from point-angle to slope-intercept representation, dealing with doubles is inevitable even if we are careful to start with integer coordinates for our lines.

The issue of double precision appears most obviously in Sections 5.4.2 and 5.5.2, where intersection behavior differed markedly from theory.

This also becomes an issue for answering many basic questions integral to our algorithm. For example, it became an issue for computing intersections. When computing a single intersection double precision isn't much of an issue since we just need something that's "very near" the real point of intersection. When we need to start comparing intersections, as we do when determining if a set of $n$ lines all intersect at a single point, double precision becomes an issue. Similar issues present themselves when trying to determine if a point lies on a line or edge.

In order to bypass the issue raised by double precision we define a algorithm-wide constant $c$, which is some small decimal. Now, instead of asking if $n$ lines intersect a exactly the same point, we can ask whether they intersect at points whose distance is less than $c$. The same strategy can be applied to most of the other complications arising from double precision. Any even better strategy, for which reasonable heuristics would need to be devised, would be to scale this $c$ up with the size of the bounding box rather than leaving it as a constant.

One place where double precision simply has to be lived with is in the computation of our bounding box. In the Java implementation provided in the appendix to this paper, solving for $b3$ in the bounding box equation can sometimes take on the IEEE-defined floating point value "Not a Number". This is because the

solution has the term $m_1 - m_2$ (difference in slopes) in its denominator, and for values of $m_1, m_2$ which are sufficiently close, this can result in effectively dividing by zero.

There is no easy solution to this issue. We could make our algorithm robust by removing one of the lines or by treating the two as parallel. However, the resulting farthest Voronoi diagram would not be the "true" diagram of the input lines. Each implementation must make its own choice here. The implementation presented in the appendix to this paper chooses to leave things non-robust and simply allow the algorithm to throw an error, which lets the user know that he needs to reexamine his input lines and retry.

# 7 Further Work

The content of this paper suggests several areas of further work.

- This paper has implicitly assigned an equal weight to each of the lines under consideration. However, the literature for other choices of sites has explored the possibility that we would like to assign weights to the sites [3][7][4]. Under multiplicative weighting, our distance function becomes $d(\ell, p)/w(\ell)$, where $w(\ell)$ is the weight we assigned to $\ell$. As an open problem, we leave modifying our algorithm to accomodate weights. These modifications would have to take into account that I line, which is parallel to no other line, may have no faces.

  Another, more straightforward, way of calculating the multiplicatively-weighted diagram is to return to the approach we discussed in the Related Work section. Now, instead of assigning length 1 to all the segments in our reduction, we instead assign lengths equal to the weights assigned to each line, and compute the closest and farthest-segment diagrams using the area distance function as done by Barquet et.al.[4].

- This paper has focused exclusively on Euclidean distance. However, the Voronoi diagram literature often makes use of other distance metrics such as the $L_1$(Manhattan) and $L_\infty$ metrics [6]. It would be interesting to reexamine the properties of closest and farthest-line Voronoi diagrams introduced in this paper, under these alternative distance functions. Do our properties still hold? Do similar properties hold? Our algorithm makes Euclidean assumptions in a multitude of places. It would likely take significant work to adapt our algorithm if it could be adapted at all.

- Finally, if this algorithm were to be used in a software package, our implementation would need to be made robust in several of the ways discussed in the paper. Our universal constant $c$ fails to do a good job as the bounding box becomes sufficiently large. Allowing it to scale up with the diagonal length of the bounding box would help mitigate some of the error associated with double precision. Ultimately other trade-offs would need to be made as magnitudes became sufficiently large. For example, calculating the intersection point of two lines $\ell_1, \ell_2$ whose slopes are sufficiently close and whose distance to one another is sufficiently large at $x = 0$, results in a fraction whose numerator $(b_2 - b_1)$ is approaching infinity and whose denominator $(m_1 - m_2)$ is approaching 0. Eventually, this will lead to a quotient of "Not a Number". Thus, well-documented trade-offs for these cases, such as removing one of the lines or treating the two as parallel, would be necessary.

  Another approach would be to implement the algorithm on top of an exact arithmetic package such

as those provided in LENA and CGAL. This would allow us to remove our constant $c$ and the complications arising from it. It would also allow us to easily detect when we simultaneously leave two faces. With careful coding for these cases, we would likely be able to remove some of the special cases presented in Section 5.5.

# References


[1] A. Aggarwal, L.J. Guibas, J. Saxe, P.W. Shor, A linear time algorithm for computing the Voronoi diagram of a convex polygon, *Discrete & Computational Geometry 4*, 1989, pp. 345-405.

[2] Aurenhammer, F., Drysdale, R. L., and Krasser, H., Farthest line segment Voronoi diagrams, *Inf. Process. Lett. 100, 6* Dec. 2006. pp. 220-225.

[3] F. Aurenhammer and H. Edelsbrunner, An optimal algorithm for constructing the weighted Voronoi diagram in the plane, *Pattern Recognition, 17(2)* 1984. pp. 251-257.

[4] G. Barequet, M.T. Dickerson and R.L.S. Drysdale, On 2-Point site Voronoi diagrams, *Discrete Applied Mathematics, Volume 122, Issues 1-3* Oct. 15, 2002, pp. 37-54.

[5] de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O. *Computational Geometry: Second Edition* 1997, pp. 181 ex. 8.4

[6] D.T. Lee, C.K. Wong, Voronoi diagrams in the $L_1$ ($L_\infty$) metrics with 2-dimensional storage applications, *SIAM Journal on Computing 9*, 1980, pp. 200-211.

[7] D.T. Lee, V.B. Wu, Multiplicative weighted farthest neighbor Voronoi diagrams in the plane In *Proc. Int. Workshop on Discrete Mathematics and Algorithms, Hong Kong*, 1993, pp. 154-168

[8] Preparata, F. P. and M. I. Shamos *Computational Geometry: An Introduction* Spinger-Verlag, New York, 1985.

[9] Shamos, M.I, and Hoey, D. Closest-point problems, In *Proc 16th IEEE Symp. Foundatmns of Computer Science.*, Oct 1975, pp. 151-162.

[10] Weisman, A., Chew, L. P., and Kedem, K. Voronoi diagrams of moving points in the plane and of lines in space: tight bounds for simple configurations. *Inf. Process. Lett. 92, 5* , Dec. 2004

# A Appendix: Solutions for $b_3$ in Equation 1

The two solutions for $b_3$ in the equation presented in Section 5.2 are presented below.

$$\left\{ b3 = \frac{1}{2\,(1 + m1^2)\,(m1 - m2)} (2\,b2\,m1^3 + 2\,m3\,b1\,m1^2 - 2\,b1\,m2\,m1^2 - 2\,m1^2\,b2\,m3 + 2\,b2\,m1 + 2\,b1\,m3 - 2\,b1\,m2 - 2\,b2\,m3 + 2\,\mathrm{sqrt}( distMax^2\,m1^4 + m1^6\,distMax^2 + 4\,m1^4\,distMax^2\,m2\,m3 - 2\,m1^3\,distMax^2\,m2\,m3^2 - 2\,m1^3\,distMax^2\,m2^2\,m3 + m1^2\,distMax^2\,m2^2\,m3^2 + m1^4\,distMax^2\,m3^2 - 2\,m1^5\,distMax^2\,m3 - 2\,m1^5\,distMax^2\,m2 + m1^4\,distMax^2\,m2^2 + 4\,distMax^2\,m1^2\,m2\,m3 - 2\,distMax^2\,m1\,m2\,m3^2 - 2\,distMax^2\,m2^2\,m1\,m3 + distMax^2\,m1^2\,m3^2 - 2\,distMax^2\,m1^3\,m3 - 2\,distMax^2\,m1^3\,m2 + distMax^2\,m2^2\,m1^2 + distMax^2\,m2^2\,m3^2)) \right\},$$

$$\left\{ b3 = \frac{1}{2\,(1 + m1^2)\,(m1 - m2)} (2\,b2\,m1^3 + 2\,m3\,b1\,m1^2 - 2\,b1\,m2\,m1^2 - 2\,m1^2\,b2\,m3 + 2\,b2\,m1 + 2\,b1\,m3 - 2\,b1\,m2 - 2\,b2\,m3 - 2\,\mathrm{sqrt}( distMax^2\,m1^4 + m1^6\,distMax^2 + 4\,m1^4\,distMax^2\,m2\,m3 - 2\,m1^3\,distMax^2\,m2\,m3^2 - 2\,m1^3\,distMax^2\,m2^2\,m3 + m1^2\,distMax^2\,m2^2\,m3^2 + m1^4\,distMax^2\,m3^2 - 2\,m1^5\,distMax^2\,m3 - 2\,m1^5\,distMax^2\,m2 + m1^4\,distMax^2\,m2^2 + 4\,distMax^2\,m1^2\,m2\,m3 - 2\,distMax^2\,m1\,m2\,m3^2 - 2\,distMax^2\,m2^2\,m1\,m3 + distMax^2\,m1^2\,m3^2 - 2\,distMax^2\,m1^3\,m3 - 2\,distMax^2\,m1^3\,m2 + distMax^2\,m2^2\,m1^2 + distMax^2\,m2^2\,m3^2)) \right\}$$

# B Code Appendix

The following is a complete Java 1.5 implementation of the algorithm outlined in this paper.

```
import java.util.*;

/**
 * @author Mark Henle
 * Contains the main method and meat of
 * Voronoi-calculating code
 */
public class Worker {
  // Extreme coordinates along x and y axis
  // for the bounding box
  public static double right_bound;
  public static double left_bound;
  public static double top_bound;
  public static double bottom_bound;

  // Corners of the bounding box determined from the
  // extreme coordinates above
  static PointD top_right;
  static PointD top_left;
  static PointD bottom_right;
  static PointD bottom_left;

  // These variables are a bit messy.  They are set
  // in bbintersection and are then used elswhere.  Thus, they
  // are not thread-safe
  static PointD top_inter = null;
  static PointD bottom_inter= null;
  static PointD right_inter = null;
  static PointD left_inter = null;

  // This is a number we use in various places to prevent double
  // imprecision from hurting us (see fuzzyEqual, isSingleIntersection)
  public static final double FUDGE = .0001;
  // Amount to add to our calculated max distance
  // Needed for two line case where the computed distance
  // is always zero
  public static final double MAX_DIST_MARGIN = 10;
  // Amount to add in each direction to give us
  // extra room in our bounding box
  public static final double BOUNDING_ROOM = 20;

  /**
   * Main method which kicks off the Voronoi calculation
   * and then optionally displays it
   * @param args 0: file which contains a list of lines in
   * the format: x,y,angleInRadians\n
   *          1: "display" to display the diagram using our
   * rudimentary graphics package or no argument for no GUI representation
   */
  public static void main (String[] args) {
    if(args.length < 1) {
      System.err.println("You must provide a file of input lines");
      return;
    }
    String fileName = args[0];
    ArrayList<Line> lines = Parser.parse(fileName);
    Collections.sort(lines, new LineComparator());
    // Find diagonal length of bounding box of all
    // intersection points in the arrangeent
    DimensionD vdBounding = findVDBoundingBox(lines);
    setVDBounding(vdBounding);
    VoronoiDiagram vd = calculateFVD(lines);
    System.out.println(vd);
    if(args.length > 1 && args[1].equals("display"))
      DisplayVD.display(vd, "vd");
```

```
}

/**
 * Calculates the farthes Voronoi Diagram of the given lines
 * @param lines - The lines whose FVD we wish to compute
 * @return
 */
public static VoronoiDiagram calculateFVD(List<Line> lines) {
  // Duplicate lines don't make sense so delete them
  removeDups(lines);
  VoronoiDiagram vd = recursiveFVD(lines);
  return vd;
}

/**
 * Begins recursive divide-and-conquer computation
 * of the farthest Voronoi Diagram
 * @param lines - Lines whose FVD we wish to compute
 * @return
 */
public static VoronoiDiagram recursiveFVD(List<Line> lines) {
  VoronoiDiagram vd = null;
  if(lines.size() > 2) {
    // If they all intersect at a single point use a base case
    if(isSingleIntersection(lines)) {
      vd = singleIntersectionFVD(lines);
    }
    /*
     * If middle two are parallel compute FVD's of the middle parallel block
     * and the blocks on it's left and right and merge them together one
     * at a time
     */
    else if(lines.get(lines.size()/2-1).getAngle() == lines.get(lines.size()/2).getAngle()) {
      double angle = lines.get(lines.size()/2-1).getAngle();
      int i = 0; // Left end (inclusive) of block of parallel lines
      while(lines.get(i).getAngle() != angle) { i++; }
      int j = lines.size()/2; // Right end (non-inclusive) of block of parallel lines
      while(j < lines.size() && lines.get(j).getAngle() == angle) { j++; };
      // Call base case for a group of parallel lines
      VoronoiDiagram vd2 = allParallelFVD(lines.subList(i, j));
      // temp holds the intermediate merge between
      // the left and middle (parallel) blocks
      VoronoiDiagram temp;
      // Merge in non-empty parts on left and right of parallel section
      if(i > 0) {
        VoronoiDiagram vd1 = recursiveFVD(lines.subList(0, i));
        temp = mergeFVD(vd1, vd2, lines.subList(0, j));
      }
      else
        temp = vd2;
      if(j < lines.size()) {
        VoronoiDiagram vd3 = recursiveFVD(lines.subList(j, lines.size()));
        vd = mergeFVD(temp, vd3, lines);
      }
      else
        vd = temp;
    }
    // In general just recurse on each half and merge
    else {
      VoronoiDiagram vd1 = recursiveFVD(lines.subList(0,lines.size()/2));
      VoronoiDiagram vd2 = recursiveFVD(lines.subList(lines.size()/2,lines.size()));
      vd = mergeFVD(vd1, vd2, lines);
    }
  }
  // The two-line base case
  else if(lines.size() == 2) {
    if (lines.get(0).getAngle() == lines.get(1).getAngle()) {
      vd = allParallelFVD(lines);
```

```
    }
    else {
      vd = baseFVD2(lines);
    }
  }
  // The one-line base case
  else {
    vd = baseFVD1(lines);
  }
  return vd;
}

/**
 * Handles the base case of two lines
 * @param lines - two line list
 * @return
 */
public static VoronoiDiagram baseFVD2(List<Line> lines) {
  VoronoiDiagram vd = new VoronoiDiagram();
  // Make a copy when we set the list of lines so we
  // don't run into list modification errors
  List<Line> newLines = new ArrayList<Line>();
  newLines.add(lines.get(0));
  newLines.add(lines.get(1));
  vd.setLines(newLines);
  Line l1 = lines.get(0);
  Line l2 = lines.get(1);
  // l3 and l4 are the two angle bisectors of l1 and l2
  Line l3 = lineBetween(l1, l2);
  Line l4 = lineBetween(l1, l2);
  l4.setAngle(l4.getAngle()+Math.PI/2);
  PointD interPoint = intersection(l3, l4);
  Vertex v_inter = new Vertex(interPoint);
  Face f1 = new Face();
  Face f2 = new Face();
  Face f3 = new Face();
  Face f4 = new Face();

  /*
   * Calculate the 4 points where our 2 angle bisectors
   * hit the bounding box and arrange them counter-clockwise
   * along the bounding box
   */
  PointD[] bbinters = bbintersection(l3);
  PointD p1 = bbinters[0];
  PointD p2 = bbinters[1];
  bbinters = bbintersection(l4);
  PointD p3 = bbinters[0];
  PointD p4 = bbinters[1];
  PointD[] p = {p1,p2,p3,p4};
  p1 = p[0];
  p2 = findNextAlongBB(p1, p);
  p3 = findNextAlongBB(p2, p);
  p4 = findNextAlongBB(p3, p);

  Vertex v1 = new Vertex(p1);
  Vertex v2 = new Vertex(p2);
  Vertex v3 = new Vertex(p3);
  Vertex v4 = new Vertex(p4);

  /*
   * Draw the four faces created by the two lines
   * and then connect them up by setting the twin
   * relationships between their edges correctly
   */
  // Draw first face
  HalfEdge[] path1 = findBBPath(p1,p2);
```

```
f1.setEdge(path1[0]);
for(HalfEdge e : path1) {
  e.setFace(f1);
}
HalfEdge toInter1 = new HalfEdge(v2, path1[path1.length-1], null, null, f1);
v2.setIncident(toInter1);
path1[path1.length-1].setNext(toInter1);
HalfEdge fromInter1 = new HalfEdge(v_inter, toInter1, path1[0], null, f1);
v_inter.setIncident(fromInter1);
toInter1.setNext(fromInter1);
path1[0].setPrev(fromInter1);

// Draw second face
HalfEdge[] path2 = findBBPath(p2,p3);
f2.setEdge(path2[0]);
for(HalfEdge e : path2) {
  e.setFace(f2);
}
HalfEdge toInter2 = new HalfEdge(v3, path2[path2.length-1], null, null, f2);
v3.setIncident(toInter2);
path2[path2.length-1].setNext(toInter2);
HalfEdge fromInter2 = new HalfEdge(v_inter, toInter2, path2[0], null, f2);
toInter1.setTwin(fromInter2);
fromInter2.setTwin(toInter1);
toInter2.setNext(fromInter2);
path2[0].setPrev(fromInter2);

// Draw third face
HalfEdge[] path3 = findBBPath(p3,p4);
f3.setEdge(path3[0]);
for(HalfEdge e : path3) {
  e.setFace(f3);
}
HalfEdge toInter3 = new HalfEdge(v4, path3[path3.length-1], null, null, f3);
v4.setIncident(toInter3);
path3[path3.length-1].setNext(toInter3);
HalfEdge fromInter3 = new HalfEdge(v_inter, toInter3, path3[0], null, f3);
toInter2.setTwin(fromInter3);
fromInter3.setTwin(toInter2);
toInter3.setNext(fromInter3);
path3[0].setPrev(fromInter3);

// Draw fourth face
HalfEdge[] path4 = findBBPath(p4,p1);
f4.setEdge(path4[0]);
for(HalfEdge e : path4) {
  e.setFace(f4);
}
HalfEdge toInter4 = new HalfEdge(v1, path4[path4.length-1], null, null, f4);
toInter4.setTwin(fromInter1);
fromInter1.setTwin(toInter4);
v1.setIncident(toInter4);
path4[path4.length-1].setNext(toInter4);
HalfEdge fromInter4 = new HalfEdge(v_inter, toInter4, path4[0], null, f4);
toInter3.setTwin(fromInter4);
fromInter4.setTwin(toInter3);
toInter4.setNext(fromInter4);
path4[0].setPrev(fromInter4);

/*
 * Assign faces to lines:
 * Line with an intersection point whose path to
 * p1 is shortest owns f1, f3
 */
PointD[] inters1 = bbintersection(l1);
PointD[] inters2 = bbintersection(l2);
```

```
  findBBPath(inters1[0], p1);
  findBBPath(inters1[1], p1);
  double l1dist = Math.min(findBBPath(inters1[0], p1).length, findBBPath(inters1[1], p1).length);
  double l2dist = Math.min(findBBPath(inters2[0], p1).length, findBBPath(inters2[1], p1).length);
  if(l1dist > l2dist) {
    Line temp = l1;
    l1 = l2;
    l2 = temp;
  }
  if(l1dist == l2dist) {
    HalfEdge[] shortestPath1 = findBBPath(inters1[0], p1).length < findBBPath(inters1[1], p1).length ?
        findBBPath(inters1[0], p1) : findBBPath(inters1[1], p1);
    HalfEdge[] shortestPath2 = findBBPath(inters2[0], p1).length < findBBPath(inters2[1], p1).length ?
        findBBPath(inters2[0], p1) : findBBPath(inters2[1], p1);
    double firstLeg1, firstLeg2;
    if(shortestPath1.length > 1) {
      firstLeg1 = distance(shortestPath1[0].getOrig().getPoint(),
        shortestPath1[0].getNext().getOrig().getPoint());
      firstLeg2 = distance(shortestPath2[0].getOrig().getPoint(),
        shortestPath2[0].getNext().getOrig().getPoint());
    }
    else {
      firstLeg1 = distance(shortestPath1[0].getOrig().getPoint(), p1);
      firstLeg2 = distance(shortestPath2[0].getOrig().getPoint(), p1);
    }
    if(firstLeg1 > firstLeg2) {
      Line temp = l1;
      l1 = l2;
      l2 = temp;
    }
  }

  Face[] l1faces = new Face[2];
  l1faces[0] = f1;
  l1faces[1] = f3;
  Face[] l2faces = new Face[2];
  l2faces[0] = f2;
  l2faces[1] = f4;

  // Add line-face pairs to VD
  vd.add(l1, l1faces);
  vd.add(l2, l2faces);
  return vd;
}

/**
 * Create a Voronoi Diagram for lines which all
 * intersect at a single point.  It is a generalization
 * of the method we used for baseFVD2
 * @param lines - Lines whos FVD we wish to compute
 * @return
 */
public static VoronoiDiagram singleIntersectionFVD(List<Line> lines) {
  VoronoiDiagram vd = new VoronoiDiagram();
  List<Line> newLines = new ArrayList<Line>();
  // Make a copy when we set the list of lines so we
  // don't run into list modification errors
  newLines.addAll(lines);
  vd.setLines(newLines);
  Line l1 = lines.get(lines.size()-1);
  Line l2 = lines.get(0);
  Line l3 = lines.get(1);
  Face f1 = new Face();
  Face f2 = new Face();
  // Calculate bisectors
  Line l4 = lineBetween(l1, l2);
  l4.setAngle((l4.getAngle()+Math.PI/2) % (Math.PI*2));
  Line l5 = lineBetween(l2, l3);
```

```
l5.setAngle((l5.getAngle()+Math.PI/2) % (Math.PI*2));
// If we wrapped around we need to add pi/2 to our bisectors
if(between(l3, l1, l2)) {
  l4.setAngle((l4.getAngle()+Math.PI/2) % (Math.PI*2));
}
if(between(l1, l2, l3)) {
  l5.setAngle((l5.getAngle()+Math.PI/2) % (Math.PI*2));
}
PointD interPoint = intersection(l1, l2);
Vertex v_inter = new Vertex(interPoint);

/*
 * Calculate the 4 points where our 2 angle bisectors
 * hit the bounding box and arrange them counter-clockwise
 * along the bounding box
 */
PointD[] bbinters = bbintersection(l4);
PointD p1 = bbinters[0];
PointD p2 = bbinters[1];
bbinters = bbintersection(l5);
PointD p3 = bbinters[0];
PointD p4 = bbinters[1];

PointD[] p = {p1,p2,p3,p4};
p1 = p[0];
p2 = findNextAlongBB(p1, p);
p3 = findNextAlongBB(p2, p);
p4 = findNextAlongBB(p3, p);

Vertex v1 = new Vertex(p1);
Vertex v2 = new Vertex(p2);
Vertex v3 = new Vertex(p3);
Vertex v4 = new Vertex(p4);

// Compute first face
HalfEdge[] path1 = findBBPath(p1,p2);
f1.setEdge(path1[0]);
for(HalfEdge e : path1) {
  e.setFace(f1);
}
HalfEdge toInter1 = new HalfEdge(v2, path1[path1.length-1], null, null, f1);
v2.setIncident(toInter1);
path1[path1.length-1].setNext(toInter1);
HalfEdge fromInter1 = new HalfEdge(v_inter, toInter1, path1[0], null, f1);
v_inter.setIncident(fromInter1);
toInter1.setNext(fromInter1);
path1[0].setPrev(fromInter1);

// Compute second face
HalfEdge[] path2 = findBBPath(p3,p4);
f2.setEdge(path2[0]);
for(HalfEdge e : path2) {
  e.setFace(f2);
}
HalfEdge toInter2 = new HalfEdge(v4, path2[path2.length-1], null, null, f2);
v4.setIncident(toInter2);
path2[path2.length-1].setNext(toInter2);
HalfEdge fromInter2 = new HalfEdge(v_inter, toInter2, path2[0], null, f2);
v_inter.setIncident(fromInter2);
toInter2.setNext(fromInter2);
path2[0].setPrev(fromInter2);

Face[] faces = {f1, f2};
vd.add(l2, faces);
HalfEdge firstFromInter1 = fromInter1;
HalfEdge firstFromInter2 = fromInter2;
/*
```

```
 * This loop performs the same operations as above
 * but starts connecting the edge to and from the
 * intersection point by setting their twins
 */
for(int i = 0; i < lines.size()-1; i++) {
  PointD oldp4 = p4;
  HalfEdge oldToInter1 = toInter1;
  HalfEdge oldToInter2 = toInter2;
  l1 = lines.get(i);
  l2 = lines.get((i+1)%lines.size());
  l3 = lines.get((i+2)%lines.size());
  f1 = new Face();
  f2 = new Face();
  l4 = lineBetween(l1, l2);
  l4.setAngle((l4.getAngle()+Math.PI/2) % (Math.PI*2));
  l5 = lineBetween(l2, l3);
  l5.setAngle((l5.getAngle()+Math.PI/2) % (Math.PI*2));
  if(between(l3, l1, l2)) {
    l4.setAngle((l4.getAngle()+Math.PI/2) % (Math.PI*2));
  }
  if(between(l1, l2, l3)) {
    l5.setAngle((l5.getAngle()+Math.PI/2) % (Math.PI*2));
  }
  bbinters = bbintersection(l4);
  p1 = bbinters[0];
  p2 = bbinters[1];
  bbinters = bbintersection(l5);
  p3 = bbinters[0];
  p4 = bbinters[1];

  // Swap in this case
  // if p1.equals(oldp2) do nothing
  if(p1.equals(oldp4)) {
    PointD temp = p1;
    p1 = p2;
    p2 = temp;
  }
  p = new PointD[]{p1,p2,p3,p4};
  p1 = p[0];
  p2 = findNextAlongBB(p1, p);
  p3 = findNextAlongBB(p2, p);
  p4 = findNextAlongBB(p3, p);


  v1 = new Vertex(p1);
  v2 = new Vertex(p2);
  v3 = new Vertex(p3);
  v4 = new Vertex(p4);

  path1 = findBBPath(p1,p2);
  f1.setEdge(path1[0]);
  for(HalfEdge e : path1) {
    e.setFace(f1);
  }
  toInter1 = new HalfEdge(v2, path1[path1.length-1], null, null, f1);
  v2.setIncident(toInter1);
  path1[path1.length-1].setNext(toInter1);
  fromInter1 = new HalfEdge(v_inter, toInter1, path1[0], null, f1);
  v_inter.setIncident(fromInter1);
  toInter1.setNext(fromInter1);
  path1[0].setPrev(fromInter1);
  fromInter1.setTwin(oldToInter1);
  oldToInter1.setTwin(fromInter1);

  path2 = findBBPath(p3,p4);
  f2.setEdge(path2[0]);
  for(HalfEdge e : path2) {
```

```
      e.setFace(f2);
    }
    toInter2 = new HalfEdge(v4, path2[path2.length-1], null, null, f2);
    v4.setIncident(toInter2);
    path2[path2.length-1].setNext(toInter2);
    fromInter2 = new HalfEdge(v_inter, toInter2, path2[0], null, f2);
    v_inter.setIncident(fromInter2);
    toInter2.setNext(fromInter2);
    path2[0].setPrev(fromInter2);
    fromInter2.setTwin(oldToInter2);
    oldToInter2.setTwin(fromInter2);

    faces = new Face[]{f1, f2};
    vd.add(l2, faces);
  }
  firstFromInter1.setTwin(toInter1);
  toInter1.setTwin(firstFromInter1);
  firstFromInter2.setTwin(toInter2);
  toInter2.setTwin(firstFromInter2);
  return vd;
}

/**
 * Compute the Voronoi Diagram in the special case where
 * all lines are parallel
 * @param lines - The lines who FVD we wish to compute
 * @return
 */
public static VoronoiDiagram allParallelFVD(List<Line> lines) {
  /*
   * The strategy is to find the two extreme parallel lines
   * by y-intercept and then use the line parallel and equidistant
   * to compute their associated faces
   */
  VoronoiDiagram vd = new VoronoiDiagram();
  Line l1, l2, midLine;
  // Special vertical line case
  if(lines.get(0).getAngle() == Math.PI/2) {
    double minX, maxX;
    Line minLine, maxLine;
    minX = maxX = lines.get(0).x;
    minLine = maxLine = lines.get(0);
    for(Line line : lines) {
      if(line.x < minX) {
        minX = line.x;
        minLine = line;
      }
      if(line.x > maxX) {
        maxX = line.x;
        maxLine = line;
      }
    }
    l1 = minLine;
    l2 = maxLine;
    midLine = new Line((l1.x+l2.x)/2, 0, Math.PI/2);
  }
  // General case
  else {
    double m1 = Math.tan(lines.get(0).getAngle());
    double b1 = lines.get(0).y - lines.get(0).x*m1;
    double minInter, maxInter;
    minInter = maxInter = b1;
    Line minLine, maxLine;
    minLine = maxLine = lines.get(0);
    for(Line line : lines) {
      double m = Math.tan(line.getAngle());
      double b = line.y - line.x*m;
      if(b < minInter) {
        minInter = b;
```

```
        minLine = line;
      }
      if(b > maxInter) {
        maxInter = b;
        maxLine = line;
      }
    }
    l1 = minLine;
    l2 = maxLine;
    midLine = new Line(0, (minInter+maxInter)/2, l1.getAngle());
  }

  Face f1 = new Face();
  Face f2 = new Face();

  PointD[] bbinters = bbintersection(midLine);
  PointD p1 = bbinters[0];
  PointD p2 = bbinters[1];
  Vertex v1 = new Vertex(p1);
  Vertex v2 = new Vertex(p2);

  HalfEdge[] path1 = findBBPath(p1,p2);
  f1.setEdge(path1[0]);
  for(HalfEdge e : path1) {
    e.setFace(f1);
  }
  HalfEdge toP1 = new HalfEdge(v2, path1[path1.length-1], path1[0], null, f1);
  v2.setIncident(toP1);
  path1[path1.length-1].setNext(toP1);
  path1[0].setPrev(toP1);

  HalfEdge[] path2 = findBBPath(p2,p1);
  f2.setEdge(path2[0]);
  for(HalfEdge e : path2) {
    e.setFace(f2);
  }
  HalfEdge toP2 = new HalfEdge(v1, path2[path2.length-1], path2[0], null, f2);
  v1.setIncident(toP2);
  path2[path2.length-1].setNext(toP2);
  path2[0].setPrev(toP2);
  toP2.setTwin(toP1);
  toP1.setTwin(toP2);

  boolean l1IntersectsF1 = false;
  HalfEdge start = f1.getEdge();
  HalfEdge curEdge = start;
  do {
    l1IntersectsF1 = intersection(l1, curEdge) != null;
  } while(!l1IntersectsF1 && (curEdge = curEdge.getNext()) != start);

  // If l1 intersects f1 then it doesn't own it
  if(l1IntersectsF1) {
    Line temp = l1;
    l1 = l2;
    l2 = temp;
  }
  // Link the two extreme lines together
  // so can easily find on using the other
  l1.setParallelPartner(l2);
  l2.setParallelPartner(l1);
  vd.add(l1, new Face[]{f1});
  vd.add(l2, new Face[]{f2});
  List<Line> list = new ArrayList<Line>();
  list.add(l1);
  list.add(l2);
  vd.setLines(list);
  return vd;
}
```

```
/**
 * Calculate the Voronoi Diagram of a single line
 * @param lines - line whose FVD we wish to compute
 * @return
 */
public static VoronoiDiagram baseFVD1(List<Line> lines) {
  /*
   * Use the line to divide the plane in two
   * so we have the entire plane in our preferred
   * two-face format
   */
  VoronoiDiagram vd = new VoronoiDiagram();
  List<Line> newLines = new ArrayList<Line>();
  newLines.add(lines.get(0));
  vd.setLines(newLines);
  Line line = lines.get(0);
  Face f1 = new Face();
  Face f2 = new Face();

  PointD[] bbinters = bbintersection(line);
  PointD p1 = bbinters[0];
  PointD p2 = bbinters[1];

  if(p1.x < p2.x) {
    PointD p3 = p1;
    p1 = p2;
    p2 = p3;
  }

  Vertex v1 = new Vertex(p1);
  Vertex v2 = new Vertex(p2);

  HalfEdge[] path1 = findBBPath(p1,p2);
  f1.setEdge(path1[0]);
  for(HalfEdge e : path1) {
    e.setFace(f1);
  }
  HalfEdge toP1 = new HalfEdge(v2, path1[path1.length-1], path1[0], null, f1);
  v2.setIncident(toP1);
  path1[path1.length-1].setNext(toP1);
  path1[0].setPrev(toP1);

  HalfEdge[] path2 = findBBPath(p2,p1);
  f2.setEdge(path2[0]);
  for(HalfEdge e : path2) {
    e.setFace(f2);
  }
  HalfEdge toP2 = new HalfEdge(v1, path2[path2.length-1], path2[0], null, f2);
  v1.setIncident(toP2);
  path2[path2.length-1].setNext(toP2);
  path2[0].setPrev(toP2);
  toP2.setTwin(toP1);
  toP1.setTwin(toP2);

  Face[] faces = new Face[2];
  faces[0] = f1;
  faces[1] = f2;

  vd.add(line, faces);
  return vd;
}

/**
 * Merge the Voronoi Diagrams of two sets of lines
 * by drawing the two necessary polygonal lines
 * @param vd1 FVD of first set of lines
 * @param vd2 FVD of second set of lines
 * @param lines Lines whose complete FVD we wish to compute
 * @return
```

```
 */
public static VoronoiDiagram mergeFVD(VoronoiDiagram vd1, VoronoiDiagram vd2, List<Line> lines) {
  VoronoiDiagram vd = null;

  List<Line> list = new ArrayList<Line>();
  list.addAll(vd1.getLines());
  list.addAll(vd2.getLines());
  drawPolygonalLine(vd1, vd2, list, true);
  drawPolygonalLine(vd1, vd2, list, false);
  // With the faces in each FVD cut down to size
  // we need only put them together
  vd = combine(vd1, vd2, list);
  return vd;
}

/**
 * Draw one of the polygonal lines between vd1 and vd2
 * @param vd1
 * @param vd2
 * @param lines
 * @param first  Whether to use the first or second bounding box
 *         intersection of one of our first two lines's angle
 *         bisector
 */
public static void drawPolygonalLine(VoronoiDiagram vd1, VoronoiDiagram vd2,
                                      List<Line> lines, boolean first) {
  Line l1 = lines.get(0);
  Line l2 = lines.get(lines.size()-1);
  // Get the bisctor that cuts through the obtuse angle of l1 and l2
  Line l3 = lineBetween(l1, l2);
  l3.setAngle((l3.getAngle()+Math.PI/2) % (Math.PI*2));
  // Fat angle might be wrong for densely angled group of lines
  // since fat angle may be owned by lines in the middle of the array
  boolean existsBetween = false;
  for(int i = 1; i < lines.size()-1; i++)
    existsBetween = existsBetween || strictBetween(lines.get(i), l1, l2);
  if(existsBetween) {
    l3.setAngle((l3.getAngle()+Math.PI/2) % (Math.PI*2));
  }

  PointD[] bb_inters = bbintersection(l3);
  // first polygonal line comes from one point
  // the second from the other point
  PointD p3 = first ? bb_inters[0] : bb_inters[1];

  /*
   * Start deciding which line face is on the left
   * side of the VD edge we're going to draw.
   * We'll call this line l1 and other l2 so that later
   * we'll always know that when l1's face ends we need
   * to patch it up so that the edges point in the same
   * direction as we're moving through the VD.  For l2, we
   * know ot always connect the edges backwards as we create them.
   * We'll call their corresponding VDs vd1 and vd2
   */
  // Previous intersection of polygonal line with a Voronoi edge
  PointD lastPoint = p3;
  PointD lastLastPoint = null; // The one before that
  HalfEdge lastEdge = null; // Last edge in one of the old FVDs we hit

  if(edgeContaining(vd1.getLineFaces(l1)[0], lastPoint) == null && vd1.getLineFaces(l1).length == 1)
    l1 = l1.getParallelPartner();
  if(edgeContaining(vd2.getLineFaces(l2)[0], lastPoint) == null && vd2.getLineFaces(l2).length == 1)
    l2 = l2.getParallelPartner();
  // If we had to switch out one of the parallel lines for its parallel partner
  // then recompute everything
  // We don't want to change the angle so just
  // compute the new intersection
  l3.setPoint(intersection(l1,l2));
```

```
bb_inters = bbintersection(l3);
// Since our parallel partner is farther from lastPoint
// this is the side it's face will be on
lastPoint = p3 = sameSide(bb_inters, l2, lastPoint);


// Check both faces of each line for the starting point
// Assert: lastEdge1 will never be null for parallel lines
//         since we are handling them above so we should never
//         get an array index out of bounds
HalfEdge lastEdge1 = edgeContaining(vd1.getLineFaces(l1)[0], lastPoint);
if (lastEdge1 == null)
    lastEdge1 = edgeContaining(vd1.getLineFaces(l1)[1], lastPoint);

HalfEdge lastEdge2 = edgeContaining(vd2.getLineFaces(l2)[0], lastPoint);
if (lastEdge2 == null)
    lastEdge2 = edgeContaining(vd2.getLineFaces(l2)[1], lastPoint);

Face f1 = lastEdge1.getFace();
Face f2 = lastEdge2.getFace();

// This is fine for first intersection - later ones should
// use "stay across line" method as implemented by findFirstIntersection
HalfEdge e1 = findFirstIntersectionExcluding(f1, lastEdge1, l3);
HalfEdge e2 = findFirstIntersectionExcluding(f2, lastEdge2, l3);

PointD interPoint = null;
// Take first intersection point
if(distance(lastPoint, intersection(l3, e2)) < distance(lastPoint, intersection(l3, e1)))
  interPoint = intersection(l3, e2);
else
  interPoint = intersection(l3, e1);
// Lines orthogonal to original lines
Line l2orth = new Line(intersection(l1, l2), l2.getAngle()+Math.PI/2);
Line l3orth = new Line(l2orth.getPoint(), l3.getAngle()+Math.PI/2);
// Take the bounding box intersection which is on the same side of l3orth
// as lastPoint
PointD bb_interOrth2 = sameSide(bbintersection(l2orth), l3orth, lastPoint);

// If l2's face is on the left of our polygonal line
// then switch l1 and l2 so that l1 has it
// (and switch VDsto match)
if(leftOn(lastPoint, interPoint, bb_interOrth2)) {
  Line tempLine = l1;
  l1 = l2;
  l2 = tempLine;
  VoronoiDiagram tempVD = vd1;
  vd1 = vd2;
  vd2 = tempVD;
  Face tempFace = f1;
  f1 = f2;
  f2 = tempFace;
  HalfEdge tempEdge = lastEdge1;
  lastEdge1 = lastEdge2;
  lastEdge2 = tempEdge;
  tempEdge = e1;
  e1 = e2;
  e2 = tempEdge;
}
HalfEdge lastEdgeIntersected1 = e1;
HalfEdge lastEdgeIntersected2 = e2;
/*
 * End deciding.  l1 now has the property
 * described above
 */
// New origin of lastEdge2
// Don't set it 'til we've finishe drawing the face
Vertex start2 = new Vertex(lastPoint);
start2.setIncident(lastEdge2);
```

```
// These are the new edges we've drawn with our polygonal line
// for the left (1) and right (2) faces
List<HalfEdge> queueEdges1 = new ArrayList<HalfEdge>();
queueEdges1.add(lastEdge1);
List<HalfEdge> queueEdges2 = new ArrayList<HalfEdge>();
queueEdges2.add(lastEdge2);

boolean firstRun = true;
while(true) {
  HalfEdge mostRecent1 = queueEdges1.get(queueEdges1.size()-1);
  HalfEdge mostRecent2 = queueEdges2.get(queueEdges2.size()-1);
  Line l4 = lineBetween(l1, l2);
  // Take one bisector and if it's not the right one
  // (i.e. we're not currently one it), take the other one
  if(!onLine(lastPoint, l4))
    l4.setAngle(l4.getAngle()+Math.PI/2);
  l3 = l4;

  // For the first run we've already computed this
  // Otherwise compute the next edge our polygonal line hits
  if(!firstRun) {
    /*
     * Keep track of last edge intersected for face we haven't completed
     * We can ensure O(nlogn) by moving counter-clockwise around the left face
     * and clockwise around the right face, so we don't O(n) searches of a face
     * that could potentially contain O(n) edges resulting in O(n^2) run time
     */
    if(lastEdgeIntersected1 != null)
      e1 = findFirstIntersection(f1, lastLastPoint, lastEdge, l3, lastEdgeIntersected1, true);
    else
      e1 = findFirstIntersection(f1, lastLastPoint, lastEdge, l3, f1.getEdge(), true);
    if(lastEdgeIntersected2 != null)
      e2 = findFirstIntersection(f2, lastLastPoint, lastEdge, l3, lastEdgeIntersected2, false);
    else
      e2 = findFirstIntersection(f2, lastLastPoint, lastEdge, l3, f2.getEdge(), false);
    lastEdgeIntersected1 = e1;
    lastEdgeIntersected2 = e2;
  }
  // If we've hit the bounding box finish both faces
  // and break out of the loop
  if(onBB(intersection(l3, e2)) && onBB(intersection(l3, e1))) {
    interPoint = intersection(l3, e1);
    queueEdges2.get(0).setOrig(start2);
    if(queueEdges2.size() > 1)
      queueEdges2.get(0).setPrev(queueEdges2.get(1));
    if(queueEdges1.size() > 1) {
      queueEdges1.get(0).setNext(queueEdges1.get(1));
    }

    Vertex lastVert = new Vertex(lastPoint);
    HalfEdge newEdge1 = new HalfEdge(lastVert, mostRecent1, e1, null, f1);
    lastVert.setIncident(newEdge1);
    e1.setPrev(newEdge1);
    mostRecent1.setNext(newEdge1);
    f1.setEdge(newEdge1); // We could have cut out edge f1 used to point to

    Vertex interVert = new Vertex(interPoint);
    interVert.setIncident(e1);
    e1.setOrig(interVert);

    HalfEdge newEdge2 = new HalfEdge(interVert, e2, mostRecent2, newEdge1, f2);
    mostRecent2.setPrev(newEdge2);
    interVert.setIncident(newEdge2);
    e2.setNext(newEdge2);
    f2.setEdge(newEdge2); // We could have cut out edge f2 used to point to

    newEdge1.setTwin(newEdge2);
```

```
    break;
  }
  // If we e1 is closer than e2 then close up f1 and start a new one
  // on the opposite side of e1 while adding a new edge to queueEdges2
  if(distance(lastPoint, intersection(l3, e2)) >= distance(lastPoint, intersection(l3, e1))) {
    lastEdgeIntersected1 = null;
    lastEdge = e1;
    interPoint = intersection(l3, e1);

    if(queueEdges1.size() > 1) {
      queueEdges1.get(0).setNext(queueEdges1.get(1));
    }
    Vertex lastVert = new Vertex(lastPoint);
    HalfEdge newEdge1 = new HalfEdge(lastVert, mostRecent1, e1, null, f1);
    lastVert.setIncident(newEdge1);
    e1.setPrev(newEdge1);
    mostRecent1.setNext(newEdge1);
    f1.setEdge(newEdge1); // We could have cut out edge f1 used to point to

    Vertex interVert = new Vertex(interPoint);
    interVert.setIncident(e1);
    e1.setOrig(interVert);
    HalfEdge newEdge2 = new HalfEdge(interVert, null, mostRecent2, newEdge1, f2);
    if(queueEdges2.size() > 1)
      mostRecent2.setPrev(newEdge2);
    newEdge1.setTwin(newEdge2);
    queueEdges2.add(newEdge2);

    f1 = e1.getTwin().getFace();
    l1 = vd1.getFaceLine(f1);

    queueEdges1 = new ArrayList<HalfEdge>();
    queueEdges1.add(e1.getTwin());

  }
  // If we e2 is closer than e1 then close up f2 and start a new one
  // on the opposite side of e2 while adding a new edge toqueueEdges1
  else {
    lastEdgeIntersected2 = null;
    lastEdge = e2;
    interPoint = intersection(l3, e2);
    queueEdges2.get(0).setOrig(start2);
    if(queueEdges2.size() > 1)
      queueEdges2.get(0).setPrev(queueEdges2.get(1));
    Vertex interVert = new Vertex(interPoint);
    HalfEdge newEdge2 = new HalfEdge(interVert, e2, mostRecent2, null, f2);
    interVert.setIncident(newEdge2);
    e2.setNext(newEdge2);
    mostRecent2.setPrev(newEdge2);
    Vertex lastVert = new Vertex(lastPoint);
    f2.setEdge(newEdge2); // We could have cut out the edge f2 used to point to
    // Usually redundant but necessary
    // for starting correctly
    mostRecent2.setOrig(lastVert);

    // Create queue 2 edges but wait until we finish the new face
    // to connect the first one to prevent curEdge.getPrev() from
    // returning null in various functions
    HalfEdge newEdge1 = new HalfEdge(lastVert, mostRecent1, null, newEdge2, f1);
    if(queueEdges1.size() > 1)
      mostRecent1.setNext(newEdge1);
    newEdge2.setTwin(newEdge1);
    queueEdges1.add(newEdge1);

    f2 = e2.getTwin().getFace();
    l2 = vd2.getFaceLine(f2);
```

```
        HalfEdge twin = e2.getTwin();
        queueEdges2 = new ArrayList<HalfEdge>();
        queueEdges2.add(twin);
        // Don't move e2 orig to interVert until
        // we've finished the face
        start2 = interVert;
        start2.setIncident(lastEdge2);
      }

      lastLastPoint = lastPoint;
      lastPoint = interPoint;
      firstRun = false;
    }
  }

  /**
   * Find the intereseciton point of two lines
   * @param l1
   * @param l2
   * @return
   */
  public static PointD intersection(Line l1, Line l2) {
    if(l1.getAngle() == l2.getAngle())
      return null;

    // Vertical line
    if(fuzzyEqual(l1.getAngle(), Math.PI/2)) {
      PointD p = new PointD(l1.x, 0);
      double m2 = Math.tan(l2.getAngle());
      double b2 = l2.y - l2.x*m2;
      p.y = m2*p.x + b2;
      return p;
    }
    if(fuzzyEqual(l2.getAngle(), Math.PI/2)) {
      PointD p = new PointD(l2.x, 0);
      double m1 = Math.tan(l1.getAngle());
      double b1 = l1.y - l1.x*m1;
      p.y = m1*p.x + b1;
      return p;
    }

    // Because of double imprecision we need not worry
    // about dividing by zero here
    double m1 = Math.tan(l1.getAngle());
    double m2 = Math.tan(l2.getAngle());

    double b1 = l1.y - l1.x*m1;
    double b2 = l2.y - l2.x*m2;

    double x = (b2-b1)/(m1-m2);
    double y = m1*x+b1;

    return new PointD(x,y);
  }

  /**
   * Find the intereseciton point of a line and segment
   * if such an intersection exists
   * @param l1
   * @param e
   * @return
   */
  public static PointD intersection(Line l1, HalfEdge e) {
    // Convert edge to a line and use line-line
    // intersectionn method
    PointD p1 = e.getOrig().getPoint();
    PointD p2 = e.getNext().getOrig().getPoint();
    double angle = Math.atan((p2.y-p1.y)/(p2.x-p1.x));
    Line l2 = new Line(p1, angle);
    PointD interPoint = intersection(l1, l2);
```

```
  // Check that intersection lies on the edge
  if(interPoint == null)
    return null;
  if(between(interPoint.x, p1.x, p2.x) && between(interPoint.y, p1.y, p2.y))
    return interPoint;
  return null;
}

/**
 * Return true if the value of a lies between or on b and c
 * allowing for double imprecision with a fudge factor
 * @param a
 * @param b
 * @param c
 * @return
 */
public static boolean between (double a, double b, double c) {
  if(c > b)
    return a - FUDGE <= c && a + FUDGE >= b;
  if(c < b)
    return a + FUDGE >= c && a - FUDGE <= b;
  return fuzzyEqual(a, c) && fuzzyEqual(a, b);
}

/**
 * Return true if l1's angle lies strictly
 * in the acute angle between l2 and l3
 * @param l1
 * @param l2
 * @param l3
 * @return
 */
public static boolean strictBetween (Line l1, Line l2, Line l3) {
  if(Math.abs(l3.getAngle() - l2.getAngle()) < Math.PI/2) {
     return strictBetween (l1.getAngle(), l2.getAngle(), l3.getAngle());
  }
  else {
    double max = Math.max(l3.getAngle(), l2.getAngle());
    double min = Math.min(l3.getAngle(), l2.getAngle());
    return strictBetween (l1.getAngle(), min+Math.PI, max) ||
           strictBetween (l1.getAngle(), min, max-Math.PI);
  }
}

/**
 * Return true if the value of a lies strictly between b and c
 * @param a
 * @param b
 * @param c
 * @return
 */
public static boolean strictBetween (double a, double b, double c) {
  if(c > b)
    return a < c && a > b;
  if(c < b)
    return a > c && a < b;
  return false;
}

/**
 * Find a line's two intersection points with
 * the bounding box
 * @param line
 * @return
 */
public static PointD[] bbintersection(Line line) {
  PointD p1 = null, p2 = null;
  top_inter = intersection(line, new Line(0, top_bound, 0));
  bottom_inter= intersection(line, new Line(0, bottom_bound, 0));
```

```
  right_inter = intersection(line, new Line(right_bound, 0, Math.PI/2));
  left_inter = intersection(line, new Line(left_bound, 0, Math.PI/2));

  // Check if we intersected the top and if so whether we did so before hitting
  // the left and right sides of the bounding box
  // We do the same for each intersection
  if(top_inter != null && top_inter.x <= right_bound && top_inter.x >= left_bound) {
    p1 = top_inter;
  }
  if(bottom_inter != null && bottom_inter.x <= right_bound && bottom_inter.x >= left_bound)
    if(p1 == null)
      p1 = bottom_inter;
    else
      p2 = bottom_inter;

  if(p2 == null) {
    if(right_inter != null && right_inter.y <= top_bound && right_inter.y >= bottom_bound) {
      if(p1 == null)
        p1 = right_inter;
      else
        if(!p1.equals(right_inter))
          p2 = right_inter;
    }
    if(p1 == null || p2 == null) {
      if(left_inter != null && left_inter.y <= top_bound && left_inter.y >= bottom_bound) {
        p2 = left_inter;
      }
    }

  }
  // This handles the rare case that our point
  // is on the corner of the bounding box, which is a
  // special case we can avoid by nudging it one way or the other
  moveOffBounding(p1);
  moveOffBounding(p2);

  PointD[] bb_inters = new PointD[2];
  bb_inters[0] = p1;
  bb_inters[1] = p2;

  return bb_inters;
}

/**
 * In case point is exactly on corner of bounding
 * box move it right or left.
 * Given double precision this is an extraordinary case
 * @param p
 */
public static void moveOffBounding(PointD p) {
  if(p.equals(top_right))
    p.x = p.x - FUDGE;
  if(p.equals(bottom_right))
    p.x = p.x - FUDGE;
  if(p.equals(top_left))
    p.x = p.x + FUDGE;
  if(p.equals(bottom_left))
    p.x = p.x + FUDGE;
}

/**
 * Compute Euclidian distance between two vertices
 * @param v1
 * @param v2
 * @return
 */
public static double distance (Vertex v1, Vertex v2) {
  return distance(v1.getPoint(), v2.getPoint());
}
/**
```

```
 * Compute Euclidian distance between two points
 * @param v1
 * @param v2
 * @return
 */
public static double distance (PointD p1, PointD p2) {
  return Math.sqrt(Math.pow(p2.x-p1.x, 2) + Math.pow(p2.y-p1.y, 2));
}

/**
 * Give the line passing through the intersection
 * of l1 and l2 and also passing through the area
 * swept out by their acute angle.
 * @param l1
 * @param l2
 * @return
 */
public static Line lineBetween (Line l1, Line l2) {
  Line lret = null;
  if(Math.abs(l1.getAngle() - l2.getAngle()) <= Math.PI/2)
    lret = new Line(intersection(l1, l2), (l1.getAngle() + l2.getAngle())/2);
  else
    lret = new Line(intersection(l1, l2), (l1.getAngle()+Math.PI + l2.getAngle())/2);

  return lret;
}

/**
 * Return true if l1's angle lies in the acute angle between
 * l2 and l3
 * @param l1
 * @param l2
 * @param l3
 * @return
 */
public static boolean between (Line l1, Line l2, Line l3) {
  if(Math.abs(l3.getAngle() - l2.getAngle()) < Math.PI/2) {
     return between (l1.getAngle(), l2.getAngle(), l3.getAngle());
  }
  else {
    double max = Math.max(l3.getAngle(), l2.getAngle());
    double min = Math.min(l3.getAngle(), l2.getAngle());
    return between (l1.getAngle(), min+Math.PI, max) || between (l1.getAngle(), min, max-Math.PI);
  }
}
/**
 * This method has no absolute meaning since a line has
 * no direction but is useful in determining if two points
 * are on the same side of a line
 * @param l
 * @param query
 * @return
 */
public static boolean leftOn (Line l, PointD query) {
  PointD p1 = l.getPoint();
  PointD p2 = l.getPoint();
  p2.x = p2.x + Math.cos(l.getAngle());
  p2.y = p2.y + Math.sin(l.getAngle());
      return area2(p1, p2, query) >= 0;
}

/**
 * Determines if the query point lies on or to the left of
 * the directed edge e
 * @param e
 * @param query
 * @return
 */
public static boolean leftOn (HalfEdge e, PointD query) {
  return leftOn(e.getOrig().getPoint(), e.getNext().getOrig().getPoint(), query);
```

```
}

/**
 * Determines if the query point lies on or to the left of
 * the directed edge from p1 to p2
 * @param p1
 * @param p2
 * @param query
 * @return
 */
public static boolean leftOn (PointD p1, PointD p2, PointD query) {
      return area2(p1, p2, query) >= 0;
}

/**
 * Computes the area of three points.  It's positive if they're
 * in counter-clockwise order and negative otherwise
 * @param a
 * @param b
 * @param c
 * @return
 */
public static double area2(PointD a, PointD b, PointD c) {
    return (b.x-a.x)*(c.y-a.y)-(c.x-a.x)*(b.y-a.y);
}

/**
 * For the given face find which of its edges
 * is intersected by l excluding excludedEdge
 * @param f
 * @param excludedEdge
 * @param l
 * @return
 */
public static HalfEdge findFirstIntersectionExcluding(Face f, HalfEdge excludedEdge,
                                                      Line l) {
  HalfEdge start = f.getEdge();
  HalfEdge curEdge = start;

  do {
    if(curEdge.equals(excludedEdge))
      continue;
    PointD interPoint = intersection (l, curEdge);
    if(interPoint != null)
      return curEdge;
  } while ((curEdge = curEdge.getNext()) != start);

  return null;
}

/*
 * After all the areas owned by different lines are
 * corrected this method simply combines the two diagrams
 */
public static VoronoiDiagram combine (VoronoiDiagram vd1, VoronoiDiagram vd2, List<Line> lines) {
  VoronoiDiagram vd = new VoronoiDiagram();
  vd.setFaceToLine(vd1.getFaceToLine());
  vd.getFaceToLine().putAll(vd2.getFaceToLine());
  vd.setLineToFace(vd1.getLineToFace());
  vd.getLineToFace().putAll(vd2.getLineToFace());
  vd.setLines(lines);
  return vd;
}

/**
 * Finds which edge in the face contains the given point
 * @param f
 * @param p
 * @return
 */
public static HalfEdge edgeContaining(Face f, PointD p) {
```

```
  HalfEdge start = f.getEdge();
  HalfEdge curEdge = start;
  HalfEdge winner = null;
  do {
    PointD p1 = curEdge.getOrig().getPoint();
    PointD p2 = curEdge.getNext().getOrig().getPoint();
    if(!between(p.x, p1.x, p2.x) || !between(p.y, p1.y, p2.y))
        continue;

    PointD p3;
    if((p2.x - p1.x) != 0) {
      double slope = (p2.y-p1.y)/(p2.x - p1.x);
      // Where p should lie if it's on the edge
      p3 = new PointD(p.x, (p.x-p1.x)*slope + p1.y);
    }
    else {
      p3 = new PointD(p1.x, p.y);
    }
    if(distance(p, p3) < FUDGE) {
      winner = curEdge;
    }
  } while((curEdge = curEdge.getNext()) != start);

  return winner;
}

/**
 * Return the first point in the array queryPoints which
 * is on the same side of line l as point p
 * @param queryPoints
 * @param l
 * @param p
 * @return
 */
public static PointD sameSide(PointD[] queryPoints, Line l, PointD p) {
  for(PointD point : queryPoints) {
    if(leftOn(l, point) == leftOn(l, p))
      return point;
  }
  return null;
}

/**
 * Returns true if point lies on the bounding box
 * @param p
 * @return
 */
public static boolean onBB(PointD p) {
  if(fuzzyEqual(p.x, right_bound))
    return true;
  if(fuzzyEqual(p.x, left_bound))
    return true;
  if(fuzzyEqual(p.y, top_bound))
    return true;
  if(fuzzyEqual(p.y, bottom_bound))
    return true;
  return false;
}

/**
 * Find next point counterclockwise along
 * the bounding box from p in the array points
 * @param p
 * @param points
 * @return
 */
public static PointD findNextAlongBB(PointD p, PointD[] points) {

  // Loop around the bounding box twice in case we start on the bottom
  // and need to walk all the way back around to the bottom
```

```
  int runs = 0;
  PointD curPoint = p;
  while(runs++ < 2) {
    if(fuzzyEqual(curPoint.x, right_bound)) {
      PointD winner = top_right;
      for(PointD point : points) {
        if(fuzzyEqual(point.x, right_bound) && point.y > curPoint.y && point.y < winner.y)
          winner = point;
      }
      if(winner != top_right)
        return winner;
      curPoint = top_right;
    }
    if(fuzzyEqual(curPoint.y, top_bound)) {
      PointD winner = top_left;
      for(PointD point : points) {
        if(fuzzyEqual(point.y, top_bound) && point.x < curPoint.x && point.x > winner.x)
          winner = point;
      }
      if(winner != top_left)
        return winner;
      curPoint = top_left;
    }
    if(fuzzyEqual(curPoint.x, left_bound)) {
      PointD winner = bottom_left;
      for(PointD point : points) {
        if(fuzzyEqual(point.x, left_bound) && point.y < curPoint.y && point.y > winner.y)
          winner = point;
      }
      if(winner != bottom_left)
        return winner;
      curPoint = bottom_left;
    }
    if(fuzzyEqual(curPoint.y, bottom_bound)) {
      PointD winner = bottom_right;
      for(PointD point : points) {
        if(fuzzyEqual(point.y, bottom_bound) && point.x > curPoint.x && point.x < winner.x)
          winner = point;
      }
      if(winner != bottom_right)
        return winner;
      curPoint = bottom_right;
    }
  }
  return null;
}

/**
 * Find the counter-clockwise path along the bounding box
 * from p1 to p2
 * @param p1
 * @param p2
 * @return
 */
public static HalfEdge[] findBBPath(PointD p1, PointD p2) {
  ArrayList<HalfEdge> path = new ArrayList<HalfEdge>();

  Vertex v1 = new Vertex(p1);
  // Loop around the bounding box until we hit p2
  PointD lastPoint = p1;
  Vertex lastVert = v1;
  while(true) {
    // Keep walking around the bounding box and every time we
    // find a new vertex continue so we can come up here and
    // add an edge
    if(path.size() > 0) {
      HalfEdge e = new HalfEdge(lastVert, path.get(path.size()-1), null, null, null);
      lastVert.setIncident(e);
      path.get(path.size()-1).setNext(e);
```

```
      path.add(e);
    }
    else {
      HalfEdge e = new HalfEdge(lastVert, null, null, null, null);
      path.add(e);
    }
    if(fuzzyEqual(lastPoint.x, right_bound) && lastPoint != top_right) {
      if(fuzzyEqual(p2.x, right_bound) && p2.y > lastPoint.y) {
        break;
      }
      else {
        lastPoint = top_right;
        lastVert = new Vertex(lastPoint);
        continue;
      }
    }
    if(fuzzyEqual(lastPoint.y, top_bound) && lastPoint != top_left) {
      if(fuzzyEqual(p2.y, top_bound) && p2.x < lastPoint.x) {
        break;
      }
      else {
        lastPoint = top_left;
        lastVert = new Vertex(lastPoint);
        continue;
      }
    }
    if(fuzzyEqual(lastPoint.x, left_bound) && lastPoint != bottom_left) {
      if(fuzzyEqual(p2.x, left_bound) && p2.y < lastPoint.y) {
        break;
      }
      else {
        lastPoint = bottom_left;
        lastVert = new Vertex(lastPoint);
        continue;
      }
    }
    if(fuzzyEqual(lastPoint.y, bottom_bound) && lastPoint != bottom_right) {
      if(fuzzyEqual(p2.y, bottom_bound) && p2.x > lastPoint.x) {
        break;
      }
      else {
        lastPoint = bottom_right;
        lastVert = new Vertex(lastPoint);
        continue;
      }
    }
  }
  return path.toArray(new HalfEdge[0]);
}

/**
 * returns whether d1 and d2 are withing FUDGE
 * of one another
 * @param d1
 * @param d2
 * @return
 */
public static boolean fuzzyEqual(double d1, double d2) {
  return Math.abs(d1-d2) < FUDGE;
}

/**
 * Find the first edge in face f which we intersect
 * and which lies on the the opposite side of lastEdge
 * as lastLastPoint starting from start.
 * We proceed using .getNext() or .getPrev() depending
 * depending on whether forward is true
 * @param f
 * @param lastLastPoint
```

```
 * @param lastEdge
 * @param l
 * @param start
 * @param forward
 * @return
 */
public static HalfEdge findFirstIntersection(Face f, PointD lastLastPoint, HalfEdge lastEdge, Line l, Ha
  HalfEdge curEdge = start;

  do {
    if(curEdge == lastEdge.getTwin()) {
      continue;
    }
    PointD interPoint = intersection (l, curEdge);
    if(interPoint == null)
      continue;
    if(leftOn(lastEdge, interPoint) != leftOn(lastEdge, lastLastPoint)) {
      return curEdge;
    }
  } while ((forward ? (curEdge = curEdge.getNext()) : (curEdge = curEdge.getPrev())) != start);

  System.err.println("Error in finding exit edge");
  return null;
}

/**
 * Returns whether the point p is on the line l
 * @param p
 * @param l
 * @return
 */
public static boolean onLine(PointD p, Line l) {
  // Calculation is unreliable for near-vertical slopes
  // Since slope is cloe to infinity
  if(fuzzyEqual(l.getAngle(), Math.PI/2))
      return fuzzyEqual(l.x, p.x);
  double slope = Math.tan(l.getAngle());
  PointD p3 = new PointD(p.x, (p.x-l.x)*slope + l.y);
  return (distance(p, p3) < FUDGE);
}

/**
 * Computes the lenght of the diagonal of the bounding box
 * of all the intersection points among our lines
 * @param lines
 * @return
 */
public static double findMaxDist(List<Line> lines) {
  // If we have only one intersection or their all parallel
  // return our constant
  if(lines.size() < 2 || lines.get(0).getAngle() == lines.get(lines.size()-1).getAngle())
    return MAX_DIST_MARGIN;
  // We need only examine intersections between angle-adjacent lines
  DimensionD boundingBox = new DimensionD(intersection(lines.get(0), lines.get(lines.size()-1)));
  for(int i = 0; i < lines.size()-1; i++) {
    PointD inter_point = intersection(lines.get(i), lines.get(i+1));
    if (inter_point != null)
      boundingBox.includePoint(inter_point);
  }
  PointD top_left = new PointD(boundingBox.left, boundingBox.top);
  PointD bottom_right = new PointD(boundingBox.right, boundingBox.bottom);
  double dist = distance(top_left, bottom_right);
  dist += MAX_DIST_MARGIN; //Fudge factor for case of fewer than 3 lines
  return dist;
}

/**
 * Compute a suitable bounding box for our Voronoi diagram
 * @param lines
```

```
 * @return
 */
public static DimensionD findVDBoundingBox(List<Line> lines) {
  DimensionD d = new DimensionD();

  int numOrientations = 0;
  Line last = null;
  for(Line l : lines) {
    if(last == null)
      numOrientations++;
    else
      if(last.getAngle() != l.getAngle())
        numOrientations++;
    last = l;
  }
  // For a single orientation we just
  // want to be able to see all the lines
  if(numOrientations == 1) {
    for(Line l : lines)
      d.includePoint(l.getPoint());
    d.top += BOUNDING_ROOM;
    d.bottom -= BOUNDING_ROOM;
    d.left -= BOUNDING_ROOM;
    d.right += BOUNDING_ROOM;
    return d;
  }
  // For two orientations compute all intersections
  // of both bisectors of each pair of extreme lines
  else if(numOrientations == 2) {
    Line o1e1, o1e2, o2e1, o2e2;
    Line start, l;
    start = l = lines.get(0);
    o1e1 = o1e2 = start;
    double minB, maxB;
    double m = Math.tan(start.getAngle());
    double b = start.y - start.x*m;
    minB = maxB = b;

    int i;
    for(i = 0; lines.get(i).getAngle() == start.getAngle(); i++) {
      l = lines.get(i);
      m = Math.tan(l.getAngle());
      b = l.y - l.x*m;
      if(b > maxB)
        o1e1 = l;
      if(b < minB)
        o1e2 = l;
    }
    start = lines.get(i);
    o2e1 = o2e2 = start;
    m = Math.tan(start.getAngle());
    b = start.y - start.x*m;
    minB = maxB = b;
    for(; i < lines.size(); i++) {
      l = lines.get(i);
      m = Math.tan(l.getAngle());
      b = l.y - l.x*m;
      if(b > maxB)
        o2e1 = l;
      if(b < minB)
        o2e2 = l;
    }
    Line[] bisectors = new Line[8];
    bisectors[0] = lineBetween(o1e1, o2e1);
    bisectors[1] = lineBetween(o1e1, o2e1);
    bisectors[1].setAngle(bisectors[1].getAngle()+Math.PI/2);
    bisectors[2] = lineBetween(o1e1, o2e2);
    bisectors[3] = lineBetween(o1e1, o2e2);
    bisectors[3].setAngle(bisectors[3].getAngle()+Math.PI/2);
    bisectors[4] = lineBetween(o1e2, o2e1);
```

```
    bisectors[5] = lineBetween(o1e2, o2e1);
    bisectors[5].setAngle(bisectors[5].getAngle()+Math.PI/2);
    bisectors[6] = lineBetween(o1e2, o2e2);
    bisectors[7] = lineBetween(o1e2, o2e2);
    bisectors[7].setAngle(bisectors[7].getAngle()+Math.PI/2);

    for(Line l1 : bisectors) {
      for(Line l2 : bisectors) {
        if(fuzzyEqual(l1.getAngle(),l2.getAngle()))
          continue;
        d.includePoint(intersection(l1, l2));
      }
    }

    // Give some extra room on each side
    d.top += BOUNDING_ROOM;
    d.bottom -= BOUNDING_ROOM;
    d.left -= BOUNDING_ROOM;
    d.right += BOUNDING_ROOM;
    return d;
  }
  // ASSERT: numOrientations >= 3
  double distMax = findMaxDist(lines);
  // Move the intersection point of l1 and l3 so it is
  // distMax units from the intersection of l1 and l2
  // Then compute the intersection of the bisectors of
  // l1 and l2, and l1 and l3
  for(int i = 0; i < lines.size(); i++) {
    Line l1 = lines.get(i);
    Line l2 = lines.get((i+1)%lines.size());
    Line l3 = lines.get((i+2)%lines.size());
    // Always skip forward until we have three non-parallel lines
    int j = i + 1;
    if(l1.getAngle() == l2.getAngle()) {
      while (l1.getAngle() == lines.get((++j)%lines.size()).getAngle()) { }
      l2 = lines.get(j%lines.size());
      l3 = lines.get((++j)%lines.size());
    }
    if(l2.getAngle() == l3.getAngle()) {
      while (l2.getAngle() == lines.get((++j)%lines.size()).getAngle()) { }
      l3 = lines.get(j%lines.size());
    }
    double[] newb3s = findNewIntercepts(l1, l2, l3, distMax);
    Line firstl3 = new Line(0, newb3s[0], l3.getAngle());
    Line secondl3 = new Line(0, newb3s[1], l3.getAngle());
    Line l1l2bisector = boundingBoxBisector(l1, l2);
    Line l1firstl3bisector = boundingBoxBisector(l1, firstl3);
    Line l1secondl3bisector = boundingBoxBisector(l1, secondl3);
    boundingBoxBisector(l1, secondl3);
    if(intersection(l1l2bisector, l1firstl3bisector) != null)
      d.includePoint(intersection(l1l2bisector, l1firstl3bisector));
    if(intersection(l1l2bisector, l1secondl3bisector) != null)
      d.includePoint(intersection(l1l2bisector, l1secondl3bisector));

  }
  // Give some extra room on each side
  d.top += BOUNDING_ROOM;
  d.bottom -= BOUNDING_ROOM;
  d.left -= BOUNDING_ROOM;
  d.right += BOUNDING_ROOM;
  return d;
}

/**
 * Find the two intercepts for l3 which would make it so that
 * the intersection point of l1 and l3 so it is
 * distMax units from the intersection of l1 and l2
 * @param l1
```

```
 * @param l2
 * @param l3
 * @param distMax
 * @return
 */
public static double[] findNewIntercepts(Line l1, Line l2, Line l3, double distMax) {
  double m1 = Math.tan(l1.getAngle());
  double m2 = Math.tan(l2.getAngle());
  double m3 = Math.tan(l3.getAngle());
  double b1 = l1.y - l1.x*m1;
  double b2 = l2.y - l2.x*m2;
  double[] intercepts = new double[2];
  intercepts[0] = 1.0/2*(2*b2*Math.pow(m1,3)+2*m3*b1*Math.pow(m1,2)-2*b1*m2*Math.pow(m1,2)-
                  2*Math.pow(m1,2)*b2*m3+2*b2*m1+2*b1*m3-2*b1*m2-2*b2*m3+2*
                  Math.pow((Math.pow(distMax,2)*Math.pow(m1,4)+Math.pow(m1,6)*Math.pow(distMax,2)+
                  4*Math.pow(m1,4)*Math.pow(distMax,2)*m2*m3-2*Math.pow(m1,3)*
                  Math.pow(distMax,2)*m2*Math.pow(m3,2)-2*Math.pow(m1,3)*Math.pow(distMax,2)*
                  Math.pow(m2,2)*m3+Math.pow(m1,2)*Math.pow(distMax,2)*Math.pow(m2,2)*
                  Math.pow(m3,2)+Math.pow(m1,4)*Math.pow(distMax,2)*Math.pow(m3,2)-
                  2*Math.pow(m1,5)*Math.pow(distMax,2)*m3-2*Math.pow(m1,5)*Math.pow(distMax,2)*
                  m2+Math.pow(m1,4)*Math.pow(distMax,2)*Math.pow(m2,2)+4*Math.pow(distMax,2)*
                  Math.pow(m1,2)*m2*m3-2*Math.pow(distMax,2)*m1*m2*Math.pow(m3,2)-2*
                  Math.pow(distMax,2)*Math.pow(m2,2)*m1*m3+Math.pow(distMax,2)*Math.pow(m1,2)*
                  Math.pow(m3,2)-2*Math.pow(distMax,2)*Math.pow(m1,3)*m3-2*Math.pow(distMax,2)*
                  Math.pow(m1,3)*m2+Math.pow(distMax,2)*Math.pow(m2,2)*Math.pow(m1,2)+
                  Math.pow(distMax,2)*Math.pow(m2,2)*Math.pow(m3,2)),(1.0/2)))/
                  ((1+Math.pow(m1,2))*(m1-m2));
  intercepts[1] = 1.0/2*(2*b2*Math.pow(m1,3)+2*m3*b1*Math.pow(m1,2)-2*b1*m2*Math.pow(m1,2)-
                  2*Math.pow(m1,2)*b2*m3+2*b2*m1+2*b1*m3-2*b1*m2-2*b2*m3-2*
                  Math.pow((Math.pow(distMax,2)*Math.pow(m1,4)+Math.pow(m1,6)*Math.pow(distMax,2)+
                  4*Math.pow(m1,4)*Math.pow(distMax,2)*m2*m3-2*Math.pow(m1,3)*
                  Math.pow(distMax,2)*m2*Math.pow(m3,2)-2*Math.pow(m1,3)*Math.pow(distMax,2)*
                  Math.pow(m2,2)*m3+Math.pow(m1,2)*Math.pow(distMax,2)*Math.pow(m2,2)*
                  Math.pow(m3,2)+Math.pow(m1,4)*Math.pow(distMax,2)*Math.pow(m3,2)-
                  2*Math.pow(m1,5)*Math.pow(distMax,2)*m3-2*Math.pow(m1,5)*Math.pow(distMax,2)*
                  m2+Math.pow(m1,4)*Math.pow(distMax,2)*Math.pow(m2,2)+4*Math.pow(distMax,2)*
                  Math.pow(m1,2)*m2*m3-2*Math.pow(distMax,2)*m1*m2*Math.pow(m3,2)-2*
                  Math.pow(distMax,2)*Math.pow(m2,2)*m1*m3+Math.pow(distMax,2)*Math.pow(m1,2)*
                  Math.pow(m3,2)-2*Math.pow(distMax,2)*Math.pow(m1,3)*m3-2*Math.pow(distMax,2)*
                  Math.pow(m1,3)*m2+Math.pow(distMax,2)*Math.pow(m2,2)*Math.pow(m1,2)+
                  Math.pow(distMax,2)*Math.pow(m2,2)*Math.pow(m3,2)),(1.0/2)))/
                  ((1+Math.pow(m1,2))*(m1-m2));
  return intercepts;
}

/**
 * Get the bisector lying on the right and left sides of the
 * lower and upper envelopes respectively
 * @param l1
 * @param l2
 * @return
 */
public static Line boundingBoxBisector(Line l1, Line l2) {
  Line l1l2bisector = lineBetween(l1,l2);
  if(l2.getAngle() > l1.getAngle()) {
    if(l2.getAngle() - l1.getAngle() <= (Math.PI/2))
      l1l2bisector.setAngle(l1l2bisector.getAngle()+(Math.PI/2));
  }
  else {
    if(l2.getAngle() + Math.PI - l1.getAngle() < (Math.PI/2))
      l1l2bisector.setAngle(l1l2bisector.getAngle()+(Math.PI/2));
  }
  return l1l2bisector;
}

/**
 * Set our global bounding box variables to the ones
 * we compute
```

```
     * @param bounding
     */
    public static void setVDBounding(DimensionD bounding) {
      right_bound = bounding.right;
      left_bound = bounding.left;
      top_bound = bounding.top;
      bottom_bound = bounding.bottom;

      top_right = new PointD(right_bound, top_bound);
      top_left = new PointD(left_bound, top_bound);
      bottom_right = new PointD(right_bound, bottom_bound);
      bottom_left = new PointD(left_bound, bottom_bound);
    }

    /**
     * Returns true if all the lines intersect at the same point
     * allowing for double imprecision using an externally defined
     * fudge factor
     * @param lines
     * @return
     */
    public static boolean isSingleIntersection(List<Line> lines) {
      if(lines.size() < 2)
        return true;
      PointD first = intersection(lines.get(0), lines.get(lines.size()-1));
      boolean allSame = true;
      for(int i = 0; i < lines.size()-1; i++) {
        PointD inter_point = intersection(lines.get(i), lines.get(i+1));
        if(inter_point == null)
          return false;
        boolean close = distance(inter_point, first) < FUDGE;
        allSame = allSame && close;
      }
      return allSame;
    }

    /**
     * Remove lines which are duplicates of one another
     * @param lines
     */
    public static void removeDups(List<Line> lines) {
      int i = 0;
      while(i < lines.size()-1) {

        if(lines.get(i).equals(lines.get(i+1))) {
          lines.remove(i);
        }
        else
          i++;
      }
    }
}
```

```
/**
 * @author freefal
 * This class represents a rectangular
 * area in R^2.  It is used to pass
 * bounding box info btw. methods
 */
public class DimensionD {
  public boolean initialized = false;
  public double top;
  public double bottom;
  public double left;
  public double right;

  public DimensionD() {

  }
  public DimensionD(PointD p) {
    top = bottom = p.y;
    left = right = p.x;
    initialized = true;
  }
  public void includePoint(PointD p) {
    if(initialized) {
      top = Math.max(top, p.y);
      bottom = Math.min(bottom, p.y);
      left = Math.min(left, p.x);
      right = Math.max(right, p.x);
    }
    else {
      top = bottom = p.y;
      left = right = p.x;
      initialized = true;
    }
  }
  public String toString() {
    return "{" + top + ", " + bottom + ", " + left + ", " + right + "}";
  }
}
```

```
/**
 * @author Mark Henle
 * This class represents a face in
 * a Voronoi diagram represented as
 * a double connected edge list
 */

public class Face {
  // Arbitrary half-edge on this face
  private HalfEdge edge;

  public HalfEdge getEdge() {
    return edge;
  }

  public void setEdge(HalfEdge edge) {
    this.edge = edge;
  }

  public String toString() {
    return toString(true);
  }

  public String toString(boolean forward) {
    StringBuffer strBuf = new StringBuffer();
    HalfEdge start = getEdge();
    HalfEdge curEdge = start;
    strBuf.append(curEdge.getOrig().getPoint().toString());
    curEdge = forward ? curEdge.getNext() : curEdge.getPrev();
    do {
      strBuf.append(", ");
      strBuf.append(curEdge.getOrig().getPoint().toString());
    } while ((forward ? (curEdge = curEdge.getNext()) : (curEdge = curEdge.getPrev())) != start);
    return strBuf.toString();
  }

}
```

```
/**
 * @author Mark Henle
 * This class represents a directed edge in a
 * a double connected edge list
 */
public class HalfEdge {
  protected Vertex orig; // first point
  protected HalfEdge prev; // previous edge in current edge list
  protected HalfEdge next; // next edge in current edge list
  protected HalfEdge twin; // edge in opposite direction (on the other side)
  protected Face face; // current list (this edge's list)

  public HalfEdge() {

  }

  public HalfEdge(Vertex orig, HalfEdge prev, HalfEdge next, HalfEdge twin, Face face) {
    this.orig = orig;
    this.prev = prev;
    this.next = next;
    this.twin = twin;
    this.face = face;
  }
  public Face getFace() {
    return face;
  }
  public void setFace(Face face) {
    this.face = face;
  }
  public HalfEdge getNext() {
    return next;
  }
  public void setNext(HalfEdge next) {
    this.next = next;
  }
  public Vertex getOrig() {
    return orig;
  }
  public void setOrig(Vertex orig) {
    this.orig = orig;
  }
  public HalfEdge getPrev() {
    return prev;
  }
  public void setPrev(HalfEdge prev) {
    this.prev = prev;
  }
  public HalfEdge getTwin() {
    return twin;
  }
  public void setTwin(HalfEdge twin) {
    this.twin = twin;
  }
  public boolean equals(Object o) {
    HalfEdge e = (HalfEdge)o;
    return e.getOrig().equals(getOrig()) && e.getNext().getOrig().equals(getNext().getOrig());
  }

  public String toString() {
    if(next == null) {
      //System.outprintln(this.prev);
    }
    return "(" + orig.x + ", " + orig.y + ") (" + next.orig.x + ", " + next.orig.y + ")";
  }
}
```

```
/**
 * @author Mark Henle
 * Represents a line that passes through
 * the given point with the given angle
 */

public class Line {
  // Allow direct access to these for sanity
  public double x;
  public double y;
  private double angle; // in radians
  // We keep around only the 2 extreme parallel lines
  // This allows us to get from one to the other if
  // we need the other line for its face
  private Line parallelPartner;

  public Line(double x, double y, double angle) {
    this.x = x;
    this.y = y;
    setAngle(angle);
  }

  public Line(PointD p, double angle) {
    this.x = p.x;
    this.y = p.y;
    setAngle(angle);
  }

  public double getAngle() {
    return angle;
  }

  /**
   * Take modulus pi since a 0 and pi angle
   * line are identical and having two representations
   * makes our life more difficult
   * @param angle
   */
  public void setAngle(double angle) {
    if(angle < 0)
      angle = angle + ((int)(-angle/Math.PI)+1)*Math.PI;
    angle = angle % Math.PI;
    this.angle = angle;
  }

  public PointD getPoint() {
    return new PointD(x, y);
  }

  public void setPoint(PointD p) {
    this.x = p.x;
    this.y = p.y;
  }

  public Line copy() {
    Line line = new Line(x, y, angle);
    return line;
  }


  /**
   * Conver to slope-intercept representation
   * since this is unique and then compare
   * @param o
   */

  public boolean equals(Object o) {
    Line l = (Line)o;
    if(l.getAngle() != getAngle())
      return false;
```

```
    if(l.getAngle() == Math.PI/2)
      return l.x == x;

    double m1 = Math.tan(l.getAngle());
    double m2 = Math.tan(getAngle());

    double b1 = l.y - l.x*m1;
    double b2 = y - x*m2;

    return Worker.fuzzyEqual(m1, m2) && Worker.fuzzyEqual(b1, b2);
  }

  public String toString() {
    return "Line: (" + x + ", " + y + ", " + angle + ")";
  }

  public Line getParallelPartner() {
    return parallelPartner;
  }

  public void setParallelPartner(Line parallelPartner) {
    this.parallelPartner = parallelPartner;
  }
}
```

```
import java.util.Comparator;

/**
 * @author Mark Henle
 * This class implements our ordering of lines
 * by their slopes
 */
public class LineComparator implements Comparator {

  public int compare(Object o1, Object o2) {
    Line l1 = (Line)o1;
    Line l2 = (Line)o2;
    double diff = l1.getAngle() - l2.getAngle();
    if(diff > 0)
      return 1;
    if(diff < 0)
      return -1;
    return 0;
  }

}
```

```
import java.io.*;
import java.util.*;

/**
 * @author Mark Henle
 * This class parses an input file
 * It takes line starting with % as commented out
 * and takes '-' as a cue to stop reading the file
 */
public class Parser {
  public static ArrayList<Line> parse (String fileName) {
    ArrayList<Line> lines = new ArrayList<Line>();
    try {
      BufferedReader reader = new BufferedReader(new FileReader(fileName));
      String lineOfText;
      while ((lineOfText = reader.readLine()) != null) {
        // Commented out
        if(lineOfText.charAt(0) == '%')
          continue;
        if(lineOfText.charAt(0) == '-')
          break;
        String[] pieces = lineOfText.split(",");
        double x = Double.parseDouble(pieces[0]);
        double y = Double.parseDouble(pieces[1]);
        double angle = Double.parseDouble(pieces[2]);
        Line line = new Line(x, y, angle);
        lines.add(line);
      }

    } catch(Exception e) {
      System.err.println("Unable to correctly read input");
      e.printStackTrace();
    }
    return lines;
  }
}
```

```
/**
 * @author Mark Henle
 * This class represents a 2D point
 * with double coordinates
 */
public class PointD {
  public double x;
  public double y;

  public PointD(double x, double y) {
    this.x = x;
    this.y = y;
  }

  public boolean equals (PointD p) {
    return x == p.x && y == p.y;
  }

  public String toString() {
    return "(" + x + ", " + y + ")";
  }
}
```

```
/**
 * @author Mark Henle
 * This class represents a vertex in
 * a Voronoi diagram represented as
 * a double connected edge list
 */

public class Vertex {
  // Allow direct access to these for sanity
  public double x;
  public double y;
  private HalfEdge incident; // An arbitrary incident half-edge

  public Vertex(double x, double y, HalfEdge incident) {
    this.x = x;
    this.y = y;
    this.incident = incident;
  }

  public Vertex(PointD p) {
    this.x = p.x;
    this.y = p.y;
  }

  public Vertex(PointD p, HalfEdge incident) {
    this.x = p.x;
    this.y = p.y;
    this.incident = incident;
  }

  public HalfEdge getIncident() {
    return incident;
  }
  public void setIncident(HalfEdge incident) {
    this.incident = incident;
  }
  public PointD getPoint() {
    return new PointD(x, y);
  }
  public boolean equals(Object o) {
    Vertex v = (Vertex)o;
    return v.x == x && v.y == y;
  }

}
```

```
import java.util.*;

/**
 * @author Mark Henle
 * A VD is a mapping between the lines whose VD
 * we are finding and their associated faces
 */
public class VoronoiDiagram {
  // List of lines sorted by angle
  List<Line> lines;

   // Mapping between each line and its two associated faces
  HashMap<Line, Face[]> lineToFace;

  // Mapping between face and it's associated line
  HashMap<Face, Line> faceToLine;

  public VoronoiDiagram() {
    lines = new ArrayList<Line>();
    lineToFace = new HashMap<Line, Face[]>();
    faceToLine = new HashMap<Face, Line>();
  }

  public void add(Line line, Face[] faces) {
    lineToFace.put(line, faces);
    faceToLine.put(faces[0], line);
    if(faces.length > 1)
      faceToLine.put(faces[1], line);
  }

  /**
   * Get line associated with the given face
   * @param f
   * @return
   */
  public Line getFaceLine(Face f) {
    return faceToLine.get(f);
  }

  /**
   * Get faces associated with the given line
   * @param line
   * @return
   */
  public Face[] getLineFaces(Line line) {
    return lineToFace.get(line);
  }

  public HashMap<Face, Line> getFaceToLine() {
    return faceToLine;
  }

  public void setFaceToLine(HashMap<Face, Line> faceToLine) {
    this.faceToLine = faceToLine;
  }

  public HashMap<Line, Face[]> getLineToFace() {
    return lineToFace;
  }

  public void setLineToFace(HashMap<Line, Face[]> lineToFace) {
    this.lineToFace = lineToFace;
  }

  public List<Line> getLines() {
    return lines;
  }

  public void setLines(List<Line> lines) {
    this.lines = lines;
```

```
  }
  public String toString() {
    String retString = "";
    int counter = 1;
    for(Line l : lines) {
      retString += "Line " + counter++ + ": " + l.toString() + "\n";
      retString += "Face 1: " + lineToFace.get(l)[0] + "\n";
      if(lineToFace.get(l).length > 1)
        retString += "Face 2: " + lineToFace.get(l)[1] + "\n";
    }
    return retString;
  }
}
```

```
import java.awt.*;
import javax.swing.*;
import java.util.*;

/**
 * @author Mark Henle
 * This class outputs a Voronoi diagram in a rudimentary
 * way without scaling.  It it designed for debugging purposes
 */

public class DisplayVD extends Canvas {
  public static int WIDTH = 400;
  public static int HEIGHT = 400;
  public static int THICKNESS = 5; // Thickness of VD borders
  VoronoiDiagram vd;
  // Color of each line and it's associated faces
  HashMap<Line, Color> lineColors = new HashMap<Line, Color>();

  public static void display(VoronoiDiagram vd, String name) {
    JFrame frame = new JFrame();
    frame.setTitle(name);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    DisplayVD canvas = new DisplayVD();
    canvas.setVd(vd);
    canvas.setSize(new Dimension(WIDTH,HEIGHT));
    frame.add(canvas);
    frame.pack();
    frame.setVisible(true);
  }

  public void paint(Graphics g) {
    super.paint(g);
    Graphics2D g2d = (Graphics2D)g;
    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
    java.util.List<Line> lines = vd.getLines();

    for(Line line : lines) {
      Color randomColor = lineColors.get(line);
      if(randomColor == null) {
        randomColor = new Color((int)(((line.getAngle()+1)*(line.x+100)*
                                (line.y+100)*200000)%Math.pow(2, 24)));
        lineColors.put(line, randomColor);
      }

      g.setColor(randomColor);

      Face[] faces = vd.getLineFaces(line);
      if(faces == null)
        System.out.println(line);
      for(Face face : faces) {
        HalfEdge start = face.getEdge();
        HalfEdge curEdge = start;
        Polygon poly = new Polygon();
        // Iterate around the face counter-clockwise
        // until we return to the start edge
        do {
          Vertex v = curEdge.getOrig();
          poly.addPoint((int)v.x, (int)v.y);
        } while ((curEdge = curEdge.getNext()) != start);

        g.fillPolygon(poly);
      }
    }
    for(Line line : lines) {
      /*
      if(line.getAngle() != 0.2686061718819273)
        continue;
      */
```

```
      double x_slope = Math.cos(line.getAngle());
      double y_slope = Math.sin(line.getAngle());
      // either sin or cos > 1/2 so doubling the size
      // of the screen guarantees we leave it
      int x1 = (int)(line.x + x_slope*2*WIDTH);
      int y1 = (int)(line.y + y_slope*2*HEIGHT);
      int x2 = (int)(line.x - x_slope*2*WIDTH);
      int y2 = (int)(line.y - y_slope*2*HEIGHT);

      g.setColor(lineColors.get(line));
      g.drawLine(x1, y1, x2, y2);
    }

  }

  public VoronoiDiagram getVd() {
    return vd;
  }

  public void setVd(VoronoiDiagram vd) {
    this.vd = vd;
  }
}
```