# Topological Sweep in Degenerate Cases

Eynat Rafalin[1], Diane Souvaine[1], and Ileana Streinu[2]

[1] Department of Electrical Engineering and Computer Science, Tufts University, Medford, MA 02155. [***] {erafalin, dls}@eecs.tufts.edu
[2] Department of Computer Science, Smith College, Northampton, MA 01063. [†] streinu@cs.smith.edu

**Abstract.** Topological sweep can contribute to efficient implementations of various algorithms for data analysis. Real data, however, has degeneracies. The modification of the topological sweep algorithm presented here handles degenerate cases such as parallel or multiply concurrent lines without requiring numerical perturbations to achieve general position. Our method maintains the $O(n^2)$ and $O(n)$ time and space complexities of the original algorithm, and is robust and easy to implement. We present experimental results.

## 1 Introduction

Dealing with degenerate data is a notoriously untreated problem one has to face when implementing Computational Geometry algorithms. Most of the theoretical developments have avoided the special cases or proposed too general solutions that may produce other side-effects. One of the few papers that actively deals with degenerate cases is Burnikel *et al.* [2]. The authors argue forcefully that perturbation is not always effective in practice and that it is simpler (in terms of programming effort) and more efficient (in terms of running time) to deal directly with degenerate inputs. Their paper presents two implementations for solving basic problems in computational geometry. Note that the running time of their implementation can be sensitive to the amount of degeneracy.

The work in this paper is motivated by the practical need to have a robust implementation of topological sweep to be used within implementations of several geometric algorithms for computing statistical measures of data depth, both those already coded (e.g. [9]) and those currently in development. The implementation is general enough to be used in place of any topological sweep subroutine in existing code.

The *topological sweep* method of Edelsbrunner and Guibas [5] is one of the classical algorithms in Computational Geometry. It sweeps an arrangement of $n$ planar lines in $O(n^2)$ time and $O(n)$ space with a topological line and is a critical ingredient in several space and time efficient algorithms (e.g. [12], [6], [9], [7]). The technique has been adapted for specific applications (e.g. [10]) and has a

[***] Partially supported by NSF grant EIA-99-96237
[†] Partially supported by NSF RUI grant 9731804

useful variant ([1]). The algorithm and its variations have been implemented by several groups (e.g. [11], [9]...).

Computational Geometry libraries such as LEDA and CGAL offer implementations of related but slightly less efficient line sweep algorithms. To the best of our knowledge, no robust code dealing with all degeneracies is currently available.

In contrast, the method proposed here is simple to compute and does not require special preprocessing. The modified algorithm was coded and the code was verified on different types and sizes of data sets. The code was incorporated in an implementation of an algorithm for statistical data analysis.

## 2 The Topological Sweep Algorithm [5]

Let $H$ be a arrangement of $n$ lines in the plane. Vertical line sweep could report all intersection pairs sorted in order of x-coordinate in $\Theta(n^2 \log n)$ time and $O(n)$ space (e.g. [3], [4]). If one only needs to report the intersection points of the lines according to a partial order related to the levels in the arrangement, greater efficiency is possible. To report all the intersection points of the lines, in quadratic time and linear space, we use a topological line (*cut*) that sweeps the arrangement. A topological line is a monotonic line in $y$-direction, which intersects each of the $n$ lines in the arrangement exactly once. The cut is specified by the sequence of edges, one per line, each intersected by the topological line. A sweep is implemented by starting with the leftmost cut which includes all semi-infinite edges ending at -$\infty$, and pushing it to the right until it becomes the rightmost cut, in a series of elementary steps. An elementary step is performed when the topological line sweeps past a vertex of the arrangement. To keep the sweep line a topological line we can only sweep past a vertex which is the intersection point of two consecutive edges in the current cut (a *ready* vertex). See Fig. 1 for an example of an arrangement of 7 lines and a topological cut.

### 2.1 Data Structures

The algorithm uses the following data structures:

- E[1:n] is the array of line equations. E[i] = $(a_i,\ b_i)$ if the $i^{th}$ line of the arrangement sorted by slope $a_i$ is $y = a_i x + b_i$. This array is static.
- HTU[1:n] is an array representing the upper horizon tree. HTU[i] is a pair $(\lambda_i,\ \rho_i)$ of indices indicating the current lines that delimit the segment of $l_i$ in the upper horizon tree to the left and the right respectively. If the segment is the leftmost on $l_i$ then $\lambda_i = -1$. If it is the right most then $\rho_i = 0$.
- HTL[1:n] represents the lower horizon tree and is defined similarly.
- I is a set of integers represented as a stack that correspond to points that are currently *ready* to be processed. If $i$ is in I then $c_i$ and $c_{i+1}$ (the $i^{th}$ and $i^{th}+1$ lines of the cut) share a common right endpoint and represent a legal next move for the topological line.

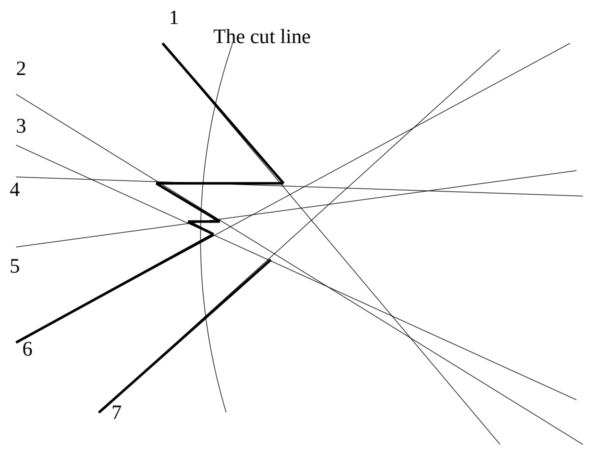

**Fig. 1.** An arrangement of 7 lines and a cut defined by its edges. The cut edges are marked as bold lines

- M[1:n] is an array holding the current sequence of indices from $E[i]$ that form the lines $m_1$, $m_2..m_n$ of the cut.
- N[1:n] is a list of pairs of indices indicating the lines delimiting each edge of the cut, one from the left and the other from the right.

## 2.2 Horizon Trees and the Construction of the Cut

The upper (respectively lower) horizon tree of the cut is constructed by extending the cut edges to the right. When two edges intersect only the one of higher (respectively lower) slope continues to the right (see Fig. 2 for an example of upper and lower horizon trees). The upper (lower) horizon tree is initially created by inserting the lines in decreasing (increasing) order of slope into the structure. To insert line $l_{k+1}$ we begin at its endpoint on the left boundary. We walk in counterclockwise order around the bay formed by the previous lines to find the intersection point of $l_{k+1}$ with an edge. Given the lower and upper horizon trees, the right endpoint of the cut is identified by the leftmost of the two right delimiters of HTU[i] and HTL[i].

**Time Complexity** Each elementary step and the accompanying updates to the upper and lower horizon trees, the cut arrays, and the ready stack take amortized constant time. There are at most $n^2$ elementary steps and therefore the total time complexity is $O(n^2)$. The original paper [5] contains a detailed analysis.

## 2.3 Dealing with degeneracies

The original paper proposes dealing with degenerate cases such as parallel lines or multiple concurrent lines by using a primitive procedure to treat two parallel

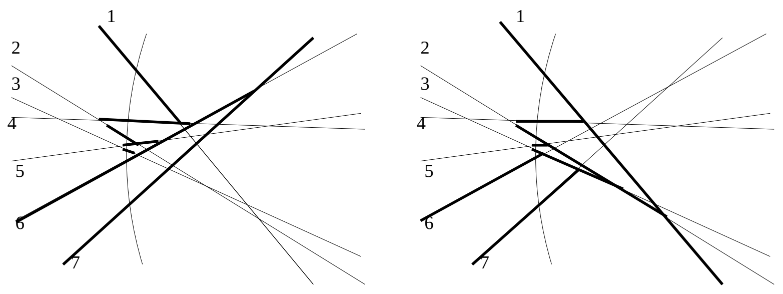

**Fig. 2.** Upper and Lower Horizon Trees

**Table 1.** Upper and Lower Horizon Trees

| | Upper Horizon tree | | Lower Horizon tree | |
|---|---|---|---|---|
| Line | Left delimiter | Right delimiter | Left delimiter | Right delimiter |
| 1 | -∞ | 4 | -∞ | ∞ |
| 2 | 4 | 5 | 4 | 1 |
| 3 | 5 | 6 | 5 | 2 |
| 4 | 2 | 6 | 2 | 1 |
| 5 | 3 | 6 | 3 | 2 |
| 6 | -∞ | 7 | -∞ | 3 |
| 7 | -∞ | ∞ | -∞ | 3 |

lines as non-parallel and three concurrent lines as non-concurrent ([5], [4]). This original method generates zero length edges and vertices at infinity and then performs topological sweep on an arrangement in general position that is $\epsilon$ different from the original arrangement. The original method demands the computation of the power series expansion (in the perturbation parameter $\epsilon$) of the value of a determinant until the first non-zero term is encountered (see also Gomez *et al* [8]). Edelsbrunner and Guibas also propose to detect degeneracies in configurations in $O(n^2)$ time and $O(n)$ space by checking for edges of length zero.

## 3 The Modified Algorithm

### 3.1 Data Structures

The modified algorithm uses many of the same data structures as in the original algorithm: $E[]$ holds the line equations; $M[]$, the order of the lines along the cut; $HTL[]$ and $HTU[]$, the horizon trees. The cut data structure $N[]$ and the entries in the ready stack $I$, however, each have an additional field. $N[i]$ is now a triplet ($\lambda_i$, $r_{up,i}$, $r_{down,i}$) of indices. If the right endpoint of $N[i]$ is generated by its intersection with a line from above (resp. from below), then $r_{up,i}$ (resp. $r_{down,i}$) is the index of that line; otherwise $r_{up,i}$ (resp. $r_{down,i}$) is null. At least one is non-null, but both may be in instances where the right endpoint of the cut edge is the intersection point of three or more lines. The left delimiter $\lambda_i$ can be any line that intersects the current edge at its left endpoint. Given the lower and upper horizon trees, these delimiters are computed in constant time.

Each entry in the augmented stack $I$ of ready vertices is now a pair of integers $(i, k)$ where the segments of the lines $m_i$, $m_{i+1}$... $m_{i+k}$ (the $i^{th}$ until $(i + k)^{th}$ edges of the cut) share a common right endpoint and represent a legal next move. These modifications do not change the asymptotic space complexity of the algorithm.

One new data structure is needed. $MATCH[i]$ is a pair of indices pointing to the uppermost and lowermost cut edges that currently share the same right end point as line $l_i$. $MATCH[i]$ is initialized at the beginning of the algorithm and reset to the pair (i, i) every time $l_i$ participates in an elementary step, meaning that the match to line $l_i$ is trivially line $l_i$ itself. $MATCH$ is updated at the conclusion of each elementary step to detect new alignment of right end points. The fact that the edges along the cut sharing the same right endpoint form at most one connected component of adjacent edges (see Lemma 1 below) allows us to update only two boundary edges (top and bottom of the edges found so far) and ignore the intervening entries.

## 3.2 Computing Ready Vertices

Define a *matching pair* as a pair of consecutive lines $l_i, l_j$ in the cut where $r_{up,i}$ is $j$ and $r_{down,j}$ is $i$. When at most two lines participate in an intersection, a *matching pair* implies a *ready vertex*. For the more general case, we define a *matching sequence* of consecutive lines $l_i, ..., l_j$ in the cut where every adjacent pair of lines forms a *matching pair*. A *ready vertex* is generated by a *complete matching sequence* where the bottom line is $l_i$ and the top line is $l_j$ and in which $r_{down,i}$ and $r_{up,j}$ are both *null*.

After each update to the cut, we test whether two newly adjacent lines form a matching pair. If so, this new pair either augments a *matching sequence* already represented in $MATCH[]$ or initializes a new one. Updating $MATCH[]$ and checking whether the *matching sequence* is not *complete* takes constant time (see Lemma 3 below).

## 3.3 Parallel Lines and Identical Lines

Parallel lines create an intersection point at infinity. Their intersection point is not treated differently than that of any other line. Identical lines are lines that have the same slope and $y$-intercept. We currently treat identical lines together as a single line, under the assumption that the application that calls topological sweep will note and handle the impact of duplicate lines. For example, our application code, that computes depth contours based on the levels of the arrangement, handles this phenomenon (see [9]).

### 3.4 Additional Changes to the Algorithm

Tests called 'above' and 'closer' are used when the data structures are constructed and updated. Each comparison has three possible outcomes (instead of two): TRUE, FALSE and EQUAL. The EQUAL state is not part of the original algorithm since it is only generated in degenerate cases. (see 6). To deal with parallel lines, special test cases were added, that check if the lines in question are parallel or not. The tests are peformed during the initialization phase of the algorithm

Each update step may involve two or more updates, depending on the size of the intersection. The different data structures demand different update strategies. One update method is to replace each of the lines from $i$ to $j$ in a matching sequence. Lines are paired from the outermost lines to the innermost ($i$ to $j$, $i+1$ to $j-1$, etc). This procedure is used to update the left delimiters of the cut and the upper and lower horizon trees and to update the order of the lines along the cut (M). Another method of updating is by computing the updated lines one by one. This method is used to update the right delimiters of the horizon trees and the cut. The horizon trees must be processed consecutively, otherwise the cut will not be updated correctly.

## 4 Example of a Degenerate Case

For the arrangement of 5 lines depicted in Fig. 3, the information in Table 4, describing the current cut, is computed from the upper and lower horizon trees. For each line $l_i$, the right delimiter of the lower horizon tree initially becomes $r_{up,i}$ and the right delimiter of the upper horizon tree initially becomes $r_{down,i}$. In the next step each pair of delimiters is compared to see which implied intersection point is *closer*: the delimiters for line 1 are $\infty$ and 5, therefore only 5 remains and the right-up delimiter of line 1 becomes *null*. Line 2 and 5 are similar to line 1 since one horizon tree delimiter is $\infty$ and it becomes *null*. The delimiters of lines 3 and 4 intersect in the same point and hence both remain.

In the *matching procedure* we only look for matches for the edges that were created after processing the last ready point (the intersection between lines 1 and 4). We start by looking for a match for line 1. Since right-down of line 1 is 5 and right-up of line 5 is 1, we have a matching pair. Since both the other delimiters of lines 1 and 5 are *null*, we will have a *complete matching sequence* and must update the stack $I$ with the pair $(4,1)$. We now consider line 4. Since right-up of line 4 is 3 and right-down of line 3 is 4 we have a match. We continue to look for an existing matching sequence with line 3. Our $MATCH$ data structure indicates a matching sequence starting with line 3 and ending with line 2 (consisting of just one pair where right-down of line 2 is 3 and right-up of line 3 is 2) which we expand to a matching sequence starting at line 4 and ending with line 2: $MATCH[4]$ and $MATCH[2]$ are both the pair $(2,4)$. Note that we do not bother to update $MATCH[3]$. The right-up delimiter of 2 is null so we do not need to check for additional matches above. But the right-down of line 4 is still unmatched as right-down of 4 is 5 and right-up of line 5 is *null*. The matching sequence remains *incomplete*.

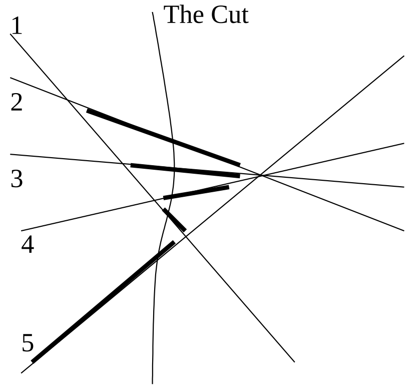

**Fig. 3.** Example of a degenerate case

| **Line i** | $\lambda_i$ | $\mathbf{r_{up,i}}$ | $\mathbf{r_{down,i}}$ |
|---|---|---|---|
| 1 | 4 | *null* | 5 |
| 2 | 1 | *null* | 3 |
| 3 | 1 | 2 | 4 |
| 4 | 1 | 3 | 5 |
| 5 | $-\infty$ | 1 | *null* |

## 5 Performance Analysis

**Lemma 1.** *The set of edges along any cut that contain the same intersection point as their right endpoint form at most one connected component.*

*Proof.* Assume the edges that contain $p$ as their right endpoint form more than one connected component (Fig. 4). Let the edges be $p_1, p_2, ...$ and the associated lines be $l_{p_1}, l_{p_2}, ....$ Since there is more than one connected component there exist at least one line $l_k$ between $p_i$ and $p_{i+1}$ that contains $p$ but whose current cut edge $c_k$ does not have $p$ as a right endpoint. Assume the right end-point $q$ of $c_k$ is delimited by a line $l$. If the slope of $l$ is smaller (resp. larger) than that of $l_k$, then $l$ intersects $l_i$ (resp. $l_{i+1}$) at a point $r$ between points $p$ and $q$. Since point $q$ has not yet been processed, point $r$ is not yet ready and has not been processed, and therefore the edge $p_i$ (resp. $p_{i+1}$) cannot yet be part of the cut. Contradiction. □

**Lemma 2.** *The total cost of updating HTU (HTL) through all the elementary steps is* $O(n^2)$. See original paper [5].

**Lemma 3.** *The total cost of comparing adjacent edges (computing the ready points) through all the elementary steps is* $O(n^2)$

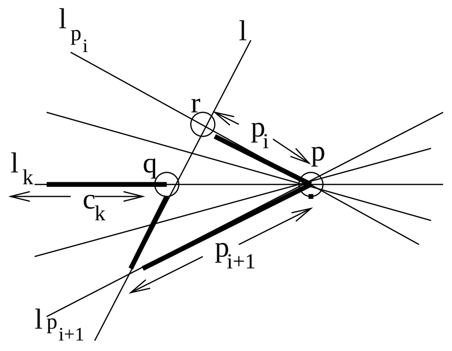

**Fig. 4.** Proof of Lemma 1

*Proof.* Each edge, $c$, that terminates at an intersection can generate at most 3 comparison tests. First a matching test is performed with the edges above and below. If a matching edge $m$ is found, the $MATCH[m]$ will also be tested. According to Lemma 1 at most one of the edges that delimits $c$ above and below can have a MATCH, otherwise more than one connected component exists. Therefore at most 3 tests will be generated. There is no need to perform additional tests since if another match exists it should have been found earlier, when the edges that form it were investigated. The total number of edges that enter all the intersections is $n^2$ and therefore the total complexity is at most $O(n^2)$. □

**Lemma 4.** *The time complexity of the algorithm is* $O(n^2)$.

*Proof.* Initialization of all data structures takes linear time after $O(n \log n)$ time to sort the lines by slope. Each elementary step takes an amortized constant time, as shown in Lemmas 2 and 3. There are $O(n^2)$ elementary steps. The total time complexity is therefore $O(n^2)$. □

## 6 Discussion

To ensure the correctness of the method, we verify that rounding errors do not produce irrecoverable mistakes.

One test compares slopes to compute whether two lines are parallel. The answer (*parallel* or *not*) will be as good as the data set provided: no round-off errors occur since there is no computation involved.

Other tests compare computed values of $x$ or $y$ coordinate to check if a point / intersection of the arrangement is *above* or *closer* than another point / intersection and return one of *true*, $false$ or *equal*. These computations involve multiplications of at most two values. When the input values are floats, using double precision to compare the computed values ensures that only errors that return an *equal* answer instead of *true* or $false$ can occur. No answer can have the opposite value to the real value (e.g. *true* instead of $false$ or $true/false$

instead of *equal*). When receiving an answer that is *equal* instead of *not-equal*, three or more lines of the arrangement will be treated as passing through the same point although they do not. Effectively a triangular face is contracted to a point but otherwise there is no change in the topological structure of the arrangement.

## 7 Experimental Results

### 7.1 Analysis Method

Our experiments checked the behavior of the code in simple and in degenerate cases. We created sets of lines by generating $n$ random numbers representing the slopes of the lines and $n$ random numbers representing the $y$-intercept of the lines. We paired the numbers to receive a representation of an arrangement of $n$ lines.

At the end of a correct sweep all the right delimiters must be $\infty$. If a mistake occurs the new topology will not allow the sweep to continue until the rightmost point and will stop too early, not reaching $\infty$. Hence, a check that all the right delimiters are $\infty$ is made to verify that the sweep was performed correctly.

To generate parallel lines we chose $m$ numbers out of the $n$ slopes generated above and $m$ numbers out of the $n$ $y$-intercepts generated above. We paired these at random where $m$ is 5 percent of $n$, and used the $n+m$ pairs as our data set. This ensured that at least $m$ pairs have the same slopes and are therefore parallel or coincident.

To generate multiple concurrent lines we computed the $n^2$ intersection points of the original arrangement we generated. We than randomly chose $n$ points out of the $n^2$ intersections and used their dual lines as our data set. The dual line of point $(a, b)$ is the line $y = ax + b$. If more than two points are on the same line their dual lines share a common point and we get a degenerate case of multiple concurrent lines. By selecting $n$ intersection points from the $n^2$ intersections of the original set and taking their dual, the probability of choosing more than one point on the same line, hence creating an intersection of multiple lines in the dual, is relatively high.

### 7.2 Results

Our code is written in C++, does not use any geometric libraries for computations, but uses GEOMVIEW for visualization of the output. The code is built modularly and can be easily modified. It is located at:

http://www.eecs.tufts.edu/r/geometry/sweep.

It was tested on a Sun Microsystems Ultra 250 SPARC processor, 400 MHz, and was compiled using the GNU C++ compiler. We ran our code on 10 different data sets of each size and type that were generated as described above. The average results are presented in Fig. 5.

The implementation is not assumed to perform better than other topological sweep implementations in terms of running time. Instead it treats cases that other implementations were slow or unable to handle. Our package creates data that is used for display purposes. If display is not needed the computation can be streamlined further.

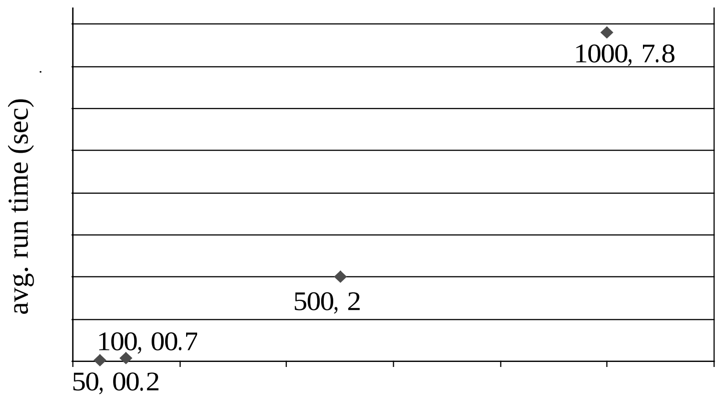

**Fig. 5.** Measured execution times (in seconds) for different set types

## 8 Future Research

### 8.1 Guided Topological Sweep

Guided topological sweep is a topological sweep that maintains additional order criteria without penalizing performance. For example, it is possible to guarantee that whenever a vertex is swept, the edges that are $k$ lines below it and $k$ lines above it are aligned with it without changing asymptotic complexity. This method is used for LMS regression in $O(n^2)$ time (see [6]). We plan to expand our algorithm to enable this type of sweep.

### 8.2 Topological Sweep in Higher Dimensions

Most statistical (and other) data sets are multi-dimensional. There are some theoretical algorithms for high dimensions but few of them have been implemented. We are working to expand our implementation to higher dimensions and use this as a sub-procedure for high-dimensional applications.

### 8.3 Applications

Degenerate data sets that include more than one point with the same x-coordinate or concurrent points form a large part of the available (and interesting to investigate) sets. An earlier implementation of the depth contours algorithm ([9]) has been reused and expanded. By calling the new topological sweep procedure it can now handle degenerate data sets.

## 9 Conclusion

We present an efficient application of the topological sweep algorithm that uses extended data structures instead of numerical methods to deal with degenerate case. The new data structures use less than 1.5 times as much space as the original data structures.

## References

[1] Te. Asano, Leonidas J. Guibas, and T. Tokuyama. Walking on an arrangement topologically. *Internat. J. Comput. Geom. Appl.*, 4:123–151, 1994.

[2] Christoph Burnikel, Kurt Mehlhorn, and Stefan Schirra. On degeneracy in geometric computations. In Daniel D. Sleator, editor, *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 16–23, Arlington, VA, January 1994. ACM Press.

[3] Mark de Berg, Mark van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry Algorithms and Applications.* Springer-Verlag, Berlin Heidelberg, 1997.

[4] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*, volume 10 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Heidelberg, West Germany, 1987.

[5] H. Edelsbrunner and Leonidas J. Guibas. Topologically sweeping an arrangement. *J. Comput. Syst. Sci.*, 38:165–194, 1989. Corrigendum in 42 (1991), 249–251.

[6] H. Edelsbrunner and D. L. Souvaine. Computing median-of-squares regression lines and guided topological sweep. *J. Amer. Statist. Assoc.*, 85:115–119, 1990.

[7] J. Gil, W. Steiger, and A. Wigderson. Geometric medians. *Discrete Mathematics*, 108:37–51, 1992.

[8] F. Gomez, S. Ramaswami, and G. Toussaint. On removing non-degeneracy assumptions in computational geometry. In *Algorithms and Complexity (Proc. CIAC' 97)*, volume 1203 of *Lecture Notes Comput. Sci.*, pages 86–99. Springer-Verlag, 1997.

[9] K. Miller, S. Ramaswami, P. Rousseeuw, T. Sellares, D. Souvaine, I. Streinu, and A. Struyf. Fast implementation of depth contours using topological sweep. In *Proceedings of the Twelfth ACM-SIAM Symposium on Discrete Algorithms*, pages 690–699, Washington, DC, January 2001.

[10] M. Pocchiola and G. Vegter. Topologically sweeping visibility complexes via pseudo-triangulations. *Discrete Comput. Geom.*, 16:419–453, December 1996.

[11] H. Rosenberger. Order $k$ Voronoi diagrams of sites with additive weights in the plane. M.Sc. thesis, Dept. Comput. Sci., Univ. Illinois, Urbana, IL, 1988. Report UIUCDCS-R-88-1431.

[12] Emo Welzl. Constructing the visibility graph for $n$ line segments in $O(n^2)$ time. *Inform. Process. Lett.*, 20:167–171, 1985.