PROGRESSIVE GEOMETRIC ALGORITHMS*

*S.P.A. Alewijnse,[†] T.M. Bagautdinov,[‡] M. de Berg,[†] Q.W. Bouts,[†] A.P. ten Brink,[†]
K. Buchin,[†] M.A. Westenberg[†]*

ABSTRACT. Progressive algorithms are algorithms that, on the way to computing a complete solution to the problem at hand, output intermediate solutions that approximate the complete solution increasingly well. We present a framework for analyzing such algorithms, and develop efficient progressive algorithms for two geometric problems: computing the convex hull of a planar point set, and finding popular places in a set of trajectories.

1 Introduction

Traditional algorithms compute the complete solution to a problem without allowing for interaction or giving intermediate results. This was the most natural approach when interaction with computing devices was cumbersome, and in many applications it may still be the desired approach. Modern applications, however, often involve large data sets that need to be explored and analyzed interactively. In this paper we therefore study geometric algorithms that do not focus solely on computing the complete solution, but that produce meaningful partial solutions early on. This way the user can decide whether it is necessary to continue the computation and, perhaps, in which way the computation should proceed. This is useful in an exploratory setting with a human user, but also in a traditional setting where another algorithm uses the given algorithm as a subroutine.

We are interested in algorithms that report, on their way to computing a complete solution, partial solutions that approximate the complete solution increasingly well. We will call such algorithms *progressive algorithms*, and we will use the following framework. We have an *error function* that assigns to every partial solution S a non-negative value $err(S)$. A partial solution S with $err(S) = 0$ is called a *complete solution*. A progressive algorithm is an algorithm that works in *rounds*, reporting partial solutions with smaller error guarantees in each round, until in the last round a complete solution is reported. The error guarantees are specified by a *convergence function* $f_{\text{conv}} : \{1, \dots, k\} \rightarrow \mathbb{R}_{\geq 0}$, where k is the number of rounds. More precisely, a *k-round progressive algorithm with convergence function* f_{conv} with respect to error function err is an algorithm that outputs, after the r -th round (for $1 \leq r \leq k$), a partial solution S_r with $err(S_r) \leq f_{\text{conv}}(r)$. Preferably a progressive algorithm is also *monotone*: the actual error should decrease in each round and each partial solution should be a “refinement” of the previous one. (The meaning of the latter depends on the

*M. de Berg, Q.W. Bouts and K. Buchin are supported by the Netherlands Organisation for Scientific Research (NWO) under project no. 024.002.003, 639.023.208 and 612.001.207, respectively.

[†]TU Eindhoven, The Netherlands

[‡]Ecole Polytechnique Fédérale de Lausanne, Switzerland

application. For a progressive convex-hull algorithm, for instance, one could require that the convex polygon reported at any given round fully contains the convex polygon reported in the previous round.) A more extensive discussion of our framework is given in Section 2.

Progressive algorithms are closely related to *anytime algorithms*, which were already studied in the AI and real-time systems communities in the nineties; see for example the early survey by Zilberstein [12]. Intuitively, anytime algorithms are algorithms that can be interrupted at any time to report an intermediate solution; the quality of the intermediate solution should improve as time progresses. Heuristics like simulated annealing and genetic algorithms are typical techniques to obtain anytime algorithms. Several notions of quality have been proposed for the intermediate solutions reported by anytime algorithms [12], including the *certainty* with which the reported solution is correct, the *accuracy* of the solution, and its *specificity*. Our progressive algorithms are similar to anytime algorithms with accuracy as the quality measure, where the accuracy is quantified by the error function. The main difference is that we focus on *provable guarantees* on the running time and the error. Indeed, the convergence function used in our framework is closely related to the *performance profile* used to analyze anytime algorithms, but in the anytime-algorithms literature the performance profile is typically determined experimentally while we require a mathematical worst-case analysis. Another difference is that we require a solution with zero error to be reported after a predetermined number of iterations. Thus progressive algorithms combine the ideas behind anytime algorithms with the rigorous analysis from algorithms research.

Our results. In Section 2 we present our theoretical framework for progressive algorithms and how to analyze them, and we establish some basic results about progressive algorithms: we show how to turn progressive algorithms whose running time is good in an amortized sense into progressive algorithms with a good running time for each individual round, and we show how to combine $(1 + \varepsilon)$ -approximation algorithms with exact algorithms to obtain efficient progressive algorithms. We also present a simple but optimal progressive sorting algorithm, which we use to discuss some limitations of our framework.

In Section 3 we turn our attention to computing the convex hull $\text{CH}(P)$ of a set P of n points in the plane. As partial solutions we allow convex polygons $C \subseteq \text{CH}(P)$, and we define the error of a partial solution C based on the relative widths of C and $\text{CH}(P)$: we define

$$\text{err}_{\text{width}}(C) := \max_{\theta} \{1 - \text{width}_{\theta}(C)/\text{width}_{\theta}(\text{CH}(P))\},$$

where $\text{width}_{\theta}(C)$ denotes the width of C in direction θ . We describe an efficient progressive convex-hull algorithm that uses a divide-and-conquer approach. The algorithm runs in at most $k := \lfloor \log n \rfloor$ rounds, each taking $O(n)$ time, and after the r -th round it reports a convex polygon $C_r \subseteq \text{CH}(P)$ with error $O(1/2^{2r})$. A similar result can be obtained using coresets [1], by applying our technique to convert approximation algorithms into progressive algorithms.

In Section 4 we study the computation of *popular places* (sometimes also called *convergence patterns* [8, 9]) in trajectory data [3]. We are given a collection $\mathcal{T} := \{T_1, \dots, T_m\}$ of trajectories describing the movement of various entities, and we want to find the places vis-

ited by many entities. We assume each trajectory is specified by a finite sequence of points, which are the locations of the entity at certain time instances. (Often timestamps are also given for the locations, but this is not relevant for our problem.) Following Benkert *et al.* [3], we define a κ -popular place, for a given parameter κ , to be a unit square that is visited by at least κ entities. In the discrete setting a popular place is thus a unit square containing input points from at least κ trajectories. The goal is now to compute the region $A_\kappa(\mathcal{T}) \subset \mathbb{R}^2$ such that a unit square σ is popular if and only if its center lies in $A_\kappa(\mathcal{T})$. Benkert *et al.* give an output-sensitive algorithm that computes $A_\kappa(\mathcal{T})$ in $O((n + |A_\kappa(\mathcal{T})|) \log n)$ time, where n is the total number of input points and $|A_\kappa(\mathcal{T})|$ denotes the combinatorial complexity of $A_\kappa(\mathcal{T})$. It follows from our results that $|A_\kappa(\mathcal{T})| = O(n\kappa)$, which is tight in the worst case. We give a progressive algorithm running in $k := \lfloor \log n \rfloor$ rounds, each taking $O(n\kappa)$ time, and with convergence function $O(1/2^r)$, where the error of a partial solution $S \subseteq A_\kappa(\mathcal{T})$ is defined as the size of the largest square $\sigma^* \subseteq A_\kappa(\mathcal{T})$ that is disjoint from the partial solution.

2 The framework

Our theoretical framework for progressive algorithms is as follows. Let D be the data set that forms the input to the problem. The set D defines a set $\mathcal{S}(D)$ of valid *partial solutions*, and we are given an *error function* $\text{err} : \mathcal{S}(D) \rightarrow \mathbb{R}_{\geq 0}$ that assigns to every partial solution a non-negative value. A *complete solution* is a partial solution S with $\text{err}(S) = 0$.

A progressive algorithm is an algorithm that works in *rounds*, reporting partial solutions with smaller error guarantees in each round, until a complete solution is reported. The guarantees are specified by a *convergence function* $f_{\text{conv}} : \{1, \dots, k\} \rightarrow \mathbb{R}_{\geq 0}$, where k is the number of rounds and $0 = f_{\text{conv}}(k) \leq f_{\text{conv}}(k-1) \leq \dots \leq f_{\text{conv}}(1)$. A *k-round progressive algorithm with convergence function f_{conv}* (with respect to error function err) is now defined as an algorithm that outputs after the r -th round, for $r = 1, \dots, k$, a partial solution S_r with $\text{err}(S_r) \leq f_{\text{conv}}(r)$.

One performance measure for progressive algorithms is given by the convergence function, which specifies how rapidly the error decreases. Another important measure is the running time, which we can quantify in two ways.

- Let $T_r(n)$ be the worst-case running time for round r . We can use $T_{\max}(n) := \max_r T_r(n)$, the *maximum time per round*, as efficiency measure. Using little time for every round ensures that the user never has to wait very long for the next partial solution to become available and makes the algorithm predictable.
- Let $T_{\leq r}(n)$ be the worst-case total running time for rounds $1, \dots, r$. We can use $T_{\max}^*(n) := \max_r T_{\leq r}(n)/r$, the *maximum amortized time per round*, as efficiency measure. This gives the possibility to have shorter rounds in the beginning—and, hence, report the first partial solutions very quickly—while later rounds can take more time.

Ideally $T_{\leq k}(n)$, the total time needed for all k rounds, is equal to the running time of the best known non-progressive algorithm for the problem at hand.

Trivially, any progressive algorithm with convergence rate $f_{\text{conv}}(r)$ that runs in k rounds and needs $T_{\text{max}}(n)$ time per round is also a progressive algorithm with convergence rate $f_{\text{conv}}(r)$ and $T_{\text{max}}^*(n) = T_{\text{max}}(n)$. Conversely, we can convert any algorithm with a good amortized time to an algorithm with a good maximum time per round, as shown next.

Theorem 1 *If there exists a progressive algorithm with convergence function $f_{\text{conv}}(r)$ that runs in k rounds and whose maximum amortized time per round is $T_{\text{max}}^*(n)$, then there also exists a progressive algorithm with convergence function $f_{\text{conv}}(r)$ that runs in at most k rounds whose maximum time per round is $O(T_{\text{max}}^*(n))$.*

Proof. Consider a progressive algorithm ALG whose maximum amortized time per round is $T_{\text{max}}^*(n)$. Start running ALG, but do not report the solutions immediately when they become available. Instead, maintain the latest partial solution computed by ALG, and after each time step $r \cdot T_{\text{max}}^*(n)$, for $r = 1, 2, \dots, k$, report the currently maintained solution. Since $r \cdot T_{\text{max}}^*(n) \geq T_{\leq r}(n)$ for any r , and f_{conv} is non-increasing, the solution reported by the new algorithm after round r is at least as good as the solution reported by the original algorithm after round r . Hence, $f_{\text{conv}}(r)$ is also a valid convergence function for the new algorithm. $\square$

A desirable property of progressive algorithms is that they are *monotone*. For example, we would like that the error is guaranteed to decrease (or at least not increase) from one round to the next. Note that this is not implied by the fact that the convergence function is non-increasing, as the convergence function only gives an upper bound on the error. In many cases we would prefer to have monotonicity not only on the error, but on the actual solution. For example, for a progressive convex-hull algorithm we would prefer to have $C_r \subseteq C_{r+1}$, where C_r denotes the convex polygon reported after round r . Thus the goal is to develop fast progressive algorithms with good convergence rate, which are also monotone.

Relation to approximation algorithms. Since progressive algorithms report solutions that approximate the complete solution increasing well, there is a clear connection to approximation algorithms and, in particular, PTASs. Suppose we have an approximation algorithm $\text{ALG}(\varepsilon)$ that computes a solution with error ε . Then we can obtain a progressive algorithm by running $\text{ALG}(\varepsilon)$ with increasingly smaller values of ε , each time starting from scratch. This may seem very inefficient, and it is somewhat against the spirit of progressive algorithms. (Furthermore, it does not automatically give a monotone algorithm.) Depending on the running time of the approximation algorithm, however, one can often choose a sequence of values for ε that gives a good running time per round, as shown in the next theorem. A similar technique was also used by Russell and Zilberstein [11] to convert so-called contract algorithms to anytime algorithms. We will need that $T_{\text{apx}}(\varepsilon, n)$ is at least linear in $1/\varepsilon$, meaning that $T_{\text{apx}}(\varepsilon_1, n) + T_{\text{apx}}(\varepsilon_2, n) \leq T_{\text{apx}}(\frac{\varepsilon_1 \varepsilon_2}{\varepsilon_1 + \varepsilon_2}, n)$ for any $n, \varepsilon_1, \varepsilon_2$.

Theorem 2 *Suppose we have an approximation algorithm that can compute, for any given $\varepsilon > 0$, a solution with error ε in time $T_{\text{apx}}(\varepsilon, n)$, where $T_{\text{apx}}(\varepsilon, n)$ is at least linear in $1/\varepsilon$. Suppose furthermore that we have an algorithm that computes the exact solution in time $T_{\text{exact}}(n)$. Then for any $k > 1$ there is a k -round progressive algorithm with convergence function $f_{\text{conv}}(r) = 1/2^r$, and $T_{\text{max}}(n) = O\left(\frac{1}{k} \cdot (T_{\text{apx}}(1/2^{k-1}, n) + T_{\text{exact}}(n))\right)$.*

Proof. In the first $k - 1$ rounds we run the approximation algorithm, using $\varepsilon = 1/2^r$ in round r . In round k we run the exact algorithm. This way we obtain a progressive algorithm with $f_{\text{conv}}(r) = 1/2^r$. For $r < k$ we have

$$T_{\leq r}(n) = \sum_{i=1}^r T_{\text{apx}}(1/2^i, n) \leq 2 \cdot T_{\text{apx}}(1/2^r, n),$$

because $T_{\text{apx}}(\varepsilon, n)$ is at least linear in $1/\varepsilon$. Moreover,

$$T_{\leq k}(n) \leq 2 \cdot T_{\text{apx}}(1/2^{k-1}, n) + T_{\text{exact}}(n).$$

Using once more that $T_{\text{apx}}(\varepsilon, n)$ is at least linear in $1/\varepsilon$, we see that

$$\begin{aligned} T_{\max}^*(n) &= \max_r T_{\leq r}(n)/r \\ &\leq \frac{2 \cdot T_{\text{apx}}(1/2^{k-1}, n)}{(k-1)} + \frac{T_{\text{exact}}(n)}{k}. \end{aligned}$$

The result now follows from Theorem 1. □

Remark. Theorem 2 shows that in many cases it is possible to obtain a progressive algorithm if a $(1 + \varepsilon)$ -approximation algorithm is available. Note, however, that this does not necessarily lead to a monotone algorithm: there is no guarantee that the solution computed when running the approximation algorithm with error bound $\varepsilon/2$ is a refinement of the solution computed with error bound ε . (Indeed, there is not even a guarantee that the actual error is smaller.)

A simple example: a progressive sorting algorithm. Let X be a set of n distinct numbers that we want to sort. First we have to define our partial solutions and error function. A natural partial solution for sorting is a permutation of the numbers in X . The error of a permutation $\pi(X)$ can be defined in several ways. We will use the maximum displacement of any element $x \in X$ with respect to its correct position in the sorted order. Thus, if $\text{rank}(x)$ denotes the rank of x in X , and $\text{pos}(\pi, x)$ denotes the position of x in permutation $\pi(X)$, then

$$\text{err}_{\text{pos}}(\pi) = \max_{x \in X} |\text{rank}(x) - \text{pos}(\pi, x)|.$$

Note that $\text{err}_{\text{pos}}(\pi) = 0$ when π is a complete solution (that is, when $\pi(X)$ is sorted), as required from an error function. Our error measure is similar to *Spearman's footrule metric* [5], except that we use the maximum of the displacements rather than the sum. Standard sorting algorithms such as *QuickSort* or *MergeSort* do not have a good convergence rate in their original recursive implementation: both algorithms already spend $O(n \log n)$ time on their first recursive call on half of the elements, and after this call the error can still be $\Omega(n)$. However, it is straightforward to adapt *QuickSort* (with linear-time median finding to select the pivot) to obtain a good—in fact, optimal—convergence rate. To this end, consider the recursion tree of *QuickSort*. In a standard implementation the recursion tree is traversed in a depth-first manner. We change this to a breadth-first traversal. In

other words, in each round r we have 2^r bins of size at most $n/2^r$, where all elements in the i -th bin are smaller than those in the $(i + 1)$ -th bin, and we partition each bin around its median. Giesen *et al.* [6] studied this algorithm for Spearman’s footrule metric, and proved that it is optimal: any algorithm (even a randomized one) needs to do $\Omega(nt)$ comparison to generate a permutation whose error in Spearman’s footrule metric is guaranteed to be at most n^2/t . (In fact, they even obtained tight bounds on the exact number of comparisons.) This immediately implies that the breadth-first version of *QuickSort* is also an optimal progressive algorithm. The following theorem is essentially an interpretation of the results of Giesen *et al.* in the framework of progressive algorithms.

Theorem 3 *There is a progressive algorithm for sorting a set of n numbers that consists of $k := \lfloor \log n \rfloor$ rounds, each taking $O(n)$ time, and with convergence function $f_{\text{conv}}(r) = n/2^r - 1$ with respect to the error function $\text{err}_{\text{pos}}(\pi)$ defined above. This convergence rate is asymptotically optimal: any comparison-based algorithm computing a permutation with error at most $n/2^r - 1$ needs to do $\Omega(rn)$ comparisons in the worst case.*

One way to define monotonicity for the sorting problem is to require that, as the subsequent permutations are generated by the algorithm, the displacement $|\text{pos}(\pi, x) - \text{rank}(x)|$ of each element x is non-increasing. The *QuickSort* variant sketched above is not monotone in this sense. It would be interesting to develop a monotone progressive sorting algorithm with the same optimal convergence rate.

Discussion. Theorem 3 states that the *QuickSort* variant we sketched has convergence function $f_{\text{conv}}(r) = n/2^r - 1$, that each round takes $O(n)$ time, and that any comparison-based algorithm computing a permutation with error at most $n/2^r - 1$ needs $\Omega(rn)$ comparisons in the worst case. Nevertheless it is possible, for any constant c , to get an algorithm that runs in $\Theta(\log n)$ rounds, uses $O(n)$ time per round, and has convergence function $f_{\text{conv}}(r) = n/2^{cr} - 1$. To this end, simply redefine a round to consist of c rounds of the original algorithm. As long as c is a constant, the running time per round is still $O(n)$. The total number of rounds decreases by a factor c . Intuitively, the new solution should be worse than the original solution, because we would like to get as many intermediate solutions as long as it does not increase the total time. However, it seems hard to formalize this in our framework: the fact that the time per round is given asymptotically makes it possible to play tricks like combining multiple rounds into a single round in order to obtain a seemingly better convergence rate. By also adding a number of “dummy rounds” at the end, we would get an algorithm with the same number of rounds as the original algorithm—that is, the algorithm before we combined multiple rounds into single rounds—while the error at any given round is smaller. Thus, when the convergence rate is exponential, our framework cannot really distinguish between different constant factors in the exponent.

3 Convex hulls

Let $P := \{p_1, \dots, p_n\}$ be a set of n points in the plane. We want to develop a progressive algorithm to compute $\text{CH}(P)$, the convex hull of P . We define a partial solution to be a

convex polygon $C \subseteq \text{CH}(P)$; our algorithm actually uses polygons C that are the convex hull of a subset of P . Our error function is a measure that is also used when determining the quality of *coresets* [1] for the extent measure. For a convex polygon C and some $0 \leq \theta < \pi$, let $\text{width}_\theta(C)$ denote the *width of C in direction θ* , that is, $\text{width}_\theta(C)$ is the distance between the two tangent lines of C making an angle $\theta + \pi/2$ with the positive x -axis. The error function we consider is

$$\text{err}_{\text{width}}(C) = \max_{\theta} \left\{ 1 - \frac{\text{width}_\theta(C)}{\text{width}_\theta(\text{CH}(P))} \right\}.$$

We describe a progressive algorithm that gives a good convergence rate with respect to this error function. It combines ideas from the construction of coresets with the *QuickHull* algorithm [2].

The algorithm works in k rounds, for some $k \leq \log n$, and produces a sequence $C_1, C_2, \dots, C_k$ of convex polygons such that $C_1 \subset \dots \subset C_k = \text{CH}(P)$. Each convex polygon C_r is the convex hull of a subset $P_r \subset P$ such that all points in P_r are *extreme*, that is, vertices of the final convex hull. A first idea is to let P_r consist of the extreme points in the directions $i \cdot (2\pi/2^r)$, for $i = 0, 1, \dots, 2^r - 1$. (We identify the direction indicated by a vector $\vec{d}$ with the counter-clockwise angle that $\vec{d}$ makes with the positive x -axis.) One can show that $\text{err}_{\text{width}}(C_r) = O(1/2^i)$ when using these extreme points. However, the number of rounds is not necessarily logarithmic with this scheme and as a result the total running time is not guaranteed to be $O(n \log n)$. Moreover, the convergence rate is suboptimal. Next we describe how to overcome these problems.

We start by a preprocessing step (which is considered to be part of the first round of our progressive algorithm) that is also used in the construction of coresets. It ensures the width of $\text{CH}(P)$ is roughly the same in all directions. More precisely, we will make sure that $\text{CH}(P)$ is α -fat, for some constant $0 < \alpha < 1$, in the following sense: there are concentric circles C_{in} and C_{out} such that $C_{\text{in}} \subset \text{CH}(P) \subset C_{\text{out}}$ and $\text{radius}(C_{\text{in}}) = \alpha \cdot \text{radius}(C_{\text{out}})$.

Lemma 4 [1] *For any set P of n points in the plane that are not all collinear, there is an affine transformation τ such*

- (i) $\tau(P)$ is α -fat, and
- (ii) for any $Q \subseteq P$ we have

$$\frac{\text{width}_\theta(\text{CH}(Q))}{\text{width}_\theta(\text{CH}(P))} = \frac{\text{width}_\theta(\text{CH}(\tau(Q)))}{\text{width}_\theta(\text{CH}(\tau(P)))}.$$

Moreover, such a transformation, together with the circles C_{in} and C_{out} as defined above, can be computed in $O(n)$ time.

From now on we assume that C_{in} and C_{out} are centered at the origin. Recall that the intermediate solution we compute in step r will be the convex hull of a set $P_r \subseteq P$. Next we specify which points P_r contains. More precisely, we specify which points we include in P_r to guarantee that the error of $\text{CH}(P_r)$ is small. The set P_r will also contain some additional

points to guarantee that our progressive algorithm runs in a logarithmic number of rounds, as explained later.

For an integer m , let $\mathcal{D}(m) := \{i \cdot (2\pi/m) : 0 \leq i < m\}$ be m equally-spaced directions. For a direction θ , let $p(\theta) \in P$ denote the point that is extreme in direction θ . Furthermore, let $\vec{\rho}(\theta)$ denote the ray with direction θ emanating from the origin, and let $e(\theta)$ denote the edge of $\text{CH}(P)$ that is intersected by $\vec{\rho}(\theta)$. To simplify the description we assume that both $p(\theta)$ and $e(\theta)$ are unique; it is easy to adapt the solution to the case where there are multiple extreme points in a given direction, or $\vec{\rho}(\theta)$ passes through a vertex. Now define:

$$P(m) := \{p(\theta) : \theta \in \mathcal{D}(m)\} \cup \{p : p \text{ is an endpoint of } e(\theta) \text{ for some } \theta \in \mathcal{D}(m)\}.$$

The following lemma is similar to a lemma by Hershberger and Suri [7], although the set $P(m)$ is defined differently. For the purpose of giving a progressive algorithm our definition of $P(m)$ has the advantage that $P(m)$ can be easily extended as m increases.

Lemma 5 *Let $m \geq 4$ and $C := \text{CH}(P(m))$. Then we have $\text{err}_{\text{width}}(C) = O(1/m^2)$.*

Proof. Consider any direction θ . Let $p := p(\theta)$ the extreme point of P in direction θ and let $\bar{p}$ be the extreme point of $P(m)$ in direction θ . Let O denote the origin, which is also the center of the circles C_{in} and C_{out} such that $C_{\text{in}} \subset \text{CH}(P) \subset C_{\text{out}}$ and $\text{radius}(C_{\text{in}}) = \alpha \cdot \text{radius}(C_{\text{out}})$. Let ℓ_p , $\ell_{\bar{p}}$, and ℓ_O denote the lines through p , $\bar{p}$, and O , respectively, that are orthogonal to the direction θ . To prove the lemma it suffices to prove that $\text{dist}(\ell_p, \ell_{\bar{p}})/\text{dist}(\ell_p, \ell_O) = O(1/m^2)$.

Let $\theta_i := i \cdot (2\pi/m)$, for $0 \leq i < m$, denote the directions in $\mathcal{D}(m)$. Define i^* such that θ_{i^*} is the smallest direction greater than or equal to θ . Furthermore, define j such that $p(\theta)$ lies in $\text{cone}(\theta_j, \theta_{j+1})$, the cone defined by the rays $\vec{\rho}(\theta_j)$ and $\vec{\rho}(\theta_{j+1})$. We distinguish two cases.

Case (i): the point $p(\theta_{i^})$ lies in $\text{cone}(\theta_j, \theta_{j+1})$.* Let $q := p(\theta_{i^*})$ and let ℓ_q denote the line through q orthogonal to θ ; see Fig. 1 for an illustration. Since $q \in \mathcal{D}(m)$, we know that $\text{dist}(\ell_p, \ell_{\bar{p}}) \leq \text{dist}(\ell_p, \ell_q)$ and so we proceed to bound $\text{dist}(\ell_p, \ell_q)/\text{dist}(\ell_p, \ell_O)$, which equals

Figure 1: Illustration for case (i) in the proof of Lemma 5.

Figure 2: Illustration for case (ii) in the proof of Lemma 5.

$\text{dist}(\ell_p, \ell_q)/|Ot|$ where $t := \ell_p \cap \vec{\rho}(\theta)$. Let $\beta := \angle qso$, where $s := \ell_q \cap Op$, and note that $\beta = \angle tps$. Then $\sin \beta = |Ot|/|Op|$. Since $|Ot| \geq \text{radius}(C_{\text{in}})$ and $|Op| \leq \text{radius}(C_{\text{out}})$, we have $|Ot|/|Op| \geq \alpha = \Omega(1)$ so $\beta = \Omega(1)$. Thus, if ϕ denotes the angle $\angle qOp$, then

$$|qs| = \sin \phi \cdot |Oq| / \sin \beta = O(\sin \phi \cdot |Oq|).$$

Hence, if ϕ' denotes the angle $\angle sqp$ then

$$\begin{aligned} \text{dist}(\ell_p, \ell_q) &< |ps| &= \sin \phi' \cdot |qs| / \sin(\beta - \phi') \\ &= O(\sin \phi \cdot \sin \phi' \cdot \sin(\beta - \phi') \cdot |Oq|). \end{aligned}$$

Now we observe that $\phi \leq \theta_{j+1} - \theta_j = 2\pi/m$ and that $\phi' \leq |\theta_{i*} - \theta| < 2\pi/m$. Since $\beta = \Omega(1)$, as argued above, we conclude that $\text{dist}(\ell_p, \ell_q) = O(|Oq|/m^2)$. As

$$|Oq| < |Op| \leq (1/\alpha)|Ot| = O(|Ot|),$$

this completes the proof for Case (i).

Case (ii): the point $p(\theta_{i})$ does not lie in $\text{cone}(\theta_j, \theta_{j+1})$.* Assume without loss of generality that $p(\theta_{i*})$ lies clockwise of $\vec{\rho}(\theta_{j+1})$. Let $e(\theta_{j+1})$ be the edge of $\text{CH}(P)$ that is intersected by $\vec{\rho}(\theta_{j+1})$; see Fig. 2. Define q to be the endpoint of $e(\theta_{j+1})$ that lies in $\text{cone}(\theta_j, \theta_{j+1})$ and, as before, define ℓ_q to be the line orthogonal to ϕ through q . Then the angle that ℓ_q makes with qp is at most $|\theta_{i*} - \theta|$. This implies that, with our new definition of q , we can now follow the proof for Case (i). $\square$

We now describe our progressive convex-hull algorithm. As mentioned, the partial solution C_r reported after round r is the convex hull of a subset $P_r \subseteq P$. We will ensure that $P(2^{r+1}) \subseteq P_r$, so $\text{err}_{\text{width}}(C_r) = O(1/2^{2r})$ by Lemma 5. Our algorithm is similar to *QuickHull* [2], except that we compute the vertices added in each round in a different way so that we get fast convergence.

For $r = 1$ we use $P_r = P(4)$. To compute $P(4)$ we need to find, for $\theta = 0, \pi/2, \pi, 3\pi/2$, the extreme points in direction θ , which can trivially be done in $O(n)$ time. We also need to find the edges of $\text{CH}(P)$ intersected by the rays $\vec{\rho}(\theta)$. The latter can be done in $O(n)$ time by linear programming [10], where the points in P induce constraints. For example,

if $\vec{\rho}(\theta)$ is the positive y -axis, then we need to find the line with minimum y -intercept that has no points from P above it. We then compute $C_1 = \text{CH}(P_1)$ in $O(1)$ time.

In a generic round r of the algorithm we are given the polygon C_{r-1} and for each edge e of C_{r-1} a set $P(e)$ defined as follows. Consider the rays from the origin through the vertices of C_{r-1} . These rays partition the plane into cones, each corresponding to a unique edge of C_{r-1} . We denote the cone corresponding to an edge e of C_{r-1} by $\text{cone}(e)$. Now $P(e)$ contains the points from P that lie in $\text{cone}(e)$ but outside C_{r-1} . These are the points that may still appear as vertices of $\text{CH}(P)$ in between the endpoints of e . (Note that the sets $P(e)$ for the edges e of C_1 can easily be found in $O(n)$ time.) We now proceed as follows.

1. Determine for each direction $\theta \in \mathcal{D}(2^{r+1}) \setminus \mathcal{D}(2^r)$, that is, for each new direction we need to handle, the edge e of C_{r-1} such that $p(\theta) \in \text{cone}(e)$. This can be done in $O(2^r) = O(n)$ time in total by “walking around” C_r . Note that for each edge e such that $P(e)$ is non-empty, there can be at most one direction θ_e such that $p(\theta_e) \in \text{cone}(e)$, because in between any two directions in $\mathcal{D}(2^{r+1}) \setminus \mathcal{D}(2^r)$ there is a direction in $\mathcal{D}(2^r)$.
2. Determine for each direction $\theta \in \mathcal{D}(2^{r+1}) \setminus \mathcal{D}(2^r)$ the edge e of C_{r-1} such that $\text{cone}(e)$ contains the ray $\vec{\rho}(\theta)$. Again, if $P(e) \neq \emptyset$ there can be at most one such ray $\vec{\rho}_e$.
3. For each edge e of C_{r-1} , find a ray $\vec{\rho}'_e$ from the origin with at most $|P(e)|/2$ points from $P(e)$ on either side. This takes $O(|P(e)|)$ time using a standard median-finding algorithm.
4. Finally, for each edge e with $P(e) \neq \emptyset$ do the following.
 - If in Step 1 we found a direction θ_e with $p(\theta_e) \in \text{cone}(e)$, then compute $p(\theta_e)$ by computing the extreme point of $P(e)$ in direction θ_e .
 - If in Step 2 we found a ray $\vec{\rho}_e$ contained in $\text{cone}(e)$, then compute the edge of $\text{CH}(P)$ that intersects $\vec{\rho}_e$. This can be done by linear programming, where the points in $P(e)$ together with the endpoints of e induce the constraints.
 - If in Step 3 we found a ray $\vec{\rho}'_e$ contained in $\text{cone}(e)$, then compute the edge of $\text{CH}(P)$ that intersects $\vec{\rho}'_e$, again using linear programming.

This gives up to five new vertices of the convex hull, which are inserted in between the endpoints of e to obtain the convex chain replacing e in the new intermediate solution C_r . It remains to compute the sets $P(e')$ for the edges in this chain, which can be done in $O(|P(e)|)$ time by checking for each point in $P(e)$ to which set $P(e')$ it belongs, if any.

Theorem 6 *There is a progressive algorithm to compute the convex hull of a set of n points in the plane that runs in at most $k := \lfloor \log n \rfloor$ rounds, each taking $O(n)$ time, with convergence function $f_{\text{conv}}(r) = O(1/2^{2^r})$ with respect to the error function $\text{err}_{\text{width}}(C)$ defined above. The algorithm is monotone in the sense that, if C_r denotes the intermediate solution reported after round r , we have $C_r \subseteq C_{r+1}$ for all $1 \leq r < k$.*

Proof. The bound on the convergence rate follows from Lemma 5 and the fact that $P(2^{r+1}) \subset P_r$. The time spent in each round r is $O(n + \sum_e |P(e)|)$, where the sum is

taken over all edges e of C_{r-1} . Since $\sum_e |P(e)| < n$, the time per round is $O(n)$. The “median ray” selected in Step 3 above makes sure that $|P(e)| < n/2^r$ for any edge e of C_r , so the total number of rounds is at most $\log n$. $\square$

Discussion. (1) In progressive divide-and-conquer algorithms we need to do the divide-step in such a way that the error is kept under control, but we also need the subproblems to decrease rapidly in size to achieve a fast running time. In the convex-hull algorithm above, the error was controlled by making sure that $P(2^{r+1}) \subset P_r$, while the size of the subproblems was controlled using the “median ray” found in Step 3. For our progressive sorting algorithm, the median could play both roles, as the error function is closely related to size of the subproblems.

(2) Our convex-hull algorithm is based on Lemma 5, which gives us a set $P(m)$ with error $O(1/m^2)$. Thus, after round r we get an error of $O(1/2^{2r})$. With less effort, namely by only putting the extreme points in directions in $\mathcal{D}(m)$ into $P(m)$, we can obtain an error of $O(1/m)$, giving us an error of $O(1/2^r)$ after round r . This seems to give a worse result, but we can use the trick already mentioned earlier: we can merge pairs of rounds into a single round to again obtain an error of $O(1/2^{2r})$. By merging even more rounds, we could even obtain a seemingly better error, namely $O(1/2^{cr})$, for any constant c .

(3) An alternative to obtain a progressive convex-hull algorithm is to use our general technique to convert approximation algorithms into progressive algorithms. To this end we use the coresset algorithm of Chan [4], who showed that for any set of n points in the plane and any $\varepsilon > 0$, one can compute an ε -coreset of size $O(\sqrt{1/\varepsilon})$ in time $O(n + \sqrt{1/\varepsilon})$. If we compute the convex hull of the coresset in time $O(\sqrt{1/\varepsilon} \log(1/\varepsilon))$, we get a convex polygon with error $O(\varepsilon)$. By setting $k := \lfloor \log n \rfloor$ and applying Theorem 2—note that an optimal planar convex-hull algorithm gives $T_{\text{exact}}(n) = O(n \log n)$ —we get a k -round progressive algorithm using $O(n)$ time per round and with convergence function $f_{\text{conv}}(r) = O(1/2^r)$. Again, the convergence rate can be boosted by merging rounds. (A closer inspection of the Chan’s algorithm shows that the resulting progressive algorithm is in fact also monotone.)

4 Computing popular places from trajectory data

Let $\mathcal{T} := \{T_1, T_2, \dots, T_m\}$ be a set of trajectories in the plane, each specified by a finite sequence of points, and let κ be a parameter with $2 \leq \kappa \leq m$. With a slight abuse of notation we also use $\mathcal{T}$ to refer to the union of the point sets specifying the trajectories (where each point is labeled with its trajectory), and we set $n := |\mathcal{T}|$. A unit square is κ -popular if it contains points from at least κ trajectories. We want to develop a progressive algorithm to compute the κ -popular region $A_\kappa(\mathcal{T})$, that is, the set of all points $q \in \mathbb{R}^2$ such that σ_q , the axis-aligned unit square centered at q , is κ -popular; see Fig. 3 for an illustration.

As partial solutions we allow subsets $S \subseteq A_\kappa(\mathcal{T})$. Let $\mathcal{B}$ be a smallest enclosing square of $\mathcal{T}$ extended by half a unit on each side, so that $A_\kappa(\mathcal{T}) \subset \mathcal{B}$. We now define

$err_{pop}(S)$, the error of a partial solution S , as $size(\sigma^*)/size(\mathcal{B})$, where σ^* is the largest square contained in $A_\kappa(\mathcal{T})$ with $S \cap \sigma^* = \emptyset$, and $size(\cdot)$ denotes the side length of a square.

In the following we will use $\Sigma(T_i) := \{\sigma_p : p \in T_i\}$ to denote the set of unit squares (or intervals in the 1D case) centered at the points of T_i , and similarly we define $\Sigma(\mathcal{T}) := \{\sigma_p : p \in \mathcal{T}\}$. Note that a point $q \in \mathbb{R}^2$ is κ -popular if and only if it is contained in squares σ_p coming from at least κ sets $\Sigma(T_i)$.

Computing 1D popular regions. As a warm-up exercise, we study the 1D version of the problem. In this case the region $A_\kappa(\mathcal{T})$ can be computed non-progressively in $O(n \log n)$ time: sort the endpoints of the intervals in $\Sigma(\mathcal{T})$, and then sweep from left to right while maintaining an array $M[1..m]$ such that $M[i]$ is equal to the number of intervals from $\Sigma(T_i)$ containing the sweep point. To turn this into a progressive algorithm we adapt the progressive *QuickSort* algorithm from Section 2, as follows.

Recall that in each round of the progressive *QuickSort* algorithm we have a permutation of the input points. In this permutation, all points that have already been chosen as pivot are at their correct position, while the other points are positioned correctly with respect to the pivots—they are in the right *bin*. We then pick the median of each bin as pivot and partition the bin into two smaller bins. We adapt this algorithm as follows.

First of all, we pick two pivots in each bin B : besides the median of the points in B we also pick the point $(x_{\min}(B) + x_{\max}(B))/2$, where $[x_{\min}(B), x_{\max}(B)]$ is the interval corresponding to B . This second pivot, which we call the *geometric pivot*—note that the geometric pivot is not an endpoint of an interval in $\Sigma(\mathcal{T})$ —is used to control the error. We then partition B into three smaller bins, based on the two pivots.

Second, we compute the partial solution S_r that we report in the current round r . We do this by sweeping, as in the non-progressive algorithm: while we sweep we maintain for each trajectory the number of intervals in $\Sigma(\mathcal{T})$ containing the sweep point. The order in which we encounter the endpoints of the intervals in $\Sigma(\mathcal{T})$ is not completely left-to-right, as the points within a bin are not sorted yet. However, the pivots are at their correct position, so whenever we sweep over a pivot we can correctly determine whether it is κ -popular. We therefore call these points *safe stops* and we report the set of all safe stops that are κ -popular as our partial solution S_r . Note that $err_{pop}(S_r)$ is bounded by the maximum

Figure 3: Popular region for $\kappa = 2$ defined by three given trajectories.

length of the interval corresponding to a bin, and note that in the last round we can easily compute the complete solution by sweeping. This leads to the following theorem.

Theorem 7 Let $\mathcal{T} := \{T_1, \dots, T_m\}$ be a set of 1D trajectories with n points in total, and let κ be a parameter. Then there exists a progressive algorithm to compute the κ -popular region that runs in at most $k := \log n$ rounds, each taking $O(n)$ time, with convergence function $f_{\text{conv}}(r) = 1/2^r$ with respect to the error function $\text{err}_{\text{pop}}(S)$ defined above. The algorithm is monotone in the sense that, if S_r denotes the solution reported after round r , we have $S_r \subseteq S_{r+1}$ for all $1 \leq r < k$.

Computing 2D popular regions within a unit square. We now return to the 2D problem. We start by describing a progressive algorithm that computes the part of $A_\kappa(\mathcal{T})$ within a given unit square U . Later we show how to solve the general problem, using this algorithm as a subroutine. We assume without loss of generality that each $\sigma_p \in \Sigma(\mathcal{T})$ intersects U . To simplify the presentation we also make the following non-degeneracy assumption: each σ_p has exactly one corner in the interior of U , and no two corners have the same x - or y -coordinate.

An algorithm for sorted data. Assume we are given two lists storing the corners inside U of the squares from $\Sigma(\mathcal{T})$: a list $\mathcal{L}_x$ sorted on x -coordinate, and a list $\mathcal{L}_y$ sorted on y -coordinate. We can then compute $A_\kappa(\mathcal{T})$ in $O(n\kappa)$ time with a plane-sweep algorithm, as follows.

The sweep line ℓ is horizontal and moves downward, halting at every corner of a square $\sigma_p \in \Sigma(\mathcal{T})$ inside U . Let $\Sigma_{\text{bl}}(T_i)$ denote the set of squares in $\Sigma(T_i)$ whose bottom-left corner is inside U . Define $\Sigma_{\text{br}}(T_i)$, $\Sigma_{\text{tl}}(T_i)$, and $\Sigma_{\text{tr}}(T_i)$ similarly, but then for the bottom-right, top-left, and top-right corners. Next we describe the status information we maintain during the sweep for $\Sigma_{\text{bl}}(\mathcal{T}) := \bigcup \Sigma_{\text{bl}}(T_i)$; the status information for $\Sigma_{\text{br}}(\mathcal{T})$, $\Sigma_{\text{tl}}(\mathcal{T})$, and $\Sigma_{\text{tr}}(\mathcal{T})$ is similar.

Consider a fixed T_i , and consider all squares in $\Sigma_{\text{bl}}(T_i)$ that intersect the sweep line ℓ . We call the leftmost of these squares the *active square* (from the squares in $\Sigma_{\text{bl}}(T_i)$). Clearly we do not need to consider non-active squares when computing the popular region along ℓ . During the sweep we maintain the following information for $\Sigma_{\text{bl}}(\mathcal{T})$; see also Fig. 4.

- Let $\mathcal{L}_{\text{bl}}(\mathcal{T})$ be a list storing all squares from $\Sigma_{\text{bl}}(\mathcal{T})$, sorted from left to right according to the x -coordinate of their bottom-left endpoint. This list does not change during the sweep. We maintain an array $\text{PTR}_{\text{bl}}[1..\kappa]$ of pointers into $\mathcal{L}_{\text{bl}}(\mathcal{T})$. More precisely, this array stores pointers to the κ leftmost active squares in $\mathcal{L}_{\text{bl}}(\mathcal{T})$, ordered from left to right. Thus $\text{PTR}_{\text{bl}}[1]$ points to the leftmost active square, $\text{PTR}_{\text{bl}}[2]$ to the second-leftmost active square, and so on. We call these κ leftmost active squares *relevant*. Any point on ℓ covered by the square (pointed to by) $\text{PTR}_{\text{bl}}[\kappa]$ is covered by squares from at least κ different sets $\Sigma_{\text{bl}}(T_i)$, so the relevant squares are the only squares from $\Sigma_{\text{bl}}(\mathcal{T})$ we need to consider when computing the popular region along ℓ .

Figure 4: Information maintained for $\Sigma_{\text{bl}}(\mathcal{T})$ during the sweep. Only active squares are drawn, and the squares are labeled with the set T_i they originate from. In this example, $\kappa = 3$.

- We have an array $\text{REL}_{\text{bl}}[1..m]$ where $\text{REL}_{\text{bl}}[i] = j$ if $\text{PTR}_{\text{bl}}[j]$ points to a square in $\Sigma_{\text{bl}}(T_i)$ and $\text{REL}_{\text{bl}}[i] = \text{NIL}$ if $\Sigma_{\text{bl}}(T_i)$ does not contain a relevant square.

This information is sufficient to compute $A_\kappa(\mathcal{T}) \cap \ell$ in $O(\kappa)$ time. To this end we first merge the four arrays $\text{PTR}_{\text{bl}}[1..\kappa]$, $\text{PTR}_{\text{br}}[1..\kappa]$, $\text{PTR}_{\text{tl}}[1..\kappa]$, and $\text{PTR}_{\text{tr}}[1..\kappa]$. This gives us a list with the relevant squares from all four types, sorted according to the intersections of their boundaries with ℓ . We scan this list—this is similar to the scan for the 1D problem—to obtain the popular region along ℓ . (We should actually report the popular region in between the sweep line ℓ at its current and previous position, not just the popular region along ℓ , but we omit the details.)

It remains to describe how we initialize the status information, and how we maintain it as we sweep. We focus on the information for $\Sigma_{\text{bl}}(\mathcal{T})$. Initialization in time $O(n\kappa)$ is straightforward, so we focus on the maintenance.

Suppose the sweep line ℓ halts at the bottom-left corner of a square $\sigma_p \in \Sigma_{\text{bl}}(T_i)$. We only need to do something when σ_p is relevant, that is, when $\text{REL}_{\text{bl}}[i] = j \neq \text{NIL}$. We then start moving the pointer $\text{PTR}_{\text{bl}}[j]$ to the right, until we either encounter a square $\sigma_{p'} \in \Sigma_{\text{bl}}(T_i)$ —in this case we are done—or $\text{PTR}_{\text{bl}}[j]$ becomes equal to $\text{PTR}_{\text{bl}}[j + 1]$. In the latter case we set $\text{PTR}_{\text{bl}}[j] := \text{PTR}_{\text{bl}}[j + 1]$, and we start walking with $\text{PTR}_{\text{bl}}[j + 1]$. This continues until we find a square $\sigma_{p'} \in \Sigma_{\text{bl}}(T_i)$ and we are done, or we start walking with $\text{PTR}_{\text{bl}}[\kappa]$. From then on we also need to check for each square $\sigma_{p'}$ we encounter whether the set $\Sigma_{\text{bl}}(T_{i'})$ it comes from already has a relevant square. If not, then we let $\text{PTR}_{\text{bl}}[\kappa]$ point to $\sigma_{p'}$. Updating the array REL_{bl} during these operations is a matter of simple bookkeeping.

During the maintenance, the pointers in $\text{PTR}_{\text{bl}}[1..\kappa]$ only move to the right. Hence, the total time for moving pointers is bounded by $O(n\kappa)$. The time needed to report the solution whenever the sweep line halts is $O(\kappa)$, so in total this also takes $O(n\kappa)$.

This finishes the description of how we maintain the information for $\Sigma_{\text{bl}}(\mathcal{T})$. Main-

taining the information for $\Sigma_{\text{br}}(\mathcal{T})$ is symmetrical. For $\Sigma_{\text{tl}}(\mathcal{T})$ and $\Sigma_{\text{tr}}(\mathcal{T})$ the situation is slightly different, since squares start to (rather than stop to) intersect the sweep line when encountered, but this makes it actually easier. Overall we thus spend $O(n\kappa)$ time, as claimed.

A progressive algorithm for unsorted data. We use the same basic idea as in the 1D case, namely to combine our progressive *QuickSort* algorithm with the algorithm for sorted data. Thus we sort the corners inside U of the squares from $\Sigma(\mathcal{T})$ both on x -coordinate and on y -coordinate, using progressive *QuickSort*. As in the 1D case, we not only use the median of the bins as pivots, but also geometric pivots. Thus after each round of the progressive *QuickSort* we have two partially sorted lists, $\mathcal{L}_x$ and $\mathcal{L}_y$, which contain not only (corners of) the squares in $\Sigma(\mathcal{T})$ but also certain geometric pivots. We now essentially run the algorithm we described above using the lists $\mathcal{L}_x$ and $\mathcal{L}_y$, pretending that they are correctly sorted. Next we describe the modifications that are needed, and we analyze the error in the partial solutions we report.

First consider the fact that $\mathcal{L}_y$ is only partially sorted. Recall that our algorithm is a sweep-line algorithm, where at each event we update the status information and we compute the popular region along the sweep line using a 1D scan. The former step—maintaining the status information at each event—is still done in the same way as before. The latter step—the 1D scan along the sweep line—is now only performed at the *safe stops*, that is, at the points that have been used as pivots and, hence, are correctly positioned in $\mathcal{L}_y$. This means that we have correct knowledge of which squares intersects the sweep line ℓ .

Now consider the 1D scan that we do at a safe stop in $\mathcal{L}_y$. Since the ordering of the squares along ℓ is not completely correct, some relevant squares might be considered irrelevant by our algorithm, and vice versa. We call the squares that have been identified as relevant based on the partially sorted list $\mathcal{L}_x$ *quasi-relevant*. Note that (the x -coordinates of the corners of) quasi-relevant squares need not be safe stops and vice versa. Our new 1D scan will halt at the quasi-relevant squares (more precisely, at points where a vertical edge of such a square intersect the sweep line ℓ) and at safe stops from the list $\mathcal{L}_x$. Halting at the safe stops is necessary to keep the error under control. Unfortunately the number of safe stops grows in every round and can be much larger than the $O(\kappa)$ quasi-relevant squares. We overcome this by only halting at safe stops that are neighbors of (the x -coordinates of) quasi-relevant squares. In other words, if we imagine the safe stops to partition the sweep line into bins, then we only stop at those safe stops that are an endpoint of a bin containing at least one quasi-relevant square. Finding these bins can be done before the sweep starts in $O(n)$ time in total for all squares.

Now, when we do the 1D scan we maintain the popularity of the points encountered during the scan, as usual. When we arrive at a safe stop and its popularity is at least κ , we report it as part of our partial solution. There are also other locations on the sweep line of which the popularity is known. In particular, if there is no quasi-relevant square in between two safe stops then the popularity is the same for all points in the interval. Thus, if halt at a popular safe stop during the scan and the next safe stop where we halt is also popular, then we report the entire interval between these safe stops. (This is necessary

because we are only stopping at safe stops surrounding quasi-relevant squares. Thus the distance between consecutive points where the scan halts could become large because we are skipping certain safe stops, which could lead to a large error.)

The correctness of this approach is based on the following lemma.

Lemma 8 *Suppose the sweep line ℓ halts at a safe stop in $\mathcal{L}_y$. Let p_l and p_r be consecutive safe stops in $\mathcal{L}_x$. Then for each of the four corner types, the number of relevant squares whose corner in U has x -coordinate in the range $[p_l, p_r)$ is equal to number of quasi-relevant squares whose corner in U has x -coordinate in $[p_l, p_r)$.*

Furthermore, if the number of relevant squares with corners left of p_r is strictly smaller than κ , then these numbers are also equal per trajectory.

Proof. We prove the lemma for the set $\Sigma_{\text{bl}}(\mathcal{T})$ of squares with their bottom-left corner in U . For the rest of this proof the term “corner” always refers to the bottom-left corner. Note that since the sweep line only halts at safe stops in $\mathcal{L}_y$, the information about which squares intersect ℓ is correct.

Recall that a square $\sigma_p \in \Sigma_{\text{bl}}(T_i)$ is *active* if it is the leftmost square of $\Sigma_{\text{bl}}(T_i)$ intersecting the sweep line. We call σ_p *quasi-active* if it is the leftmost square of $\Sigma_{\text{bl}}(T_i)$ intersecting the sweep line according to the (partially incorrect) ordering in $\mathcal{L}_x$. Now number the intervals between consecutive safe stops along ℓ as $I_1, I_2, \dots$. We will prove that the lemma holds for all these intervals I_j by induction on j . For simplicity we assume we have an empty interval I_0 for which the lemma trivially holds. Now consider an interval $I_j := [p_l, p_r)$ and assume the lemma holds for all intervals $I_{j'}$ with $j' < j$. We have three cases.

Case 1: there are strictly fewer than κ relevant squares whose corner lies to the left of p_r . By induction the number of relevant squares to the left of p_l is equal to the number of quasi-relevant squares to the left of p_l , and this holds in fact per trajectory. That is, a trajectory T_i has a relevant square to the left of p_l if and only if it has a quasi-relevant square to the left of p_l . Now consider a trajectory T_i with an active square in I_j . By definition it cannot have a relevant square to the left of p_l and, as we saw, this implies it does not have a quasi-relevant square to the left of p_l , which implies that T_i must have a quasi-active square in I_j . The converse is also true, so the number of active and quasi-active squares inside I_j must be equal per trajectory. Since the number of relevant squares to the left of p_r is strictly less than κ , these active and quasi-active squares must actually be relevant and quasi-relevant, respectively.

Case 2: there are κ relevant squares whose corner lies to the left of p_r , but strictly less than κ relevant squares to the left of p_l . Using the same argument as in Case 1 we derive that, if trajectory T_i has an active square in I_j , then T_i has a quasi-active square in I_j . Let z be the number of relevant squares to the left of I_j . Then the $\kappa - z$ leftmost active squares in I_j are relevant. Similarly, the $\kappa - z$ quasi-active squares in I_j that are leftmost according to the list $\mathcal{L}_x$ are quasi-relevant. These need not come from the same trajectories as the relevant squares, but their number is the same.

Case 3: there are κ relevant squares whose corner lies to the left of p_l . Now there are no relevant squares in I_j . By the induction hypothesis we know that there are also κ quasi-relevant squares to the left of p_l , so there cannot be quasi-relevant squares in I_j either. $\square$

Note that the lemma indeed proves the correctness of our approach: it implies that the popularity computed at safe stops is correct, and it proves that the popularity of an interval between two (not necessarily consecutive) safe stops, without quasi-relevant squares in between them, is indeed constant throughout the interval.

It remains to discuss the error. Since the popularity of intervals without quasi-relevant squares in them is determined correctly, we only miss popular subintervals of an interval between safe stops surrounding one or more quasi-relevant squares. But such safe stops are actually consecutive safe stops in $\mathcal{L}_x$ —no safe stops have been skipped in between—so after round r the length of such intervals is at most $1/2^r$. Since the distance between consecutive safe stops in $\mathcal{L}_y$ is $1/2^r$ as well, the error in round r is bounded by this quantity, leading to the following lemma.

Lemma 9 *The 2D popular-places problem within a unit square U can be solved with a progressive algorithm that runs in at most $k := \lfloor \log n \rfloor$ rounds, each taking $O(n\kappa)$ time, with convergence function $f_{\text{conv}}(r) = 1/2^r$ with respect to the error function $\text{err}_{\text{pop}}(S)$ defined above.*

Computing 2D popular regions—the general case. We showed how to progressively compute the popular region within a unit square U . Computing the entire popular region can be done using the algorithm above as a subroutine. To this end, consider the integer grid and let $\mathcal{U} = \{U_1, \dots, U_z\}$ be the grid cells intersecting at least one square from $\Sigma(\mathcal{T})$. The set $\mathcal{U}$, and for each $U_i \in \mathcal{U}$ the squares intersecting U_i , can be computed in $O(n)$ time using geometric hashing. (This requires the floor function, which can be avoided using an approach based on a progressive computation of a kd-tree-like structure—see Section 4.1.) We now run the algorithm described above on each of the cells U_i in parallel. Each round takes $O(\sum_i n_i \kappa) = O(n\kappa)$ time, where n_i denotes the number of squares intersecting U_i , giving the following result.

Theorem 10 *Let $\mathcal{T} := \{T_1, \dots, T_m\}$ be a set of 2D trajectories with n points in total, let $\mathcal{B}$ be the extended bounding square of $\mathcal{T}$, and let κ be a parameter. Then there exists a progressive algorithm to compute the κ -popular region that runs in at most $k := \lfloor \log n \rfloor$ rounds, each taking $O(n\kappa)$ time, with convergence function $f_{\text{conv}}(r) = 1/(\text{size}(\mathcal{B}) \cdot 2^r)$ with respect to the error function $\text{err}_{\text{pop}}(S)$ defined above. The algorithm is monotone in the sense that, if S_r denotes the solution reported after round r , we have $S_r \subseteq S_{r+1}$ for all $1 \leq r < k$. The algorithm requires the use of the floor function.*

4.1 An approach based on kd-trees

Above we described a progressive algorithm for a special case of the popular-places problem, where we want to find the solution within a unit square U . We then gave a simple grid-based

approach that reduces the general problem to this special case. The grid-based approach needs the floor function, however. Here we describe an alternative that does not need this.

Recall that $\mathcal{B}$ is a bounding square of the set $\mathcal{T}$ (which contains the points from all trajectories), extended to each side by half a unit. We recursively construct a partitioning of $\mathcal{B}$ into rectangles, similar to a kd-tree, as follows.

We perform the recursive partitioning in rounds, and in a breadth-first manner. Thus, in each round we are given a set of rectangles R —initially this set only contains $\mathcal{B}$ —which we partition as explained next. Let P be the set of corners of the squares in $\Sigma(\mathcal{T})$. For each rectangle R we have a list $\mathcal{L}_P(R)$ that contains the points in $R \cap P$, and a list $\mathcal{L}_\Sigma(R)$ of all squares $\sigma_p \in \Sigma(\mathcal{T})$ that intersect R . We now split R using the following splitting rules.

- When the maximum side length of R is at most 1, or R does not intersect any edge of a square in $\mathcal{L}_\Sigma(R)$, then R is a final region of the partitioning and the recursion ends.
- When the maximum side length of R is greater than 1, and $\mathcal{L}_P(R) = \emptyset$ but R does intersect an edge of a square in $\mathcal{L}_\Sigma(R)$, then we have some square edges fully crossing R . Note that we can only have crossing edges parallel to the shorter side of R . We now split R into three subrectangles, using vertical lines through the median of the crossing edges and through the midpoint of the longer side of R .
- Otherwise, the maximum side length of R is greater than 1 and $\mathcal{L}_P(R) \neq \emptyset$. We partition R into two or four equal-sized subrectangles by lines orthogonal to the sides of R . More precisely, if the shorter side has length at most 1 we only split orthogonal to the longer side, otherwise we split orthogonal to both the shorter and the longer side. We then split the subrectangle with the largest number of points, call it R' , into two rectangles by a line orthogonal to the longer side of R' and through the median of $\mathcal{L}_P(R')$, assuming this side has length more than 1. (If the longest edge of R' has length at most 1, then R' is a final rectangle.)

Note that the lists for the at most five subrectangles resulting from splitting R can easily be determined from the lists for R in linear time. At the end of the round, we check for each vertex of every rectangle R in the current partition whether it is popular—this can be done by inspecting all rectangles in $\mathcal{L}_\Sigma(R)$ —and we report the set of all popular vertices as a partial solution.

Lemma 11 *Each round in the process described above takes $O(n)$ time, and the total number of rounds is at most $\min(\lceil \log n \rceil, \lceil \log(\text{size}(\mathcal{B})) \rceil)$.*

Proof. Let $\mathcal{R}_r$ denote the set of rectangles at the beginning of round r . Consider a square $\sigma_p \in \Sigma(\mathcal{T})$. Because of our splitting rules—in particular because we always split orthogonal to a side of length greater 1—the square σ_p intersects $O(1)$ rectangles from $\mathcal{R}_r$. This implies that the total size of all lists $\mathcal{L}_P(R)$ and $\mathcal{L}_\Sigma(R)$ is $O(n)$, so a round takes $O(n)$ time.

The $\lceil \log n \rceil$ bound on the number of rounds follows from the fact that any non-final rectangle is split through a median point or edge, and that the recursion ends when no edge crosses a cell. The $\lceil \log(\text{size}(\mathcal{B})) \rceil$ bound follows from the fact that any non-final rectangle is

split through the midpoints of its edges that have length greater than 1, and the fact that the recursion stops when a rectangle drops below unit size. $\square$

We now analyze the error of the reported partial solutions.

Lemma 12 *The partial solution reported after round r has error at most $1/2^r$.*

Proof. Any rectangle R that is already a final rectangle of the partitioning either has maximum edge length at most 1, or it does not intersect any square from $\Sigma(\mathcal{T})$. In the former case the maximum size of a square centered inside $A_\kappa(\mathcal{T}) \cap R$ and not containing any reported vertex is at most 1, so the error is at most $1/\text{size}(\mathcal{B}) \leq 1/2^r$ because $r \leq \log(\text{size}(\mathcal{B}))$. In the latter case $A_\kappa(\mathcal{T}) \cap R = \emptyset$. Any non-final rectangle must have side length at most $\text{size}(\mathcal{B})/2^r$, so the error is at most $1/2^r$ as well. $\square$

After the recursive partitioning has ended, we apply the special-case algorithm to all rectangles R with $\mathcal{L}_\Sigma(R) \neq \emptyset$. This is possible due to the following lemma, which immediately follows from our stopping rule.

Lemma 13 *After the final round, any rectangle R with $\mathcal{L}_\Sigma(R) \neq \emptyset$ has at most unit size in both dimensions.*

This leads to the following theorem. Note that the convergence function is slightly different from the one in Theorem 10, because the grid-approach allowed us to reduce the error to $1/\text{size}(\mathcal{B})$ in one step. Also note that a more precise bound on the number of rounds is $\lceil \log n \rceil + \min\{\lceil \log n \rceil, \lceil \log(\text{size}(\mathcal{B})) \rceil\}$ —see Lemmas 9 and 11. In the theorem below this is simplified to $2\lceil \log n \rceil$.

Theorem 14 *Let $\mathcal{T} := \{T_1, \dots, T_m\}$ be a set of 2D trajectories with n points in total, let $\mathcal{B}$ be the extended bounding square of $\mathcal{T}$, and let κ be a parameter. Then there exists a progressive algorithm to compute the κ -popular region that runs in at most $k := 2\lceil \log n \rceil$ rounds, each taking $O(n)$ time, with convergence function $f_{\text{conv}}(r) = 1/2^r$ with respect to the error function $\text{err}_{\text{pop}}(S)$ defined above. The algorithm is monotone in the sense that, if S_r denotes the solution reported after round r , we have $S_r \subseteq S_{r+1}$ for all $1 \leq r < k$.*

Remark. The same result can be obtained by developing an approximation algorithm, and then using Theorem 2. The idea would be to solve the problem within a unit square U by placing a grid of edge length ε inside the square, and computing for each grid point whether it is popular or not. The resulting algorithm would probably be slightly simpler (at least conceptually). We have chosen to present the algorithm above because it nicely illustrates how the plane-sweep paradigm can be used in a progressive setting.

5 Concluding remarks

We presented a theoretical framework for analyzing so-called progressive algorithms: algorithms working in rounds and producing partial solutions with smaller errors in each

round r , until the complete solution is reported. We investigated the framework by developing and analyzing progressive algorithms for sorting, convex hulls, and popular places. This showed some interesting connections. For instance, our algorithm for popular places is a plane-sweep algorithm based on our progressive sorting algorithm.

In our framework the convergence rate can be artificially boosted by merging multiple rounds into single rounds. Hence, it can distinguish between algorithms with different polynomial convergence rates (for example, $O(1/r)$ versus $O(1/r^2)$) and between polynomial convergence and exponential convergence, but it cannot distinguish between different exponential convergence (for example, $O(1/2^r)$ and $O(1/2^{2r})$ cannot be distinguished).

Our framework is currently limited to static data sets. In some applications, however, it is natural to consider variants where the data is not fully available at the start of the computation, but more and more data becomes available as time progresses. It may be interesting to study how to apply our framework in such a streaming scenario. As for the specific problems we studied, an interesting open problem is to develop a progressive variant of the output-sensitive algorithm for computing popular places of Benkert *et al.* [3].

Acknowledgement. We thank the referees for their careful reading and for spotting some “technical typos” in the first version of this paper.

References

- [1] P.K. Agarwal, S. Har-Peled, and K.R. Varadarajan. Geometric approximation via coresets. In *Combinatorial and Computational Geometry*, volume 52, pages 1–30. Cambridge University Press, 2005.
- [2] C.B. Barber, D.P. Dobkin, and H. Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Trans. Mathematical Software (TOMS)*, 22(4):469–483, 1996.
- [3] M. Benkert, B. Djordjevic, J. Gudmundsson, and T. Wolle. Finding popular places. *Internat. J. Comput. Geom. Appl.*, 20(1):19–42, 2010.
- [4] T.M. Chan. Faster core-set constructions and data-stream algorithms in fixed dimensions. *Comput. Geom. Theory Appl.*, 35(1):20–35, 2006.
- [5] P. Diaconis and R.L. Graham. Spearman’s footrule as a measure of disarray. *Journal of the Royal Statistical Society*, 39:262–268, 1977.
- [6] J. Giesen, E. Schuberth, and M. Stojaković. Approximate sorting. *Fundamenta Informaticae*, 90(1):67–72, 2009.
- [7] J. Hersberger and S. Suri. Simplified planar coresets for data streams. In *Proc. 11th Scandinavian Workshop Algorithm Theory (SWAT)*, LNCS 5124, pages 5–16, 2008.
- [8] P. Laube, S. Imfeld, and R. Weibel. Discovering relative motion patterns in groups of moving point objects. *Internat. J. Geo. Inf. Sci.*, 19(6):639–668, 2005.

- [9] P. Laube, M. van Kreveld, and S. Imfeld. Finding REMO – detecting relative motion patterns in geospatial lifelines. In *Developments in Spatial Data Handling: Proc. 11th Int. Sympos. Spatial Data Handling*, pages 201–215, 2004.
- [10] N. Megiddo. Linear programming in linear time when the dimension is fixed. *J. ACM* 31(1):114–127, 1984.
- [11] S.J. Russell and S. Zilberstein. Composing real-time systems. In *Proc. 12th Int. Joint Conf. on Artificial Intelligence*, pages 212–217, 1991.
- [12] S. Zilberstein. Using anytime algorithms in intelligent systems. *AI Magazine*, 17:73–83, 1996.