

In-Place Randomized Slope Selection

Henrik Blunck and Jan Vahrenhold

Westfälische Wilhelms-Universität Münster, Institut für Informatik,
48149 Münster, Germany. {blunck,jan}@math.uni-muenster.de

Abstract. *Slope selection* is a well-known algorithmic tool used in the context of computing robust estimators for fitting a line to a collection \mathcal{P} of n points in the plane. We demonstrate that it is possible to perform slope selection in expected $\mathcal{O}(n \log n)$ time using only constant extra space in addition to the space needed for representing the input. Our solution is based upon a space-efficient variant of Matoušek’s *randomized interpolation search*, and we believe that the techniques developed in this paper will prove helpful in the design of space-efficient randomized algorithms using samples. To underline this, we also sketch how to compute the repeated median line estimator in an in-place setting.

1 Introduction

Computing a *line estimator*, i.e., fitting a line to a collection \mathcal{P} of n data points $\{p_1, \dots, p_n\}$ in the plane is a frequent task in statistical analysis. Some estimators (such as the *least squares* estimator) can be determined with little computational effort, they suffer, however, from corruption of the estimate by data outliers. Therefore, the robustness of an estimator is considered essential, and the additional computational cost needed to compute robust estimators is widely accepted [10]. A frequently used, robust line estimator is the so-called *Theil-Sen* estimator (see [14] and the references therein) which considers all $\binom{n}{2}$ lines induced by the pairs of points in \mathcal{P} and selects the line with median slope. In the Computational Geometry community, this problem is known as the *slope selection* problem for which an $\Omega(n \log n)$ lower bound has been established [6].

To obtain efficient algorithms, the slope selection problem is studied in the dual setting where each point (x, y) is identified with the line $\{(\xi, v) \mid v = x \cdot \xi - y\}$ and vice versa. Using properties of this duality transform [8], it is easy to show that selecting the k -th smallest slope is dual to the following problem: Given a set of n lines in the plane, find the k -th leftmost intersection point in the arrangement induced by these lines.

While the slope selection problem has been solved optimally both using deterministic and randomized algorithms, we revisit it in the space-efficient model of computation, where the main focus is to use as little extra space as possible over and above the space needed for representing the input. Besides the fact that investigating a possible dependency between time complexity and space requirement is of its own theoretical interest, there are also practical considerations that motivate the design and analysis of space-efficient algorithms. First

and foremost, algorithms with little space overhead have the potential of using the different stages of hierarchical memory, e.g., caches, to a much higher degree of efficiency. Another motivation, especially for designing algorithms for statistical data analysis, comes from the recently increased interest in sensor networks where small-scale computing devices are used to collect large amounts of data. Since the memory of such sensor devices usually is limited, and since transmitting data is more costly than local computation, it is desirable to process as much data as possible locally before transmitting (intermediate) results.

Related Work A variety of deterministic algorithms for solving the slope selection problem in optimal $\mathcal{O}(n \log n)$ running time have been presented [4, 6, 11], however, as noted by Matoušek *et al.* [14], all of them are based on relatively complicated concepts such as parametric search, sorting network, expander graphs, or cuttings. To obtain more practical results, *randomized* approaches have been taken which resulted in several algorithms with expected $\mathcal{O}(n \log n)$ running time [7, 13, 16]. Due to space constraints, we refer the reader to the article by Matoušek *et al.* [14] (and to the references therein) for a discussion of how the slope selection problem relates to the computation of robust estimators.

The Model The goal of investigating space-efficient algorithms is to design algorithms that use very little extra space in addition to the space used for representing the input. The input is assumed to be stored in an array A of size n , thereby allowing random access. We assume that a constant size memory can hold a constant number of words and that each word can hold one pointer, or an $\mathcal{O}(\log n)$ bit integer, and a constant number of words can hold one element of the input array. An *in-place* algorithm uses $\mathcal{O}(1)$ extra words of memory. Some fundamental geometric problems such as 2D convex hulls and closest pairs can be solved *in-place* and in optimal time [1, 2, 5]. More involved problems such as range searching and line-segment intersection (currently) can be solved in-place only in near-optimal running time [3, 17], or, as for the case of 3D convex hulls and related problems, using both (poly-)logarithmic extra space and time [3].

Our Results In this paper we show how to solve the slope selection problem in expected $\mathcal{O}(n \log n)$ running time while at the same time using only constant extra space. Our algorithm follows the approach of Matoušek [13], and thus we also devise an in-place variant of his *randomized interpolation search* technique. This variant, together with a algorithmic subroutine for efficiently constructing and storing a set of randomly sampled intersections, is of independent interest, since it was introduced by Matoušek to substitute Meggido's *parametric search* technique [15]. Furthermore, we sketch how to use these results to obtain an in-place algorithm for computing the *repeated median line estimator*.

2 Space-Efficient Algorithmic Tools

Space-efficient algorithms have been investigated since the 1960's, and there exists a variety of algorithmic building blocks that require little extra space

in addition to the space needed for representing the input. *Heapsort* [18] sorts n elements in-place performing $\mathcal{O}(n \log n)$ operations. Also, we can do linear-time median-finding in-place—even while maintaining the data sorted according to some other total order [1]. Finally, we will use the linear-time merging algorithm by Geffert *et al.* [9] that merges two sorted subarrays using constant extra space.

A standard technique in the design of space-efficient algorithms is to encode a single bit by a permutation of two objects q and r [?]. In our case, for lines q, r with $q <_{\xi} r$ (i.e., with q preceding r in the vertical order at $x = \xi$) the permutation qr encodes a binary zero, and the permutation rq encodes a binary one (by the nature of the problem setting, the input does not contain duplicates). Given this representation, we can encode any integer in the range $[0, \dots, n - 1]$, e.g., an index referencing an array entry, using $2 \cdot \log n$ lines, and the time needed for decoding or modifying any such pointer is $\mathcal{O}(\log n)$.

As a subroutine of our algorithm, we will need to sort a set of encoded pointers according to some order imposed on the elements referenced by these pointers. If we use *heapsort* to sort a set of r elements (pointers), we perform $\mathcal{O}(r \log r)$ operations. Since we may need to decode and dereference $\mathcal{O}(1)$ pointers to perform one such operation, the overall running time is $\mathcal{O}(r \log r \log n)$. Note, that some pointers may reference lines that are part of a pointer encoding, and thus, whenever the sorting algorithm tells us to swap two pointers, we must not move around the lines representing these pointers. Instead, we simply swap the encoded *values* by making the first pointer reflect the value of the second and vice versa. As a consequence, no line will be more than one position off its correct location (and we can check this in constant time per access). Each swap resp. update can be done in-place and in $\mathcal{O}(\log n)$ time. This results in the following corollary:

Corollary 1. *We can sort r binary encoded pointers of size $\mathcal{O}(\log n)$ bits each in-place and in $\mathcal{O}(r \log r \log n)$ time while preserving the referenced information.*

3 Randomized Interpolation Search

Due to the duality transform discussed in the introduction, the problem of selecting the line with median slope translates to selecting the intersection with median x -coordinate in the arrangement induced by the lines dual to the points in \mathcal{P} . Since the duality transform is done read-only by simply reinterpreting the coordinate-based representation of a point as the slope-intercept representation of a line, we assume that our input is given as a set \mathcal{P} of lines in the plane.

Since the arrangement induced by a set of n lines contains $\Theta(n^2)$ intersections, it is infeasible (both with respect to the desired $\mathcal{O}(n \log n)$ running time and with respect to the in-place setting) to compute all intersections. Instead, the algorithm of Matoušek [13] maintains a vertical strip $\langle b, e \rangle := [b, e] \times \mathbb{R} \subset \mathbb{R}^2$ that is guaranteed to contain the k -th smallest intersection point. This strip is iteratively refined by first constructing a sample \mathcal{R} of r intersection points (r is a parameter to be defined later) and by then constructing a (narrower) candidate strip $\langle b', e' \rangle$ based upon the x -coordinates of two appropriately chosen

sampled intersection points. The algorithm then checks whether $\langle b', e' \rangle$ indeed contains the k -th smallest intersection point. If this is not the case, the process is repeated for $\langle b, e \rangle$ but using a new sample \mathcal{R} , else the algorithm iterates with the refined strip $\langle b', e' \rangle$. Since the above technique refines the strip $\langle b, e \rangle$ based upon a non-constant number of randomly selected samples, it is referred to as *randomized interpolation search*, and Matoušek observes that it can be used as a randomized substitute for Megiddo's *parametric search* technique [15].

The efficiency of the resulting algorithm for slope selection is based upon the following lemma which (applied iteratively) implies that the number $|\mathcal{I}(b, e)|$ of intersections that lie inside $\langle b, e \rangle$ can be reduced to $\mathcal{O}(r)$ using an expected constant number of iterations:

Lemma 1 (Lemma 2.1 in [14]). *Given a set of numbers $\Theta = \{\theta_1, \theta_2, \dots, \theta_N\}$, an index k ($1 \leq k \leq N$), and an integer $r > 0$, we can compute in $\mathcal{O}(r)$ time an interval $[\theta_{\text{lo}}, \theta_{\text{hi}}]$, such that, with probability $1 - 1/\Omega(\sqrt{r})$, the k -th smallest element of Θ lies within this interval, and the number of elements in Θ that lie within the interval is at most $N/\Omega(\sqrt{r})$.*

We use the algorithm implied by the above lemma to compute the intersection with the k -th smallest x -coordinate among the $N = \binom{n}{2}$ intersections induced by the lines in \mathcal{P} . Since Matoušek *et al.* [14] proved that it is possible to choose $r := \lceil N^\beta \rceil$ for any $0 < \beta < 1$ while maintaining the asymptotic efficiency of the resulting algorithm, we set $r := \lceil \sqrt{n} \rceil$ as opposed to the original choice of $r := n$ [13]. As Matoušek [13] computes $[\theta_{\text{lo}}, \theta_{\text{hi}}]$ using $\mathcal{O}(r)$ sampled intersections, the bound on the running time given in Lemma 1 will be replaced by the $\mathcal{O}(n \log n)$ time bound for the in-place sampling algorithm discussed in the next section. In any case, we note that the probability and the bound on the size of the elements within $[\theta_{\text{lo}}, \theta_{\text{hi}}]$ is independent of whether or not the algorithm is implemented in-place. The main algorithm for slope selection (as presented by Matoušek [13]) is given below as Algorithm 1.

It remains to describe how to construct (and store!) the sampled set \mathcal{R} of $r = \lceil \sqrt{n} \rceil$ intersections in an in-place setting, i.e., using only constant extra space. Furthermore, we need to discuss how to compute $|\mathcal{I}(b, e)|$. To this effect, we describe an algorithm for the former task, which also provides a solution for the latter task. Anticipating the results presented in the next section, we combine them with the above lemma and the original analyses of Matoušek *et al.* [13, 14]:

Theorem 1. *The slope selection problem for a set of n input points in the plane can be solved in-place and in expected $\mathcal{O}(n \log n)$ running time.*

4 Constructing a Random Sample \mathcal{R} of r Intersections

In this section, we describe an algorithmic subroutine that will be invoked from Line 5 of the overall slope selection algorithm (Algorithm 1) with three parameters b , e , and r . Its purpose is to draw a random sample (with replacement) of size $r := \lceil \sqrt{n} \rceil$ from the set of all intersections induced by the lines in \mathcal{P} and

Algorithm 1 Algorithm SELECT($A[0, \dots, n-1], k, r$) for selecting the k -th leftmost intersection point in the arrangement induced by the lines in A [13].

```

1: Let  $\kappa := k$ . Let  $b := -\infty$  and  $e := \infty$ .    {Initial “guess” for the candidate strip.}
2: while not finished do
3:   Let  $N := |\mathcal{I}(b, e)|$ .                                {Number of intersections inside  $\langle b, e \rangle$ .}
4:   if  $N > r$  then
5:     Draw (with replacement) a random sample  $\mathcal{R}$  of size  $r$  from  $\mathcal{I}(b, e)$ .
6:     Let  $\kappa := (r/N)(k - |\mathcal{I}(-\infty, b)|)$ ,  $\kappa_{b'} := \max(1, \lfloor \kappa - 3\sqrt{r}/2 \rfloor)$ , and
        $\kappa_{e'} := \min(r, \lfloor \kappa + 3\sqrt{r}/2 \rfloor)$ .
7:     Compute the intersections with ranks  $\kappa, \kappa_{b'}, \kappa_{e'}$  in  $\mathcal{R}$  (ranks are with respect
       to the order of the  $x$ -coordinates).
8:     Let  $b'$  be the  $x$ -coordinate of the intersection in  $\mathcal{R}$  with rank  $\kappa_{b'}$ , and let  $e'$ 
       be the  $x$ -coordinate of the intersection with rank  $\kappa_{e'}$ .
9:     if  $|\mathcal{I}(-\infty, b')| < k < |\mathcal{I}(-\infty, e')|$  then
10:      Let  $b := b'$  and  $e := e'$ .    {The  $k$ -th leftmost intersection point is in  $\langle b', e' \rangle$ ;
        update  $b$  and  $e$ .}
11:    end if
12:  else
13:    Report the intersection with rank  $\kappa$  in  $\mathcal{R}$  as the  $k$ -th leftmost intersection
        point and finish.
14:  end if
15: end while

```

falling within $\langle b, e \rangle$. We demonstrate that we can compute and represent these r intersections in-place and in time $\mathcal{O}(n \log n + r^2 \log n)$. Due to space constraints, we refer the reader to the full version of this paper for the proofs of the respective running times.

4.1 An Overview of the Algorithm

Our subroutine follows the approach of Matoušek [13, Lemma 1]: we first draw (with replacement) a set of r random integers from $\{0, \dots, |\mathcal{I}(b, e)| - 1\}$ where $|\mathcal{I}(b, e)|$ is the number of intersections in $\langle b, e \rangle$ and sort these numbers. These numbers give the ranks of the intersections with respect to the order in which they are found. The main ingredient used for efficiently computing intersections in $\langle b, e \rangle$ is the following well-known observation: the number of intersections inside $\langle e, b \rangle$ is exactly the number of inversions between the permutation of \mathcal{P} that arranges the lines in sorted $<_b$ -order (the vertical order at $x = b$) and the permutation that arranges the lines in sorted $<_e$ -order (the vertical order at $x = e$) [12]. Thus, to efficiently compute $|\mathcal{I}(b, e)|$, we can run the classic divide-and-conquer algorithm for inversion counting (see, e.g., [12]). While doing so, we keep track of the total number of inversions/intersections seen so far and “record” an intersection if its rank matches one of the r given ranks. Since we have sorted the ranks, we can process each of them in constant extra time.

Running Matoušek’s algorithm in an in-place setting is complicated by two facts: (1) The recursion stack of an divide-and-conquer algorithm may require

$\Omega(\log n)$ extra words, and (2) we need to store the r ranks and the intersections computed so far. The problem of performing a recursive algorithm in-place has been discussed earlier [1], and we choose to process the “recursion tree” implicitly in a bottom-up, level-by-level traversal. We overcome the second problem by running the algorithm in three phases: in the first phase, we process the lines stored in $A[0, \dots, \lfloor n/2 \rfloor - 1]$ and use $A[\lfloor n/2 \rfloor, \dots, n - 1]$ to encode the ranks and the intersections found so far, and in the second phase, we reverse the roles of both subarrays. We finalize the algorithm with a third phase that processes the intersections induced by lines stored in different halves of the array.

4.2 In-Place Data Structures for Recording Intersections

Employing the encoding technique described in Section 2, we use the subarray of size $n/2$ that does not contain the lines to be processed in the current phase to represent three (implicit) data structures \mathcal{D}_R , \mathcal{D}_L , and \mathcal{D}_I . Each of these data structures is represented in a subarray containing $4 \cdot r \log n$ lines. For reasons that will become evident in Section 4.3, we also reserve a scratch space of the same size as the data structure \mathcal{D}_L .¹ The array thus is partitioned as follows:

Lines to be processed		\mathcal{D}_R	Scratch	\mathcal{D}_L	\mathcal{D}_I
0	$\frac{1}{2}n$				$n - 1$

Storing ranks The randomly generated ranks in the range $[0, \dots, n^2 - 1]$ are encoded in a “sorted-list” data structure \mathcal{D}_R . During the initialization of \mathcal{D}_R , these ranks are sorted using *heapsort* in $\mathcal{O}(r \log r \log n)$ time (see Section 2). Thus, our algorithm will be able to traverse the list and report each rank to be processed in $\mathcal{O}(\log n)$ time.

Storing lines involved in intersections We use a “sorted-list” data structure \mathcal{D}_L to record (references to) lines involved in all of the sampled intersections found so far. These (references to) lines are maintained in sorted $<_b$ -order. Every reference is inserted into \mathcal{D}_L using insertion sort (ignoring duplicates), leading to a $\mathcal{O}(r^2 \log n)$ global cost for maintaining \mathcal{D}_L .

Storing intersections The “linked-list” data structure \mathcal{D}_I records the intersections found so far by indexing into \mathcal{D}_L . To add an intersection induced by two lines ℓ_1 and ℓ_2 to \mathcal{D}_I , we first insert (references to) ℓ_1 and ℓ_2 in sorted $<_b$ -order into \mathcal{D}_L and then append the pair (i, j) referencing (the references in \mathcal{D}_L to) these two lines at the end of \mathcal{D}_I . The cost for performing a single insertion into \mathcal{D}_I is in $\mathcal{O}(\log n)$, and thus the global cost is in $\mathcal{O}(r \log n)$.

As a consequence, we have the following lemma:

Lemma 2. *The global cost for maintaining the data structures \mathcal{D}_R , \mathcal{D}_L , and \mathcal{D}_I needed to record the r intersections in an in-place manner is $\mathcal{O}(r^2 \log n)$.*

¹ We may need to adjust r by a constant factor such as to ensure that $16 \cdot r \log n \leq n/2$.

4.3 Processing one half of the subarray

The algorithms for processing the halves of $A[0, \dots, n-1]$ are symmetric, and thus we present the algorithm for processing the subarray $A[0, \dots, \lfloor n/2 \rfloor - 1]$.

Counting inversions The algorithm for counting *all* inversions in $\langle b, e \rangle$ is an extension of the iterative *mergesort* algorithm: starting from the set of lines in sorted \prec_b -order, the algorithm iteratively merges the lines into \prec_e -order while counting inversions. During each *merge*-step of the algorithm, two subarrays already in sorted \prec_e -order are merged into a single \prec_e -sorted subarray. Each of these subarrays has been processed during the previous iteration, and thus all inversions involving lines from only one of these subarrays have been processed. Since all elements in A_1 precede any element in A_2 with respect to the \prec_b -order, the inversions induced by lines from different subarrays can be computed using the following observation [12, 13]: Each element $a_i \in A_1$ induces an inversion with all elements in A_2 preceding it in sorted \prec_e -order. Thus, to compute the number of inversions it is sufficient to maintain a counter that records how many elements from A_2 have been written into sorted \prec_e -order. An obvious, yet crucial, fact guaranteeing the correctness of the inversion counting algorithm is that any two subarrays A_1 and A_2 that are merged in the j -th iteration of processing the m -th bottom-most level of the recursion tree are of the form $A_1 := A[j \cdot 2^m, \dots, (j+1) \cdot 2^m - 1]$ and $A_2 := A[(j+1) \cdot 2^m, \dots, (j+2) \cdot 2^m - 1]$.²

This leads to the following corollary which we can prove using induction and the fact that the algorithm starts with all lines in sorted \prec_b -order:

Corollary 2. *Let A_1 and A_2 be two subarrays of A that are merged in one step of the algorithm for inversion counting. Furthermore, define ℓ_{\min} to be the minimal line of $A_1 \cup A_2$ with respect to the \prec_b -order and define ℓ_{\max} to be the maximal line of $A_1 \cup A_2$ with respect to the \prec_b -order. Then, $A_1 \cup A_2$ consists exactly of those lines ℓ in A for which $\ell_{\min} \leq_b \ell \leq_b \ell_{\max}$ holds.*

The algorithm COUNTANDRECORD (Algorithm 2) extends the above inversion counting algorithm to also record all intersections inside $\langle b, e \rangle$ whose ranks are recorded in \mathcal{D}_R .

Algorithm 2 can be implemented using constant extra space, and excluding the cost for recording the relevant intersections, the cost for running Algorithm 2 is linear in the size of the union of the two subarrays to be merged. Furthermore, Algorithm 2 can be simplified (by leaving out the code in Lines 6–9) to compute $|\mathcal{I}(b, e)|$ as required in Lines 3 and 9 of Algorithm 1. Afterwards, A_1 and A_2 can be merged using, e.g., the linear-time merging algorithm of Geffert *et al.* [9]. In combination with a bottom-up divide-and-conquer approach, this gives an in-place $\mathcal{O}(n \log n)$ implementation for the “inversion counting” in Algorithm 1.

² As noted earlier [1], the in-place divide-and-conquer scheme can be modified easily to correctly handle instances where the problem size is not a power of two.

Algorithm 2 Algorithm COUNTANDRECORD($A_1, A_2, \langle b, e \rangle, c, \mathcal{D}_I$) for incrementing the count c of intersections seen so far by the number of intersections induced by lines in A_1 and A_2 while recording all relevant intersections in \mathcal{D}_I . Only intersections that fall inside $\langle b, e \rangle$ are considered.

Require: A_1 and A_2 are sorted according to $<_e$. Each element in A_1 precedes all elements in A_2 .

Ensure: A_1 and A_2 have not been modified, all intersections within $\langle b, e \rangle$ induced by lines from A_1 and A_2 have been recorded, and c reflects the number of intersections seen so far.

```

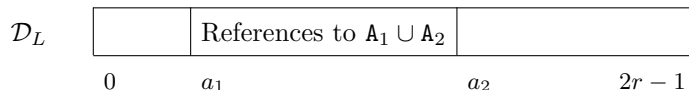
1: Let  $i_1 := 0$  and  $i_2 := 0$ . {Current position in subarray.}
2: for  $i = 0$  to  $\text{length}(A_1 \cup A_2) - 1$  do
3:   Let  $\ell_1 := A_1[i_1]$  and  $\ell_2 := A_2[i_2]$ . {The  $i$ -th element in sorted order is  $\ell_1$  or  $\ell_2$ .}
4:   if  $\ell_1 <_e \ell_2$  then
5:     Let  $c_{i_1} := i_2$ . { $\#$ (inversions induced by  $\ell_1$ ) =  $\#$ (elements in  $A_2$  preceding  $\ell_1$ ).}
6:     for each rank  $\rho$  in  $\mathcal{D}_R \cap [c, \dots, c + c_{i_1}]$  do
7:       Let  $\ell$  be the line stored at  $A_2[\rho - c]$ .
8:       Update  $\mathcal{D}_I$  to record the pair  $(\ell_1, \ell)$  as the intersection with rank  $\rho$ .
9:     end for
10:    Let  $c := c + c_{i_1}$ . {Count intersections.}
11:     $i_1 := i_1 + 1$ . {Advance  $i_1$ .}
12:   else
13:     $i_2 := i_2 + 1$ . {Advance  $i_2$  but do not do anything else.}
14:   end if
15: end for

```

Merging two subarrays into sorted $<_e$ -order When actually merging A_1 and A_2 into sorted $<_e$ -order, we also need to update the values stored in \mathcal{D}_L , since they reference the lines involved in the intersections found so far by directly indexing into A . Merging A_1 and A_2 seems to corrupt the information recorded in \mathcal{D}_L , but fortunately the information of what goes where can be computed on the fly while running COUNTANDRECORD. To see this, observe that the algorithm COUNTANDRECORD is an iterative algorithm that computes, during the i -th iteration the element with rank i in the final sorted order (Line 3 of Algorithm 2). Thus we only need to be able to find out whether the line ℓ that will be the i -th element in sorted order is an element involved in an intersection. If this is the case, we simply update the reference pointing to ℓ to point to ℓ 's position *after* the merge step: the i -th position in the union of A_1 and A_2 . In order not to corrupt the values of \mathcal{D}_L that are needed for correctly updating \mathcal{D}_I , we do not directly modify \mathcal{D}_L to record the “new” position of each referenced entry. Instead, prior to processing each level of the recursion tree, we copy \mathcal{D}_L into the scratch space that has been reserved earlier (see Section 4.2) and denote this copy as \mathcal{D}'_L . This copy will be used to record the updated values of \mathcal{D}_L .

Selecting the lines in A_1 and A_2 referenced from \mathcal{D}_L . Our algorithm for efficiently determining whether a line ℓ to be processed is one of the lines referenced from \mathcal{D}_L is based upon Corollary 3. This corollary states that the entries in \mathcal{D}_L that

reference lines in the union of A_1 and A_2 are stored consecutively in a (possibly empty) sublist $\mathcal{D}_L[a_1, \dots, a_2 - 1]$.



We obtain this sublist as follows: Prior to running the algorithm COUNTAND-RECORD on two subarrays A_1 and A_2 , we scan $A_1 \cup A_2$ to determine (in linear time) the highest and lowest line of $A_1 \cup A_2$ with respect to the $<_b$ -order. Using this information, we determine the (possibly empty) sublist $\mathcal{D}_L[a_1, \dots, a_2 - 1]$ that contains all elements in \mathcal{D}_L that reference any line in $A_1 \cup A_2$. Corollary 3 guarantees that the index a_1 used when merging the next two adjacent subarrays is obtained by setting $a_1 := a_2$ and that a_2 can be updated by scanning forward from $\mathcal{D}_L[a_2]$, i.e., a_1 and a_2 can be updated iteratively.

To efficiently use this sublist, we sort both $\mathcal{D}_L[a_1, \dots, a_2 - 1]$ and its copy $\mathcal{D}'_L[a_1, \dots, a_2 - 1]$ according to the $<_e$ -order. This order allows us to process not only A_1 and A_2 but also all references in $\mathcal{D}_L[a_1, \dots, a_2 - 1]$ during a synchronized scan. All of the above operations can be done in a (globally) efficient way:

Lemma 3. *The global cost for updating a_1 and a_2 is $\mathcal{O}(n \log n + r \log^2 n)$.*

Updating the relevant references in \mathcal{D}_L . Recall that the merging algorithm determines, during its i -th iteration, the line ℓ that is the i -th line of $A_1 \cup A_2$ in sorted $<_e$ -order (Line 3 of Algorithm 2). During the synchronized scan, we keep track of the lowest line μ referenced from $\mathcal{D}_L[a_1, \dots, a_2 - 1]$ not lower than ℓ ; μ is the next line for which we need to record its new position. Let j be the index such that $\mathcal{D}_L[j]$ references μ . In case ℓ equals μ , we record that μ will reside at the i -th position of $A_1 \cup A_2$ in sorted $<_e$ -order by updating $\mathcal{D}'_L[j]$ accordingly.

To make Algorithm 2 reflect these updates, we need to modify Lines 11 and 13 such as to compare the current line to μ and to update the index j as needed. This results in $\mathcal{O}(\log n)$ time spent per step in which the index j is updated, i.e., in $\mathcal{O}(r \log n)$ time per level of the recursion tree, since we can store the current value of μ using constant extra space and thus allow for constant-time access to μ . Thus, the overall extra time spent in these steps is in $\mathcal{O}(r \log^2 n)$.

Caveat: Updating \mathcal{D}_I and \mathcal{D}_L . Adding a new intersection to \mathcal{D}_I involves adding up to two (references to) lines to \mathcal{D}_L , and this is the sole reason for working with the copy \mathcal{D}'_L of \mathcal{D}_L and thus requiring extra scratch space. First of all, since all intersections recorded involve lines from $A_1 \cup A_2$, any (reference to a) line ℓ that is added to \mathcal{D}_L will increase the value of a_2 by one. Furthermore, to keep $\mathcal{D}_L[a_1, \dots, a_2 - 1]$ in sorted $<_e$ -order, we need to change the order with respect to which we insert into \mathcal{D}_L from the $<_b$ -order to the $<_e$ -order as soon as the insertion process reaches $\mathcal{D}_L[a_1, \dots, a_2 - 1]$. These operations are simultaneously performed on \mathcal{D}'_L as well.

For each component of a pair of indices that is added to \mathcal{D}_I to record a newly-found intersection, we need to make sure that it encodes the correct position of

the referenced line ℓ in \mathcal{D}_L , i.e., the correct position of ℓ w.r.t. the *sorted* $<_b$ -order. To compute this position, we scan $\mathcal{D}_L[a_1, \dots, a_2 - 1]$ to count the number of lines preceding ℓ in sorted $<_b$ -order. (Note that this computation is only possible because, during the inversion counting step, we have updated the references in \mathcal{D}'_L instead of the references in \mathcal{D}_L ; thus, the entries in \mathcal{D}_L still correctly reference the relevant lines at their *current* position.) If j lines precede ℓ , we update all indices in \mathcal{D}_I referencing entries in $\mathcal{D}_L[a_1 + j, \dots, |\mathcal{D}_L| - 1]$.

After having run the merging algorithm on A_1 and A_2 , we reestablish the correct relation between entries in \mathcal{D}_I and entries in \mathcal{D}_L : We replace the values encoded in $\mathcal{D}_L[a_1, \dots, a_2 - 1]$ with the values encoded in $\mathcal{D}'_L[a_1, \dots, a_2 - 1]$ and sort them according to $<_b$. This finishes processing the subarrays A_1 and A_2 .

Lemma 4. *The global extra cost incurred by updating the references stored in the data structure \mathcal{D}_L while merging subarrays is in $\mathcal{O}(r \log r \log^2 n)$.*

4.4 Finishing up

After we have processed the first half of the input array using the second half to maintain the data structures \mathcal{D}_R , \mathcal{D}_L , and \mathcal{D}_I , we now reverse the roles of the two subarrays. To do so, we first need to copy the contents of the data structures to the first half of the array. The important detail to keep in mind is that, as a result of the inversion-counting algorithm, the first half of the array is sorted according to $<_e$. Thus, when copying the contents of the data structures to the first half of the array, the order according to which we have to decide whether two adjacent elements encode a binary zero or a binary one, is the $<_e$ -order.

After we have run our subroutine on $A[\lfloor n/2 \rfloor, \dots, n - 1]$, we need to finalize the algorithm by processing all intersections induced by lines stored in different halves of the array. As it turns out, however, we do not need to actually merge the lines in $A[0, \dots, \lfloor n/2 \rfloor - 1]$ and $A[\lfloor n/2 \rfloor, \dots, n - 1]$ —it is sufficient to count the inversions and to construct the relevant intersections. This means that we can simply run the algorithm COUNTANDRECORD (Algorithm 2) without the modifications needed to record the “what-goes-where” information. As discussed in Section 2, the lines used to encode the data structures \mathcal{D}_R , \mathcal{D}_L , and \mathcal{D}_I are at most one position off their correct position (in sorted $<_e$ -order), and thus we can process them in sorted $<_e$ -order with constant extra (look-ahead) space.

As a result of this final invocation of the algorithm COUNTANDRECORD, the data structure \mathcal{D}_I will reference r pairs of entries in \mathcal{D}_L which in turn reference pairs of lines in A . To select the intersections with rank κ , $\kappa_{b'}$, and $\kappa_{e'}$ (Line 7 of Algorithm 1), we could invoke a linear-time median-find algorithm. However, since we have chosen r small enough, we can simply use the algorithm implied by Corollary ?? to sort the pairs in \mathcal{D}_I according to the x -coordinate of their intersection. The running time (including the time needed for resolving one level of indirection) is $\mathcal{O}(r \log r \log n)$. Combining this with Lemmas 2, 3, and 4 and the fact that $r \in \mathcal{O}(\sqrt{n})$, we obtain the following lemma:

Lemma 5. *A random sample \mathcal{R} of $r = \lceil \sqrt{n} \rceil$ intersections inside a strip $\langle b, e \rangle$ can be constructed in-place and in $\mathcal{O}(n \log n)$ time.*

The same algorithm can be used to explicitly construct *all* of the at most r intersection points that are considered in the last iteration of Algorithm 1 (Line 13). This finishes the proof of Theorem 1.

5 Computing the Repeated Median Line Estimator

The *repeated median line estimator* is an estimator for line-fitting, which is even more robust with respect to data outliers—see [14] and the references therein. The repeated median line estimator is obtained by first computing, for each input point p_i the line m_i with median slope among all $n - 1$ lines induced by p_i and another point in $\mathcal{P} \setminus \{p_i\}$. Among all such lines m_i , the line m that realizes the line with median slope is selected as the repeated median line estimator. In the dual setting this corresponds to finding, for each line ℓ , the intersection point on ℓ with the median x -coordinate and then to compute the median of these medians. Since the underlying algorithm is considerably more involved and maintains a much larger amount of statistical information, we are unable to match the expected running time of the original algorithm, and our in-place implementation exhibits an expected near-optimal running time.

Matoušek *et al.* [14] describe multiple variants of a randomized algorithm for computing a repeated median line estimator using $\mathcal{O}(n)$ extra space. The general approach is to proceed using randomized interpolation search in an only slightly different setting than the one described for slope selection (Section 3). In their case, the set \mathcal{R} of samples drawn in each iteration is drawn from the set \mathcal{P} , i.e., from the set of input lines. Then, they compute the repeated median line estimator for all (points dual to) lines in \mathcal{R} and a candidate strip $\langle b, e \rangle$ that is supposed to contain the (global) final result. To verify that $\langle b, e \rangle$ indeed contains the median-of-medians intersection, the algorithm of Matoušek *et al.* then counts the number of medians that lie to the left, inside, and to the right of $\langle b, e \rangle$, respectively. They do so by computing all of these counts during a single run of a modification of the inversion-counting algorithm. During this run, counters L_i and I_i are maintained for each line ℓ_i that record the number intersections involving ℓ_i left of, resp. inside $\langle b, e \rangle$.

The main purpose of computing the counters L_i and I_i is to determine, for each line ℓ_i independently, whether its median intersection lies inside $\langle b, e \rangle$. Processing all n lines in one batch (as described above) is done for the sole purpose of efficiency. Since, in an in-place setting, we cannot accommodate $\mathcal{O}(n)$ counters in a subarray of size n , we need to process the lines in smaller batches. If we choose the batch size to be $\mathcal{O}(n/\log n)$, we can compute all counts in-place. This, however, increases the number of runs needed for checking whether $\langle b, e \rangle$ is a valid candidate from one to $\mathcal{O}(\log n)$. Additionally, each step of the algorithm for updating the counters now takes $\mathcal{O}(\log n)$ extra time for updating a binary-encoded value, and thus we lose another $\mathcal{O}(\log n)$ -factor in the overall running time. Finally, the most efficient strategy for constructing the candidate strip $\langle b, e \rangle$ described in [14], computing the repeated median line by range searching and counting techniques, seems to be unavailable in an in-place setting. As

a consequence, we have to resort to an in-place variant of a less efficient randomization technique, that (in contrast to the situation described in Section 3) results in an expected number of $\mathcal{O}(\log n)$ iterations of the global algorithm.

Lemma 6. *The repeated median line estimator can be computed in-place and in expected $\mathcal{O}(n \log^4 n)$ time.*

References

1. P. Bose, A. Maheshwari, P. Morin, J. Morrison, M. Smid, and J. Vahrenhold. Space-efficient geometric divide-and-conquer algorithms. *Computational Geometry: Theory & Applications*, 2006. To appear, accepted November 2004.
2. H. Brönnimann and T. M.-Y. Chan. Space-efficient algorithms for computing the convex hull of a simple polygonal line in linear time. *Computational Geometry: Theory & Applications*, 2006. In press.
3. H. Brönnimann, T. M.-Y. Chan, and E. Y. Chen. Towards in-place geometric algorithms. In *Proc. 20th Symp. Computational Geometry*, pp. 239–246, 2004.
4. H. Brönnimann and B. M. Chazelle. Optimal slope selection via cuttings. *Computational Geometry: Theory and Applications*, 10(1):23–29, 1998.
5. H. Brönnimann, J. Iacono, J. Katajainen, P. Morin, J. Morrison, and G. T. Toussaint. Space-efficient planar convex hull algorithms. *Theoretical Computer Science*, 321(1):25–40, 2004.
6. R. Cole, J. S. Salowe, W. L. Steiger, and E. Szemerédi. An optimal-time algorithm for slope selection. *SIAM J. Computing*, 18(4):792–810, 1989.
7. M. B. Dillencourt, D. M. Mount, and N. S. Nethanyahu. A randomized algorithm for slope selection. *Intl. J. Computational Geometry and Applications*, 2(1):1–27, 1992.
8. H. Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer, Berlin, 1987.
9. V. Geffert, J. Katajainen, and T. Pasanen. Asymptotically efficient in-place merging. *Theoretical Computer Science*, 237(1–2):159–181, 2000.
10. P. Huber. *Robust Statistics*. Wiley, New York, 1981.
11. M. J. Katz and M. Sharir. Optimal slope selection via expanders. *Information Processing Letters*, 47(3):115–122, 1993.
12. J. Kleinberg and É. Tardos. *Algorithm Design*. Addison-Wesley, Boston, 2006.
13. J. Matoušek. Randomized optimal algorithm for slope selection. *Information Processing Letters*, 39(4):183–187, 1991.
14. J. Matoušek, D. M. Mount, and N. S. Nethanyahu. Efficient randomized algorithms for the repeated median line estimator. *Algorithmica*, 20(2):136–150, 1998.
15. N. Megiddo. Applying parallel computation algorithms in the design of serial algorithms. *J. ACM*, 30(4):852–865, 1983.
16. J. I. Munro. An implicit data structure supporting insertion, deletion, and search in $O(\log^2 n)$ time. *J. Computer and System Sciences*, 33(1):66–74, 1986.
17. L. Shafer and W. L. Steiger. Randomizing optimal geometric algorithms. In *Proc. 5th Canadian Conference on Computational Geometry*, pp. 133–138, 1993.
18. J. Vahrenhold. Line-segment intersection made in-place. In *Proc. 9th Workshop Algorithms and Data Structures, Lecture Notes in Computer Science 3608*, pp. 146–157, 2005.
19. J. W. J. Williams. Algorithm 232: Heapsort. *Comm. ACM*, 7(6):347–348, 1964.