# Optimal Slope Selection Via Cuttings

HERVÉ BRÖNNIMANN BERNARD CHAZELLE

Department of Computer Science
Princeton University
Princeton, NJ 08544, USA

**Abstract**

We give an optimal deterministic $O(n \log n)$-time algorithm for slope selection. The algorithm borrows from the optimal solution given in [6], but avoids the complicated machinery of the AKS sorting network and parametric searching. This is achieved by redesigning and refining the $O(n \log^2 n)$-time algorithm of [4] with the help of additional approximation tools.

## 1 Optimal Slope Selection

The problem is computing the line defined by two of $n$ given points that has the median slope among all $\binom{n}{2}$ such lines. Equivalently, the problem can be stated as that of selecting the median-abscissa vertex of the arrangement $\mathcal{A}(\mathcal{L})$ of a set $\mathcal{L}$ of $n$ lines [6]. For generality, we set out to compute the vertex with rank $I^*$ from left to right, for any given $1 \leq I^* \leq \binom{n}{2}$.

An optimal deterministic solution was given by Cole *et al.* in [6], but it requires the use of the AKS sorting network [1] and parametric searching [11], and its analysis is fairly complicated. Simple randomized algorithms were subsequently designed by Dillencourt *et al.* [7] and Matoušek [10]. An optimal deterministic algorithm was given by Katz and Sharir in [8], building on the simple $O(n \log^2 n)$-time algorithm of Chazelle *et al.* [4] and using expander graphs plus an approximation tool from [6]. The solution we give here also optimizes the algorithm of [4], using only elementary data structures. The exposition is entirely self-contained, besides the construction of $\varepsilon$-nets [2, 9].

## 2 Definitions and Notation

The lines are numbered in order of decreasing slopes. To ease the exposition, we suppose that the *lines are algebraically independent, which in particular implies that vertices of their arrangement* are incident upon only two lines and have distinct abscissae. This assumption can be removed easily by a more precise treatment of degeneracies.[1] Any vertical line $x = x_0$ cuts $\mathcal{A}(\mathcal{L})$ in $n$ points (whose ordinates are called the *intercepts*), and the number of vertices on it or to its left is noted $v(\mathcal{L}, x_0)$. More generally, we let $v(\mathcal{L}, S)$ denote the number of vertices of $\mathcal{A}(\mathcal{L})$ inside *any* region $S$. The vertical ordering[2] of the lines at $x = x_0$ defines a permutation $\pi_{x_0}$ of the lines, with $\pi_{-\infty}$ being the identity. The number of inversions of this permutation is denoted $I(\pi_{x_0})$ and is exactly $v(\mathcal{L}, x_0)$.

[1] Indeed, it suffices to treat a vertex $v = l \cap l'$ as a triplet $(v, N(l), N(l'))$, where line $l$ is numbered $N(l)$. Lexicographic order disambiguates between vertices having the same abscissa.

[2] If $x = x_0$ passes through a vertex, we make the convention that the two meeting lines are in the same order than at $x = x_0 + \varepsilon$, for infinitesimally small $\varepsilon > 0$.

For our purposes, an *$\varepsilon$-net* for a set $\mathcal{L}$ of lines is a subset $N$ such that every segment intersecting more than $\varepsilon|\mathcal{L}|$ of the lines of $L$ intersects one line of $N$. Matoušek [9] gave an algorithm that computes $\varepsilon$-nets in linear time, if $\varepsilon$ is a constant. (See also [2, 5].) The only subroutine needed by this algorithm is one that computes the arrangement of a constant number of lines of $\mathcal{L}$.

Borrowing terminology from [6], let us partition a permutation $\pi$ into a collection $\mathcal{B}$ of at most $2n/m$ blocks, each containing at most $m$ consecutive lines in $\pi$; we obtain what is called an *$m$-blocked permutation* $\pi_{\mathcal{B}}$. If any two lines belonging to different blocks are ordered as in $\pi_{x_0}$, and all inversions within a block of $\pi_{\mathcal{B}}$ actually occur in $\pi_{x_0}$, we say that $\pi_{\mathcal{B}}$ is *left-compatible* with $\pi_{x_0}$. Note that $\pi_{x_0}$ might contain inversions absent from $\pi_{\mathcal{B}}$. Because at most $\binom{m}{2}$ inversions can occur within a block of size $m$, we have

$$I(\pi_{\mathcal{B}}) \leq I(\pi_{x_0}) \leq I(\pi_{\mathcal{B}}) + nm. \tag{1}$$

Reversing the order along the $x$-axis gives the concept of *right-compatibility*, and we have the converse

$$I(\pi_{\mathcal{B}}) - nm \leq I(\pi_{x_0}) \leq I(\pi_{\mathcal{B}}). \tag{2}$$

Therefore, we see that maintaining a blocked partition compatible with a permutation gives a good estimate on the number of inversions of this permutation. The rank of a vertex $(x, y)$ is $\text{rank}(x) = I(\pi_x)$, and the problem is to find the vertex $v^*$ of $\mathcal{A}(\mathcal{L})$ of abscissa $x^*$ with $\text{rank}(x^*) = I^*$.

## 3 Updating Blocked Permutations

Assume we have an $m$-blocked permutation $\pi$ left- (resp. right-) compatible with $\pi_x$ for some $x$, and we wish to have an $m$-blocked permutation $\pi'$ compatible with $\pi_{x'}$ for a given $x' > x$ (resp $x' < x$) such that $|\text{rank}(x') - \text{rank}(x)| = O(nm)$. We modify and simplify a technique called *reblocking* and used in [6]. Let us process the lines of $\mathcal{L}$ in the order given by $\pi$. Before processing line $l$, assume we have a linked list of stacks, $s_1, \ldots, s_q$, each of them containing between $\lceil m/2 \rceil$ and $m$ elements. Initially, only $s_1$ is in the list, and it is filled with the first $m$ elements of $\pi$. For each stack $s_i$, we keep two counters: the number $\text{size}(s_i)$ of elements, and the lowest intercept at $x'$ of a line of $s_i$, which we denote $\text{low}(s_i)$. We also keep a global counter $I''$, initially set to 0. This counter will serve to count the additional inversions between $\pi$ and $\pi'$.

To insert a line $l$ into our linked stacks, we first try to insert it into $s_q$, the last stack in the list. If the intercept $y(l)$ of $l$ at $x'$ is lower than the lowest intercept of $s_q$, we try to insert $l$ into $s_{q-1}$, and iterate if necessary. Each time we go down the list from $s_i$ to $s_{i-1}$, we increase $I''$ by $\text{size}(s_i)$. Should we reach $s_1$, we insert $l$ there and update $\text{low}(s_1)$.

If after inserting a line into a stack $s_i$, this stack has $m+1$ elements, we need to split it. To do so, we compute the median $y$ of all the intercepts at $x'$, $(y(l))_{l \in s_i}$, and reinsert the lines into a new stack $s_i'$ (resp. $s_i''$) according to whether their intercept is smaller than or equal to (resp. larger than) $y$. The lines are inserted in the same order as which they have been put into $s_i$. Each time a line is inserted into $s_i'$, $I''$ is incremented by the current $\text{size}(s_i'')$. Then blocks $s_{i+1}, \ldots, s_q$ are renumbered $s_{i+2}, \ldots, s_{q+1}$, and $s_i'$ (resp. $s_i''$) becomes the new $s_i$ (resp. $s_{i+1}$).

When the whole round of insertions finishes, we have a new $m$-blocked permutation $\pi'$ obtained by concatenating all the stacks in order, which by construction is compatible with $\pi_{x'}$. If $I$ is the number of inversions of $\pi$, we claim that the number of inversions of $\pi'$ is $I' = I + I''$. Note that we insert the lines in the order given by $\pi$. Inside the blocks of $\pi$, the lines need not be in the actual order corresponding to that abscissa; however, all the inversions of $\pi$ have been accounted for in $I$ (including inversions between two lines of the same block). To start with, we observe that we count an inversion between $l'$ and $l''$ in $I''$ only if $y(l') < y(l'')$ (which implies that $l'$ is before

$l''$ in $\pi$), and if $l'$ and $l''$ are assigned different blocks in $\pi'$ before or after splitting a stack. In each case, the intersection is being witnessed: $y(l') \geq \text{low}(s) > y(l'')$ for some stack $s$. Once two elements have been inverted, they will remain in the same order in any subsequent reblocking, for it is impossible to find a witness of their reversal again. However, if two lines are in the same block in $\pi'$, it follows from the construction that they are in the same order as in $\pi$. Since we cannot count an intersection twice, $I''$ corresponds exactly to the number of additional inversions from $\pi$ to $\pi'$, which establishes our claim.

The subtle point is that the computation of $\pi'$ is done in $O(n)$ time. This can be most easily seen by the fact that the total insertion time is proportional to the number of times an element steps down in the list. But doing so adds at least $m/2$ inversions, and we know that there are at most $O(nm)$ inversions between $\pi$ and $\pi'$. Therefore going down the list cannot happen more than $O(n)$ times. The time taken by a split is also $O(m)$, and the number of split operations is bounded by the final number of blocks in $\pi'$, which is $O(n/m)$. Thus

**Lemma 3.1 (Reblocking)** *Given an $m$-blocked permutation $\pi$ left- (resp. right-) compatible with $\pi_x$ for some $x$, it is possible to compute in $O(n)$ time an $m$-blocked permutation $\pi'$ left- (resp. right-) compatible with $\pi_{x'}$ for any given $x' > x$ (resp. $x' < x$) such that $|\text{rank}(x') - \text{rank}(x)| = O(nm)$.*

The same reblocking strategy also works for halving the size of the blocks of $\pi$. Simply halve the size of all the blocks of $\pi$ as we did in splitting a stack in the paragraph above.

**Lemma 3.2 (Halving)** *Given an $m$-blocked permutation $\pi$ left (resp. right) compatible with $\pi_x$ for some $x$, it is possible to compute in $O(n)$ time an $(m/2)$-blocked partition which is still left- (resp. right-) compatible with $\pi_x$.*

# 4 The Algorithm

We first define a *vertical slab* $(l, r)$ as the portion $\{(x, y) : l < x \leq r\}$. The algorithm maintains an $m_l$-blocked permutations $\pi(l)$ left-compatible with $\pi_l$, and an $m_r$-blocked permutation $\pi(r)$ right-compatible with $\pi_r$. It also maintains the number of inversions $I_l$ (resp. $I_r$) of $\pi(l)$ (resp. $\pi(r)$). Finally, it stores a collection $\mathcal{T}$ of vertical trapezoids covering $(l, r)$, along with their *conflict lists* (the set of lines crossing them). By analogy with [3, 4], we call $\mathcal{T}$ a *cutting* for the slab $(l, r)$. For accounting purposes, we give a size of 1 to an empty conflict list. In this way, the total size of the conflict lists becomes an accurate indicator of the size of the data structure. The algorithm proceeds in a logarithmic number of *steps*, and maintains the four following invariants at step $j$ (for some large enough constant $c$).

**I1.** $I_l + 2nm_l \leq I^* \leq I_r - 2nm_r$.

**I2.** $I_r - 4nm_r < I^* < I_l + 4nm_l$.

**I3.** Any trapezoid of $\mathcal{T}$ is crossed by at most $n/c^j$ lines.

**I4.** The total size of the conflict lists of $\mathcal{T}$ is at most $cn$.

Informally, I1 says that the vertex sought lies comfortably within the slab $(l, r)$, because of equations (1,2); I2 means that the blocked permutations are no finer than needed, and I3 guarantees that the number of steps will be logarithmic.

A step of the algorithm can be a *slab refinement step*, whose purpose is to narrow the slab $(l, r)$, or a *cutting refinement step*, which subdivides the trapezoids of $\mathcal{T}$. Before explaining how they will be interleaved in the algorithm, let us describe how to perform them.

We halve the slab $(l, r)$ in the same fashion as [4]. A point is said to be *critical* if $(i)$ it lies in the interior of $(l, r)$, and either $(ii)$ it is at the intersection of some line and the upper (lower) boundary of a trapezoid or $(iii)$ it is a vertex of a trapezoid. Intersections along $l$ and $r$ are not critical points. For accounting purposes, we include the vertices with a multiplicity equal to the number of lines crossing the incident vertical boundaries. Let $V$ denote the multiset of all the critical points. From I4, we know that $V$ has size $O(n)$. We compute its median abscissa $h$.

We choose a *winning slab* between $(l, h)$ and $(h, r)$ in the following fashion: for each side $s$ (either $l$ or $r$), we reblock $\pi(s)$ at $x = h$. We then keep halving $m_s$, until I1—I2 are restored. Reblocking and halving are described in the previous section and each takes $O(n)$ time. Potentially, this could be a very long process (in particular if $x^*$ is very near $h$). To avoid this, we organize the computation as follows: each side is run in parallel (think of it as having two distinct processors, or as giving the odd cycles to side $l$ and the even cycles to side $r$), and the computation stops when I1—I2 are restored for either side $s$ (but not necessarily both). If $s$ equals $l$, the winning slab is $(h, r)$, otherwise it is $(l, h)$. We call this the *slab selection process.*

Once we know the winning slab $w$, we discard trapezoids of $\mathcal{T}$ that don't intersect $w$, keep only the portion inside $w$ of those which intersect the line $x = h$ and update their conflict list, and keep those entirely contained in $\mathcal{T}$ as they are. This yields a different cutting $\mathcal{T}'$. If $(l, h)$ is the winning slab, we leave $\pi(l)$ unchanged, and let $h$ replace $r$, with $\pi(h)$ being the permutation reblocked from $\pi(r)$ at $x = h$ and halved as many times as in the slab selection process. We proceed symmetrically if $(h, r)$ is the winning slab. Observe that I1—I4 are maintained, and that the number of critical points in the winning slab has been at least halved compared to those in the slab $(l, r)$. This concludes the description of the slab refinement step.

If we kept iterating on this process, we would quickly run out of critical points. So, we repeat the slab refinement step until the number of all critical points drops below $n/(c \log c)^2$, at which point we compute a $(1/4c)$-net of size $O(c \log c)$ for each of the crossing lists, and compute its vertical trapezoidal map inside the relevant trapezoid. Finally, we increment $j$ by one. This yields a cutting satisfying I3. Note that the new multiset of critical points is of size at most $c_1 n$ (for a constant $c_1$ not depending on $c$). Following [4], we say that an edge of a trapezoid is *free* if it runs entirely across the slab. We remove any the free edge if doing so does not create trapezoids violating I3. Since the free edges are vertically ordered, after removal, there can be at most $O(c^j)$ such edges, accounting for at most $c_2 n$ of the conflict lists elements (for another constant $c_2$ independent of $c$). In order to maintain I4, we observe that each element in the conflict list can either be charged to a new critical point (in number $c_1 n$), to an intersection with the two vertical bounding lines (exactly $2n$ of them), or to a free edge (in total number $c_2 n$). Therefore, taking $c \geq c_1 + c_2 + 2$ ensures that the new cutting satisfies I4 as well. This concludes the description of the cutting refinement step.

We organize the sequence of steps as follows: we refine the slab until one (or more) refinement for the cutting are needed. This can end in one of two ways: either we find that $x^* = h$ because we refined the blocked permutation $\pi(h)$ until it becomes the permutation $\pi_h$ and $I_h = I^*$; or $n/c^j$ becomes less than 1 after refining the cutting. In the latter case, the full arrangement between $l$ and $r$ is available, and its vertices are in number less than $cn$. We compute $\text{rank}(l)$ exactly in $O(n \log n)$ time, and select the vertex of rank $I^* - \text{rank}(l)$ in that list. In both cases, the algorithm succeeds in finding the vertex of rank $I^*$.

## 5 Running Time Analysis

As we observed above, the number of critical points is at least halved during a slab refinement step. Therefore, no more than a constant number of slab refinement steps can occur between two

consecutive cutting refinement steps. The number of cutting refinement steps is $O(\log n)$, since at each such step the maximum size of the conflict lists decreases by a factor of $c$. Thus the total number of refinement steps is $O(\log n)$.

But this number is not necessarily a good indicator of the running time: because we halve repeatedly in the slab selection process, a slab refinement step could take substantially more than $O(n)$ time. However, we can show that this is not the case in the amortized sense: Let $h_i$ be the median of the critical points at any slab refinement step $i$, and let $K_i = |\mathrm{rank}(h_i) - I^*|$; let $0 \le l_0 \cdots \le l_k$, $k = O(\log n)$, be the subsequence of such steps for which the winning slab is $(l, h_i)$, and let similarly $0 \le r_0 \cdots \le r_{k'}$, $k' = O(\log n)$, be the subsequence of such steps for which $(h_i, r)$ is the winning slab.

At the beginning of step $l_j$, $j < k$, we denote the block size of $\pi(r)$ by $m_{r,j}$, and after the halvings it becomes $m_{r,j+1}$. Thus the number of halvings at step $l_j$ is $k_{l_j} = \log(m_{r,j}/m_{r,j+1})$. From I1—I2 and equation (2), we obtain $k_{l_j} = \log(K_{l_j}/K_{l_{j+1}}) + O(1)$. Thus, the total number of halvings performed on $\pi(r)$ throughout the entire algorithm is

$$\sum_{0 \le j < k} k_{l_j} = \sum_{0 \le j < k} \Big(\log(K_{l_j}) - \log(K_{l_{j+1}})\Big) + O(k) = \log(K_{l_0}) - \log(K_{l_k}) + O(k) = O(\log n),$$

since $K_i \le \binom{n}{2}$ for any step $i$. Similarly, $\sum_{0 \le j < k'} k_{r_j} = O(\log n)$. Therefore the total number of halvings (on either side) during all slab refinement steps is also $O(\log n)$.

Since each reblocking, halving, and cutting refinement step takes $O(n)$ time, the total running time of the algorithm is $O(n \log n)$ as claimed, and the storage is $O(n)$.

**Acknowledgements.** We wish to thank Bill Steiger for helpful discussions.

## References

[1] M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3:1–19, 1983.

[2] H. Brönnimann, B. Chazelle, and J. Matoušek. Product range spaces, sensitive sampling, and derandomization. In *Proc. 34th Annu. IEEE Sympos. Found. Comput. Sci. (FOCS 93)*, pages 400–409, 1993.

[3] B. Chazelle. Cutting hyperplanes for divide-and-conquer. *Discrete Comput. Geom.*, 9(2):145–158, 1993.

[4] B. Chazelle, H. Edelsbrunner, L. Guibas, and M. Sharir. Diameter, width, closest line pair, and parametric searching. In *Proc. 8th Annu. ACM Sympos. Comput. Geom.*, pages 120–129, 1992.

[5] B. Chazelle and J. Matoušek. On linear-time deterministic algorithms for optimization problems in fixed dimension. In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, pages 281–290, 1993.

[6] R. Cole, J. Salowe, W. Steiger, and E. Szemerédi. An optimal-time algorithm for slope selection. *SIAM J. Comput.*, 18:792–810, 1989.

[7] M. B. Dillencourt, D. M. Mount, and N. S. Netanyahu. A randomized algorithm for slope selection. *Internat. J. Comput. Geom. Appl.*, 2:1–27, 1992.

[8] M. J. Katz and M. Sharir. Optimal slope selection via expanders. *Inform. Process. Lett.*, 47:115–122, 1993.

[9] J. Matoušek. Approximations and optimal geometric divide-and-conquer. In *Proc. 23rd Annu. ACM Sympos. Theory Comput.*, pages 505–511, 1991. Also to appear in *J. Comput. Syst. Sci.*

[10] J. Matoušek. Randomized optimal algorithm for slope selection. *Inform. Process. Lett.*, 39:183–187, 1991.

[11] N. Megiddo. Applying parallel computation algorithms in the design of serial algorithms. *J. ACM*, 30:852–865, 1983.