Practical Low-Dimensional Halfspace Range Space Sampling

Michael Matheny
University of Utah, USA

Jeff M. Phillips[1]
University of Utah, USA

───── **Abstract** ─────

We develop, analyze, implement, and compare new algorithms for creating ε-samples of range spaces defined by halfspaces which have size sub-quadratic in $1/\varepsilon$, and have runtime linear in the input size and near-quadratic in $1/\varepsilon$. The key to our solution is an efficient construction of partition trees. Despite not requiring any techniques developed after the early 1990s, apparently such a result was never explicitly described. We demonstrate that our implementations, including new implementations of several variants of partition trees, do indeed run in time linear in the input, appear to run linear in output size, and observe smaller error for the same size sample compared to the ubiquitous random sample (which requires size quadratic in $1/\varepsilon$). This result has direct applications in speeding up discrepancy evaluation, approximate range counting, and spatial anomaly detection.

2012 ACM Subject Classification Theory of computation → Computational geometry

Keywords and phrases Partitions, Range Spaces, Sampling, Halfspaces

Digital Object Identifier 10.4230/LIPIcs.ESA.2018.62

1 Introduction

Taming the relationship between a point set $X \subset \mathbb{R}^d$ and its interaction with halfspaces $\mathcal{H}_d$, has long been a focus of computational geometry. Understanding and controlling this interaction is at the heart of problems in range searching, linear classification, coresets, and spatial anomaly detection. This pair $(X, \mathcal{H}_d)$ describes a range space, the combinatorial set of all subsets of X defined by $h \cap X$ for any halfspace $h \in \mathcal{H}_d$. In this paper we focus on two specific and closely-interrelated (as it turns out) constructions for $(X, \mathcal{H}_d)$: ε-samples and partitions, defined next.

An ε-*sample* $Y \subset X$ of $(X, \mathcal{H}_d)$ is a small point set that approximately preserves density with respect to halfspaces: for all $h \in \mathcal{H}_d$, and error parameter $\varepsilon \in (0, 1)$ it bounds

$$\mathsf{Error}(X, Y) = \max_{h \in \mathcal{H}_d} \left| \frac{|Y \cap h|}{|Y|} - \frac{|X \cap h|}{|X|} \right| \le \varepsilon.$$

It is known that ε-samples of size $\Theta(1/\varepsilon^{2d/(d+1)})$ exist for halfspaces [3], and in general this size may be required [21]. For many years (c.f., [22, 8]) such proofs were not constructive, as they relied on the "partial coloring lemma"; until in 2010 when Bansal [5] introduced a polynomial time construction. The runtime of the low-discrepancy coloring on m points was later reduced [16] to $O(m^{3(d+1)}\mathrm{polylog}(m))$, this within the standard merge-reduce framework [10] results in a $O(n(1/\varepsilon)^{2d(3d+2)/(d+1)}\mathrm{polylog}(1/\varepsilon))$ runtime for sample construction– which is

still not very efficient. For instance for $d = 2$, this requires $O(n(1/\varepsilon)^{10+2/3}\text{polylog}(1/\varepsilon))$ time. A random sample, which can be generated in $O(n + 1/\varepsilon^2)$ time, is an ε-sample of size $O(\frac{1}{\varepsilon^2}(d + \log\frac{1}{\delta}))$ with probability at least $1 - \delta$ [27, 15]. The above discrepancy-based algorithm can be run on the output of this sample to get optimal size, but it only reduces the overall runtime of the ε-sample construction to $O(n + (1/\varepsilon)^{2d(3d+2)/(d+1)+2}\text{polylog}(1/\varepsilon))$.

There are other constructions for ε-samples, which either focus on small space (to work in a stream) [26, 4] or have better performance in practice without size guarantees below that of random sampling [2]. As with the optimal algorithms, these require the enumeration of all combinatorial halfspaces associated with a set of size roughly the size of the final ε-sample, requiring at least $\Omega((1/\varepsilon^{2d/(d+1)})^d)$ time. Indeed Suri *et al.* [26] concludes with: "*The high computational complexity of the currently known algorithms for these subroutines may be prohibitive for data stream applications. It is a long standing open problem to find efficient exact or approximation algorithms for either of them.*"

A *partition* of $(X, \mathcal{H}_d)$ is a set of pairs $\{(\Delta_1, X_1), (\Delta_2, X_2), \ldots\}$ where each Δ_i is a small complexity region and contains $X_i \subset X$, and X is the disjoint union of the X_is. It is a (t, z)-*partition* when there are $O(t)$ pairs, $|X_i| \leq 2n/t$; and each $h \in \mathcal{H}_d$ crosses $O(t^z)$ cells. The smallest possible guarantee for z is $z = (1 - 1/d)$, and an algorithm for such a construction was provided by Matoušek [20], that takes $O(n \log t)$ time after $O(n^{1+\eta})$ preprocessing time for any $\eta > 0$. Chan provided a refined algorithm which takes $O(n \log t)$ time, and has a few nicer structural properties. There are other algorithms which generate (t, z)-partitions for large values of z. For instance in $\mathbb{R}^2$ Edelsbrunner and Welzl [11] describe an algorithm with $z = 0.695$ and a structure similar to a kd-tree leads to a size of $z = \log_4(3) \leq 0.7925$ [29].

Our results. In this paper, we use partition construction algorithms to efficiently create ε-samples for $(X, \mathcal{H}_d)$. Our algorithm takes $O(n + \frac{1}{\varepsilon^2}\log\frac{1}{\varepsilon})$ time and produces an ε-sample of size $O((1/\varepsilon)^{2d/(d+1)}\log^{d/(d+1)}(1/\varepsilon))$, nearly matching the $\Omega(1/\varepsilon^{2d/(d+1)})$ lower bound.

We also implement several variants of these algorithms in $\mathbb{R}^2$. We know of no other implementation of ε-sample construction for $(X, \mathcal{H}_2)$ which is guaranteed to get subquadratic size in $1/\varepsilon$. We know of no implementations of optimal partitions, although Har-Peled [13] has implemented a related concept called a cutting, which (as we will explain) is a key ingredient for creating partitions. We choose to build our own implementation of cuttings, and explain why we did not use Har-Peled's in Section 4.

We are able to demonstrate that our algorithm indeed scales linearly in n, scales linearly in the output size, and produces ε-samples with less measured error than random samples.

Our initial goal in fast ε-sample construction comes from finding approximately maximal ranges in range spaces, as part of a large-scale spatial anomaly detection framework [18, 17]. At a high level, these algorithms follow two phases: (1) create an ε-sample S, (2) use S to find an approximately maximal range. The second step takes $O(|S|/\varepsilon)$ or $O(|S|/\varepsilon^2)$, so it is only worth using a smaller ε-sample of size roughly $1/\varepsilon^{4/3}$ if it takes less than $1/\varepsilon^{2+1/3}$ or $1/\varepsilon^{3+1/3}$ time to create. We show this is the case in theory, and in practice. Similar overall runtime gains exist when using S for classification, or approximate range counting, or other tasks where the use of S is more expensive than the new construction time.

2 Overview and Proof for Fast ε-Samples

The key to our construction of an ε-sample S for a range space $(X, \mathcal{H}_d)$ is to first create a partition over $(X, \mathcal{H}_d)$. Given such a partition algorithm, our algorithm constructs an ε-sample as follows. Randomly sample $Y \subset X$, construct the partition $\Delta = \{(\Delta_1, Y_1), \ldots, \}$ on Y, and return a single point at random from each Y_i weighted by $|Y_i|$.

▶ **Theorem 1.** *For range space $(X, \mathcal{H}_d)$ with $|X| = n$ and constant d, with constant probability an ε-sample S of size $O(\frac{1}{\varepsilon^{2d/(d+1)}} \log^{d/(d+1)} \frac{1}{\varepsilon})$ can be constructed in $O(n + \frac{1}{\varepsilon^2} \log \frac{1}{\varepsilon})$ time.*

Proof. Take a random uniform sample $Y \subset X$ of size $s = O(\frac{1}{\varepsilon_1^2})$ then Y is an ε_1-sample of $(X, \mathcal{H}_d)$ with constant probability. Next we build a $(t, 1 - 1/d)$-partition on Y in $O(s \log t)$ time [7]; this results in a set of $O(t)$ partitions of Y each containing at most $2s/t$ points such that any halfspace in $\mathcal{H}_d$ will only cross $O(t^{1-1/d})$ of them. From each partition (Δ, Y_i) we will choose a single point y_i at random to put in our result S, and weight it proportional to the number of points in the partition.

In our construction any partition contained completely inside a halfspace or outside does not contribute to the error of the sample. Only regions crossing the boundary of the halfspace h contribute to the error. The error in each boundary region is an independent bounded random variable V_i with value in the range $[0, 2\frac{s}{t}]$. There are at most $k = c \cdot t^{1-1/d}$ boundary regions for some constant c, so we can apply Hoeffding's inequality, with failure probability δ

$$\Pr[|V - \mathsf{E}[V]| \geq s\varepsilon_2] \leq 2 \exp\left(-\frac{2s^2\varepsilon_2^2}{ct^{1-1/d} \cdot 4\frac{s^2}{t^2}}\right) = 2 \exp\left(-\frac{\varepsilon_2^2 t^{1+1/d}}{2c}\right) \leq \delta.$$

Rearranging the last inequality, gives that with $t \geq (\frac{2c}{\varepsilon_2^2} \ln \frac{2}{\delta})^{d/(d+1)}$, for any one halfspace h, $|V - \mathsf{E}[V]|$ is more than $s\varepsilon_2$ with probability at most δ.

There are $O(s^d) = O(1/\varepsilon_1^{2d})$ halfspaces in $(Y, \mathcal{H}_2)$, so setting $\delta = c_2\varepsilon_1^{2d}$ for some constant c_2, and the additivity property of ε-approximations [8], gives an $(\varepsilon_1 + \varepsilon_2)$-approximation of size $t \geq \left(\frac{4dc}{\varepsilon_2^2} \ln \frac{2}{c_2\varepsilon_1}\right)^{d/(d+1)}$ with constant probability. By setting $\varepsilon_1 = \varepsilon_2 = \frac{\varepsilon}{2}$ the total error is $\varepsilon_1 + \varepsilon_2 = \varepsilon$ and the size of the ε-sample is $O\left(\frac{1}{\varepsilon^{2d/(d+1)}} \log^{d/(d+1)}\left(\frac{1}{\varepsilon}\right)\right)$ for constant d. Creating Y takes $O(n + \frac{1}{\varepsilon^2})$ time, the partition tree construction takes $O(\frac{1}{\varepsilon^2} \log \frac{1}{\varepsilon})$ time since $t = O(\mathsf{poly}(\frac{1}{\varepsilon}))$, and the re-weighting and sampling step takes $O(\frac{1}{\varepsilon^2})$ time. In total therefore the entire algorithm takes $O(n + \frac{1}{\varepsilon^2} \log \frac{1}{\varepsilon})$ time. ◀

The same proof technique will work with other (t, z)-partitions in place of Chan's [7]. In general, for $z < 1$, a scheme that generates a (t, z)-partition of t cells where any halfspace crosses at most $O(t^z)$ of the cells results in an ε-sample of size $O(\frac{1}{\varepsilon^{2/(2-z)}} \log^{1/(2-z)} \frac{1}{\varepsilon})$. For instance in $\mathbb{R}^2$, Edelsbrunner and Welzl's $z = 0.695$ result [11] in an ε-sample of size $O(\frac{1}{\varepsilon^{1.532}} \log^{0.766}(\frac{1}{\varepsilon}))$. Alternatively, Willards $z = 0.7925$ result in $\mathbb{R}^2$ [29] results in an ε-sample size of $O(\frac{1}{\varepsilon^{1.657}} \log^{0.829}(\frac{1}{\varepsilon}))$.

3 Overview of Algorithms for Constructing the Partition

Random sampling, and sampling a point from each cell of a partition is straight-forward; the challenge in our implementation of Theorem 1 is the creation of a partition. In this section we describe the key components of the two prominent optimal size ($z = 1 - 1/d$) algorithms: Matoušek's efficient partitioning [20] (ComputePartition-Mat) and Chan's Optimal partitioning [7] (ComputePartition-Chan).

These algorithms rely on a related object called a cutting, defined over $\mathbb{R}^d$ and a set of m hyperplanes H. For a parameter $r < m$, a $(1/r)$-*cutting* is a decomposition of $\mathbb{R}^d$ into $O(r^d)$ cells $\Lambda = \{\Lambda_1, \Lambda_2, \ldots\}$, so no cell is crossed by more than $O(m/r)$ hyperplanes in H. Such cuttings exist and can be computed in $O(mr^{d-1})$ time [9, 19].

Cuttings are almost enough to compute partitions. A set of n points in $\mathbb{R}^d$ induces $m = O(n^d)$ combinatorially distinct halfspaces H. Letting $r = t^{1/d}$, the total number of crossings will be $O(r^d \cdot m/r) = O(mr^{d-1})$, so the *average* per region will be $O(r^{d-1}) = O(t^{1-1/d})$. Also,

ignoring dependences, the average cell contains $O(n/r^d) = O(n/t)$ points, as desired. The main challenge is ensuring that these average properties of the cutting map to the specific properties required for the partition. In short, we can create an appropriate cutting, detect where it does not satisfy the partition properties, and then amend it so it does.

We specifically focus our implementations in the $d = 2$ setting, which for instance is enough for our original application of spatial anomaly detection we mentioned previously [17], even in higher dimensions. Our implementations are similar to the existing implementation of cuttings by Har-Peled [13], but adds several features which will aid in computing the partition. Our cutting implementation builds a cutting by iteratively adding lines in a random order while keeping track of the number of lines crossing each cell in an arrangement. From a practical point of view, it is important to force the cells of the partition to be constant size. We have focused on two methods for this, a vertical trapezoidal decomposition (Trapezoid), or a hierarchy of constant size polygons (PolyTree).

Constructing a $(1/r)$-cutting over the entire set of $O(n^d)$ halfspaces would lead to a runtime of $O(n^d r^{d-1})$ which would be prohibitively slow. Instead of using the full set of halfspaces a smaller set (a test set) can be constructed, such that the number of partitions crossed by any halfspace in this test set will not be too different from the full set $\mathcal{H}_d$.

In particular, an $(1/r)$-*test set* is a set of halfspaces H which applies to any partition $\Delta = \{(\Delta_1, X_1), (\Delta_2, X_2), \ldots\}$ and point set X of size n so $|X_i| \geq n/r$ for all $(\Delta_i, X_i) \in \Delta$. It ensures that if $\kappa = \max_{h \in H} |h \cap \Delta|$, then $\max_{h \in \mathcal{H}_d} |h \cap \Delta| \leq O(\kappa + r^{1-1/d})$. Here $h \cap \Delta$ is the set of $(\Delta_i, X_i) \in \Delta$ for which Δ_i intersects h, but do not completely contain h. Test sets can be built a number of ways, including randomly sampling lines, randomly sampling points and using the lines they induce, and using the dual arrangement.

4   Implementation Particulars of Partitions

Our implementation of Partition trees is in python. It relies on an efficient way to construct and maintain an arrangement of lines and associated points. At each step of the construction we will maintain a tree with leaves that correspond to cells $\Delta_1, \Delta_2, \ldots$ of an arrangement. Each cell will maintain a list of contained points $X_i \in \Delta_i$ and crossing lines.

As part of the construction so the result is a $(t, 1 - 1/d)$-partition Δ, with desired t parameter, cells can be refined by applying various operations to them. For instance a cutting can be constructed locally inside of a cell Δ_i, or a cell can be partitioned into a set of sub-cells.

Geometric Primitives.   All of our algorithms rely on operations over line segments. The most important operation is being able to test, within a region Δ, if a line lies completely above a line segment or if it crosses a line segment. This fairly simple operation is slightly complicated by numerical issues that can occur. For instance when constructing a test set using the BuildTestSet-Points or BuildTestSet-Dual method (see below) many lines will potentially meet at the same point. Line segments that meet in this point could be mistaken as crossing. To handle numerical issues we use python's implementation of `math.isclose` to handle point comparisons. This method allows us to assign two floating point numbers as equal if their relative values are sufficiently close [6]. Moreover, all methods that compare line segments have closed and open versions where closed versions allow end-point overlap and open versions do not. The method `segment.above_closed(line)` returns true if the `line` intersects with the segment at one the segment's end points, but is otherwise above the segment, while `segment.above_open(line)` returns false in this case. This allows us in our experiments to effectively handle degeneracies while avoiding slower exact precision libraries.

Internally our segment objects are represented by the slope, a, the y-intercept, b, and the $[x_l, x_r]$ interval on the x-axis the segment is defined over. This representation makes many operations easy, but also results in several challenges, most notably: vertical lines are undefined, unbounded segments (e.g., $(-\infty, x_r]$) require extra logic to handle crossing queries, lines which are nearly vertical can become numerically unstable, and the dual of unbounded polygons require significant extra logic to handle correctly. However, we have implemented stable functions for intersect and above relations for pairs of segments in a cell.

Using line segments and points as the primitives we also define more complicated structures notably: polygons, dual wedges, vertical line segments, and trapezoids.

PolyTrees. There are a number of ways to maintain the structure of an arrangement. A common method is to store each cell with a corresponding list of pointers to adjacent cells. Inserting a line involves finding the leftmost crossing cell, identifying the next adjacent cell the line crosses, splitting the crossed cell into an upper and lower cell, and then repeating this operation for each crossed cell.

This has a number of downsides: there are special cases if a line crosses a vertex of a cell, inserting points into the arrangement requires the maintenance of a secondary structure, and cells require a significant amount of adjacency information that must be maintained. Instead of maintaining this structure we use the idea of forcing each cell to be simple, and follow certain restrictions, as introduced by Seidel [25] and refined by Har-Peled [13].

In particular, we either maintain a decomposition into constant complexity polygons (polygons with a constant number of boundary segments) or a trapezoidal decomposition. In both cases we maintain a tree where each node in the tree consists of a line segment that separates a cell into two cells. With trapezoids an inserted line could in some cases divide a trapezoid vertically into 2 separate trapezoids and then horizontally into 4 separate trapezoids. In the case of polygons the inserted line would split the polygon into two separate polygons which could possibly be further split if the number of sides in either of the resulting polygons is greater than a chosen constant. We also enforce that no vertical segments are used to avoid limitations of our line segment representation.

Given a line h and a decomposition $\Lambda = \{\Lambda_1, \Lambda_2, \ldots\}$, the *zone* of h is the set of regions Λ_i that intersect h; we represent this as $\mathsf{Zone}_h = \Lambda \cap h$. To find the zone of a line in this structure at each node we treat the line as an infinite length segment and then traverse the line down the tree. At each node we will have three cases where the portion of the line contained in the node lies completely either above or below, or crosses the current node's line segment. In the completely above or below case we merely traverse to the above or below child of the node. In the crossing case we split the portion of the line contained in the node into two segments, above and below, and recursively query the above and below nodes. Point information is easy to maintain with this method since a point always lies on one side of the line segment, so the tree structure can be used to insert or remove points in logarithmic time to the number of cells.

More complicated structures can also be queried on these trees, most notably wedges and polygons. Wedge queries are particularly useful in ComputePartition-Chan since a wedge is the dual of a line segment, so the number of points contained in a wedge corresponds in the dual to the number of lines crossing a line segment.

Cuttings. Our cutting algorithm CreateCutting(H, r) (Algorithm 1) follows closely Algorithm 1, from Har-Peled [13]. We implement the cutting with respect to weighted lines as this speeds up and somewhat simplifies the later partitioning algorithms. We require a

---

Algorithm 1 CreateCutting(H, r).

1: $\Lambda = \mathbb{R}^2$
2: **for** $h \in H$ (ordered by a random weighted permutation) **do**
3:     Find $\mathsf{Viol}(h, \Lambda) = \{\Lambda_i \in \mathsf{Zone}_h(\Lambda) \mid |H \cap \Lambda_i| > |H|/r\}$.
4:     For all $\Lambda_i \in \mathsf{Viol}(h, \Lambda)$, replace Λ_i in Λ by $\mathsf{split}(\Lambda_i, h)$
5: **return** Λ

Algorithm 2 BuildTestSet-Dual (X, r).

1: $S = \mathrm{sample}(X, O(\sqrt{r} \log r))$; S^* is dual of S.
2: $\Lambda \leftarrow \mathsf{CreateCutting}(S^*, O(r^{1/2}))$
3: **return** the dual of V^*, where V is the set of vertices of the cells of Λ.

---

weighted permutation of lines using [12]; this ensures that the probability we see a line after some point in the permutation is equivalent to the probability we would have seen at least one instance after seeing that many distinct lines in a variant where weights are multiplicities (as advocated by Chan [7]), and each copy is treated independently in a uniform random permutation. For notational convenience, for a subset $H' \subseteq H$, let $|H'| = \sum_{h \in H'} w(h)$, where $w(h)$ is the weight implicitly stored with each halfspace $h \in H$.

In practice, we implement $\mathsf{CreateCutting}(H, r)$ slightly differently then described in Algorithm 1. Instead of choosing the $h \in H$ to process in the random weighted permutation, our approach is centered around the violated cells. We choose a cell $\Lambda_i \in \Lambda$ which is too heavy (i.e., $|H \cap \Lambda_i| > |H|/r$), and then choose some halfspace $h \in \Lambda_i \cap H$, and only replace Λ_i (not the entire Zone_h) with the result of the $\mathsf{split}(\Lambda_i, h)$, which divides Λ_i into two parts separated by h. This change has two advantages. First we do not need to find the $\mathsf{Zone}_h(\Lambda)$ which involves traversing the $\mathsf{PolyTree}$, so our approach is slightly faster. Second, we can choose the $h \in \Lambda_i \cap H$ to use in the split wisely; e.g., as the one that maximizes the smaller of the two resulting cells. We find the second heuristic produces slightly smaller cuttings in practice, but is significantly slower, and is not used in our experiments.

How $\mathsf{split}(\Lambda_i, h)$ is implemented is the difference between the $\mathsf{Trapezoid}$-based cutting and the $\mathsf{Polygon}$-based cutting we refer to as $\mathsf{PolyTree}$.

For the $\mathsf{Trapezoid}$-based method each cell Δ is a trapezoid to begin with. The split operation first inserts up to two new vertical cuts for each intersection of the line with the top or bottom of the cell and then horizontally cuts the resulting cells using the inserted line. For $\mathsf{PolyTree}$, the first important detail is that we store $H \cap \Lambda_i$ as the line segments restricted to where they intersect Λ_i.

On a split, we need to maintain which halfspaces $h \in \Lambda_i \cap H$ are in each child; if $h' \in \Lambda_i \cap H$ intersects both children, then we split h' into two line segments at the point where it intersects h, and store the corresponding segment in each child. If $h' \in \Lambda_i \cap H$ is in only one child, because we store them as segments it is easy to check which child it goes into.

Both of these algorithms are then fairly straightforward to implement once given structures for efficiently maintaining arrangements of line segments.

We find that $\mathsf{PolyTree}$ is faster and produces a smaller cutting than $\mathsf{Trapezoid}$-based ones; see Figure 1 and Figure 2 which show the runtime and cutting size as a function of input size and choice of r. For this reason we will primarily focus on $\mathsf{PolyTree}$ hereafter.

Test Set Generation.   There are a number of ways to generate test sets ($\mathsf{BuildTestSet-Dual}$, $\mathsf{BuildTestSet-Points}$, $\mathsf{BuildTestSet-Lines}$), but these do not appear to have a significant effect on the runtime of the final algorithms; again see Figure 1 and Figure 2 for comparisons
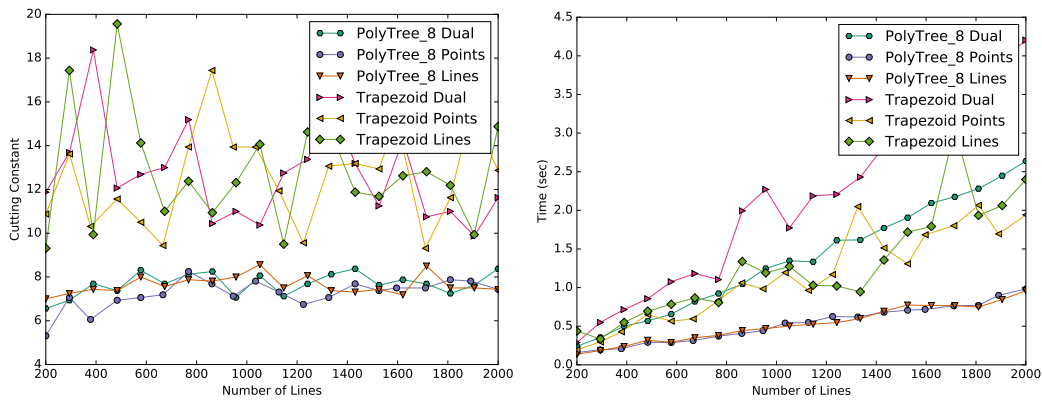
Figure 1 Size of cutting (divided by r^2) and time (in seconds) vs. the number of input lines.
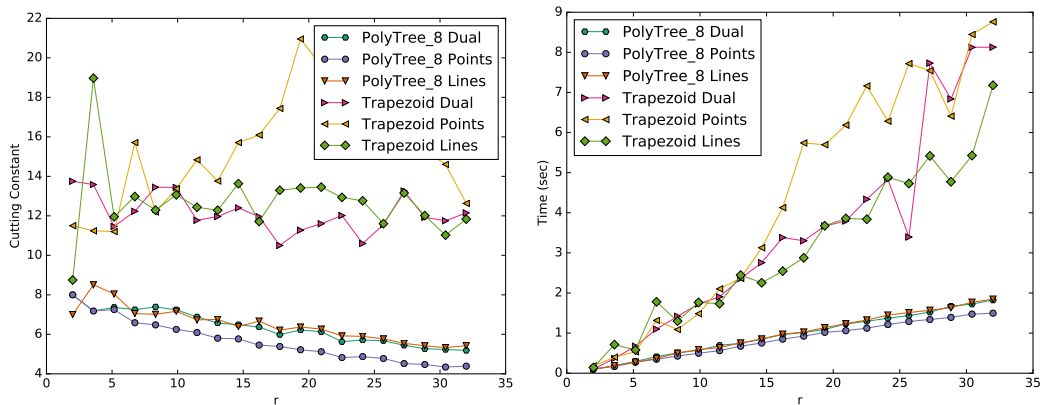
Figure 2 Size of cutting (divided by r^2) and time (in seconds) as r increases for 1000 input lines.

of the PolyTree and Trapezoid methods. The simplest method, BuildTestSet-Lines, simply samples $O(r \log^d n)$ halfplanes, from those defined by passing through d points in X. The next simplest, BuildTestSet-Points, samples $O(r^{1/d} \log n)$ points S, and then the test set is all halfplanes passing through d-tuples chosen from S; it again defines $O(r \log^d n)$ halfplanes. Finally the most complicated approach is BuildTestSet-Dual (see Algorithm 2); it produces the smallest size test set, size $O(r)$ [20], and thus is the one we advocate. It samples $O(r^{1/d} \log r)$ points $S \subset X$; it considers the dual set of halfplanes S^* of primal points S; it creates a $(1/r^{1/d})$-cutting of S^* (in the dual); and then it returns the primal halfspaces defined by the vertices of the cutting in the dual. Each halfspace h in the test set H is implicitly endowed with a weight $w(h)$, which by default is $w(h) = 1$ for all $h \in H$.

Matoušek Partitioning. We have implemented Matoušek's efficient partition trees [20]. At a high level this algorithm computes the cutting of a test set and then finds a single good cell that contain at least n/b points (for a constant b, we use $b = 16$ as default). It adds this cell to the partition, doubles the weight of all halfspaces in the test set crossing that cell, computes a new cutting and good cell. It repeats until the number of points remaining has been cut by half, and then it recurses on the remained of the points at half the precision (e.g., set $b := 1/2b$). This is too expensive to do with $b = r$, so after this we then recursively partition each cell (Δ_i, X_i) until the result is an $(1/r, 1 - 1/d)$-partition

---

Algorithm 3 ComputePartition-Mat(X, r, n, j).

1: **if** $(|X| < n/r)$ **then return** $\{(X, \Delta_0)\}$ where Δ_0 contains X.
2: $H \leftarrow$ BuildTestSet-x$(X, b/2^j)$
3: $\Delta = \emptyset$
4: **while** $(|X| \geq n/2^j)$ **do**
5:      $\Lambda \leftarrow$ CreateCutting$(H, \sqrt{b/2^j})$
6:      Find $\Lambda_i \in \Lambda$ so $|X \cap \Lambda_i| > n/b$; shrink Λ_i so $|X \cap \Lambda_i| = \lfloor n/b \rfloor$ exactly.
7:      Add $(\Lambda_i, \Lambda_i \cap X)$ to Δ; remove $\Lambda_i \cap X$ from X
8:      Double the weight $h \in H$ which cross Λ_i
9: $\Delta' = \bigcup_{(\Delta_j, X_j) \in \Delta}$ ComputePartition-Mat(X_j, r, n, j)
10: **return** $\Delta' \cup$ ComputePartition-Mat$(X, r, n, j + 1)$

Algorithm 4 ComputePartition-Chan(Δ, r, n).

1: Trim to $\Delta' = \{(\Delta_i, X_i) \in \Delta \mid |X_i| > n/r\}$; **if** $\Delta' = \emptyset$ **return** Δ
2: $H =$ BuildTestSet-x$(X, |\Delta|)$
3: **for** $(\Delta_i, X_i) \in \Delta'$ **do**
4:      Sample $L \subset H$, proportional to their weight $w(h)$, at rate q
5:      $\Lambda =$ CreateCutting(L, r_i); with r_i chosen so $|\Lambda| \leq b/4$
6:      For all $\Lambda_j \in \Lambda$, further split Λ_j (with split) until $|X_i \cap \Lambda_j| \leq |X_i|/b$
7:      Replace (Δ_i, X_i) in Δ with $\{(\Lambda_1, \Lambda_1 \cap X_i), (\Lambda_2, \Lambda_2 \cap X_i), \ldots\}$
8:      Update all weights $w(h) = w(h)(1 + 1/b)^{|h \cap \bar{\Lambda}_i|/p}$.
9: **return** ComputePartition-Chan(Δ, r, n)

---

as desired. The branching factor of the partition tree is not fixed on each level, but will be roughly b on average. Algorithm 3 presents this approach, and is initially called as ComputePartition-Mat$(X, r, |X|, 0)$.

Note that Line 9 is the refinement step where each cell is further partitioned. Since the first level is the most important for good ε-samples, faster algorithms could be used at later recurve calls at this step. In contrast, the recursive call at Line 10 is handing objects not handled in the first pass, where each pass handles roughly half of the data.

Chan Partitioning. Chan's optimal partition trees [7] are faster in theory than Matoušek's algorithm, but are more complicated to implement. The algorithm works by processing each node at a certain level in the tree in a random order. For each node it creates a cutting of approximately $b/4$ size for an appropriately large branching parameter b (our implementation uses $b = 22$ as a default). It then further splits the cells of the cutting to contain $1/b$ fraction of points at that node. It multiplicatively updates weights for halfplanes that cross each cell. This multiplicative update influences subsequent cuttings by biasing away from creating cells that are crossed by already heavily weighted lines (lines that cross many cells). After splitting all of the cells in this level of the tree the algorithm recurses on the newly created level. The ultimate partition $\Delta = \{(\Delta_1, X_1), (\Delta_2, X_2), \ldots\}$ are the leaf nodes of the tree. Algorithm 4 presents this approach, calling ComputePartition-Chan$(\{(\mathbb{R}^2, X)\}, r, |X|)$ initially.

Implementing the algorithm as described is too slow asymptotically, so Chan presents a faster variant, which requires two additional parameters p and q. Roughly $q = \sqrt{b|\Delta|}/|X|$ (see [7] for details) determines the probability that a line ends up in the reduced test set L. The parameter p, about $\sqrt{b/|\Delta|} \log n$ (again, see [7] for details), effects the number of cells Λ_i that are used to update the weight in each h (we sample each cells with probability p as

opposed to dividing by this number, as written on Line 8). Also, Line 4, where L is sampled from H, can be made more efficient by only minimally updating L each pass through the loop, since it generally has large weight lines and that set is fairly stable.

Near the bottom of the tree, Line 8 can be expensive. We make this efficient with a crucial observation that the test set H was generated by computing a cutting over the dual space. Thus these halfspaces are duals to the vertices of the PolyTree structure. Thus we can search over the PolyTree to determine the number of crossing lines. A cell of the partitioning is a polygon consisting of a constant number of line segments. A line crossing the polygon will cross at least one of the line segments and in the dual this will correspond to a point contained inside of a double wedge. For each line segment in the polygon we take its dual (a double wedge) and query the PolyTree that was used to construct the test set to determine the number of vertices contained inside of it. Since we only return the overlapped polygons and each polygon consists of at most a constant number of edges, the number of queried cells can only be a constant factor larger than the number of lines crossed by the line segment.

However, code profiling shows that the two steps involving sampling with p and q, and updating L are the most expensive parts of the algorithm. As a result we also consider a variant ComputePartition-Chan-Simple which avoids these sampling steps that were supposed to speed things up. In the context of Algorithm 4 this basically sets $p = q = 1$, so $L = H$, and Line 4 is not required.

The given algorithm is only guaranteed to compute a set of partitions in $O(n \log^{O(1)} n)$ time; incurring extra log factors due to the height of the partition tree. Chan removes log factors with a method he calls bootstrapping. We do not do this since the branching factor is high (around 22) so the depth of the tree is low, and this method is not worth the overhead.

In our implementation, we only compute the test set H once at the beginning. On each recursive call (Line 9) we can reuse it, but simply reset all of the weights to be uniform.

Ham-Sandwich Tree. We also implement an alternative using Willard's [29] Ham-Sandwich Tree. It provides a partitioning with $z = \log_3 4$, which gives a $O(\frac{1}{\varepsilon^{1.657}} \log^{0.829} \frac{1}{\varepsilon})$ sized sample, and constructs a tree with a branching factor of 4. At each level we split the point set in half with a single vertical line, and on these two resulting sets we find a single (roughly horizontal) line that divides both the left and right point set in half. Such a separator is guaranteed by the ham-sandwich theorem, and can be computed in linear time [24], but is complicated to implement. We instead approximate the ham-sandwich cut by computing a number of test lines and choosing the best separator from these. This is simple to implement, gives good cuts in practice, and can guarantee to be at most ε-imbalanced [23].

Why not use Har-Peled's implementation and CGAL? It may seem at first that we could simply use Har-Peled's implementation for ε-cuttings [13]. However, our initial goal was to use this as part of a code for spatial anomaly detection [18, 17], and there were several issues that made this less feasible. (1) We wanted to use non infinite precision floating point arithmetic. Har-Peled reports that switching to exact precision representations results in a 30-factor slow down, but was necessary for degeneracy issues. We managed these precision issues while using floating point arithmetic with careful use of open and closed operators for line above/below and intersection. (2) We can measure wedges, and line segments on the PolyTree structure which is very useful in ComputePartition-Chan. (3) Har-Peled's code created a cutting inside of a 1×1 box. This makes computing dual cuttings difficult as we first have to normalize the lines to lie in such a region, but computing the correct normalization quickly would require us to re-implement much of the PolyTree algorithm. Ultimately we opted to build our ε-cutting code from scratch rather than modify the previous code.
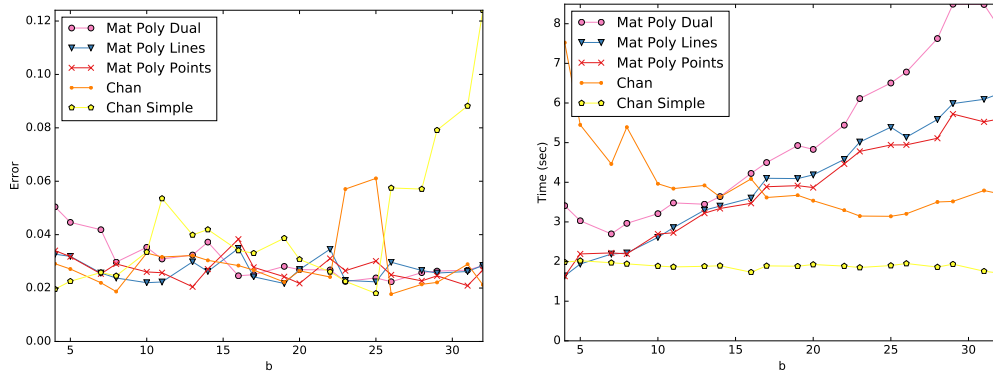
Figure 3 The Branching Factor b vs. Time and Error using the default parameters.

Har-Peled [13] also reported the cutting constant (number of cells divided by r^2) for various of his algorithms, about 7.3 (polygons) and 12.8 (trapezoids). This roughly matches the numbers we observe in Figure 1 and Figure 2.

Another option for computing cuttings and managing the partition trees is using the current 2d-arrangement implementation in CGAL [28]. This would have most likely made portions of this project much easier to implement and removed various hurdles. However, the possibilities of several factor slow-downs using exact precision would have potentially resulted in no ultimate gains in the spatial anomaly application demonstrated below.

## 5   Experiments on ε-Samples and Applications

In this section we explore the efficacy of our ε-sample algorithms based on partitions. We use as X the Chicago crime data [1] with roughly 6.5 million data points.

A key step of the analysis is measuring the accuracy of the ε-sample. That is for a sampled S we measure $\mathsf{Error}(X, Y) = \max_{h \in \mathcal{H}_d} |\frac{|Y \cap h|}{|Y|} - \frac{|X \cap h|}{|X|}|$, which unfortunately requires $|X|^{d+1}$ time to simply enumerate, which would be infeasible for large X. Instead we use techniques [18, 2, 17] which provide guaranteed approximation of this function, designed with spatial anomaly detection in mind. We have set the parameters large enough so the noise in computing Error is insignificant compared to the quantities we are evaluating. We evaluated the accuracy and efficiency of computing ε-samples with 8 different methods.

- There are 3 algorithms based on sampling one element per cell from Matousek's partition algorithm with polygonal cells, using tests created by lines Mat Poly Lines, points Mat Poly Points, or the dual approach Mat Poly Dual.
- We consider 2 algorithms based on Chan's partition algorithm Chan and Chan Simple. Each uses polygonal cells and the dual approach for the test set since this specific type of test set allowed for an optimization in the reweighting step. The Chan variant includes subsampling among cells for purpose of reweighting, while the Chan Simple simply uses all of these cells and does not require the sampling step which in practice was inefficient.
- Then Ham Tree Sample draws samples from the cells of the Willards partitioning; these are also cells of a partition, but with worse theoretical size-accuracy bounds.
- Finally we consider two baselines: random sampling, Random Sample, and another approach Biased-L2 [2] which is a greedy, but slow algorithm which has similar worst case guarantees to random sampling, but achieves better error in practice.
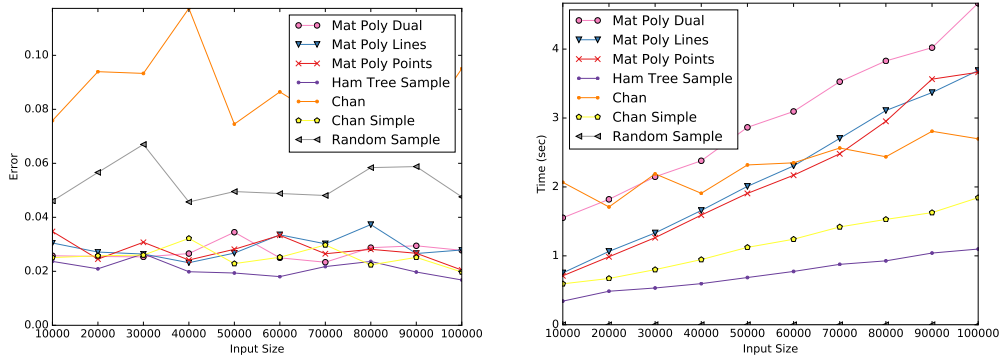
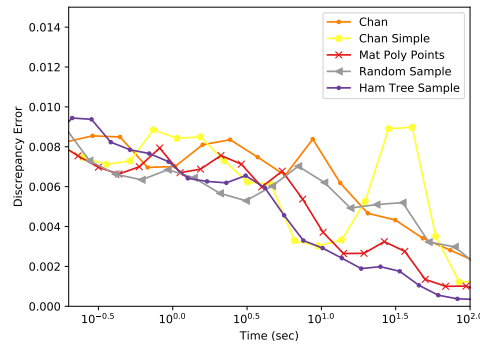**Figure 4** Input size vs. Time and Error using the default parameters.
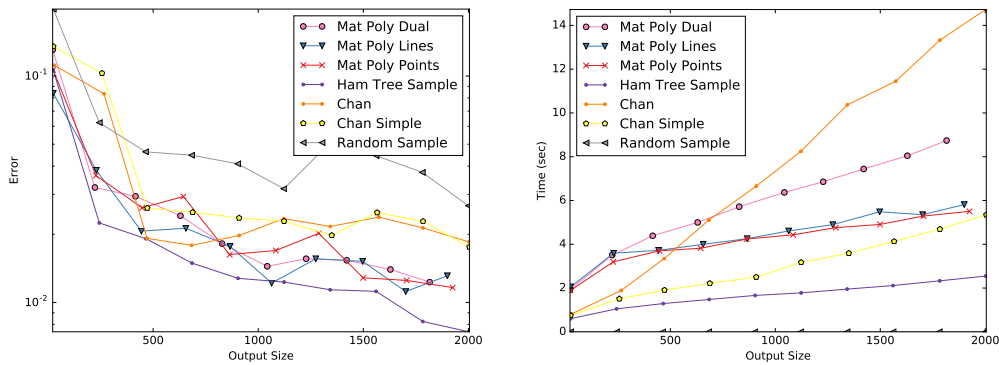
Figure 5 Output size vs. Time and Error using the default parameters.

In testing these algorithms we can control three parameters: the Branching Factor b, the Input Size n (default $n = 100,000$ sampled from the crime data set), and the Output Size k (default $k = 1,000$). We do not create a sample before creating the partition as analyzed in Theorem 1; we just create the partition on the n points, then sample a point from each cell for the ε-sample. The branching factor only effects ComputePartition-Mat (default $b = 16$) and ComputePartition-Chan (default $b = 22$) and is constant for the execution of the algorithm.

Sample Evaluation Results. We do not plot Biased-L2 since it was quite slow as a function of the Output Size. For $k = 51$ it required 360 seconds which was already more than a factor $100\times$ slower than any other algorithm, and became nearly intractable for $k > 100$. We do note however that its measured Error on small k is competitive with the best of our partitioning based methods.

Figure 3 shows how Branching Factor b affects the time and error. Matoušek-based algorithms seem to gradually decrease in Error, but the trend is very small. For Chan Simple, the Error encounters a phase shift at around $b = 25$, where the error suddenly becomes significantly worse for larger b, probably as an effect of the data set size. The timing is fairly unaffected by b for Chan-based algorithms, but increases noticeably and linearly for the Matoušek based algorithms. We conclude that $b = 22$ is a good choice for Chan-based algorithms and $b = 16$ is a good choice for Matoušek based algorithms.

Figure 4 shows the Input Size relationship to time and Error. As prescribed by the theory, Input Size has no noticeable effect on Error. Moreover, also as expected **the runtime of all algorithms scale linearly with Input Size**.

Figure 5 plots the Output Size against the time and Error. As Output Size increases, as expected the error for all methods decreases, note that Error is plotted on a log-scale. The Ham Tree Sample and Mat Poly Lines achieve the smallest Error, with Ham Tree Sample doing the best, and all proposed methods appear to improve upon the old default Random Sample in terms of Error. In particular with Output Size $k = 1000$ both Mat Poly Points and Ham Tree Sample have Error ≈ 0.01 while Random Sample has Error ≈ 0.04. For the Matoušek-based partitioning algorithms, the choice of test set does not have much effect on Error, and perform slightly worse than those based on Chan's partitioning.

Moreover, as Output Size increases the observed run time of all algorithms increases at most linearly. In some cases (e.g., Ham Tree Sample and Mat Poly Lines) the increase is sublinear as these are hierarchical methods, and the largest cost is incurred at the top of the hierarchy. Here as in other plots, we observe that Random Sample is absurdly faster than any other approach. However, even for Output Size $k = 1000$, our methods Ham Tree Sample, Chan Simple, and Mat Poly Points take only about 1, 2.5, and 4 seconds, respectively.

Spatial Anomaly Detection Evaluation. As a concrete demonstration of the usefulness of efficient ε-samples in practice, we apply our new algorithms to a framework for approximately detecting spatial anomalies – maximizing the spatial scan statistic [14]. Specifically each point is endowed with two measures ($b(x)$ the baseline quantity like population and $m(x)$ the measured quantity like disease instance), and let $m(h)$ and $b(h)$ be the fraction of all measured and baseline counts within range $h \in \mathcal{H}_d$, respectively. The main computational problem of exact scan statistics is to find $h^* = \arg \max_{h \in \mathcal{H}_d} \Phi(h)$ where for simplicity we use $\Phi(h) = \phi(m(h), b(h)) = |m(h) - b(h)|$. Approximate scan statistics [18, 17] depend on creating two samples an ε-net which approximates the density of the regions and an ε-samples which approximates the density of points. Together this allows the algorithm to find a $\hat{h}$ where $|\Phi(\hat{h}) - \Phi(h^*)| \leq \varepsilon$; and this is still statistically powerful [18]. In particular, we consider an algorithm for $\hat{h}$ which runs in time $O(n + \frac{1}{\varepsilon}k \log \frac{1}{\varepsilon} + T(n, k))$, where k is the ε-sample size and $T(n, k)$ its construction time. We fix ε to be approximately .0025 which corresponds approximately to an ε-net of size 400. and vary only k. We find approximate anomalies on the crime data set with a particular $h' \in \mathcal{H}_d$ chosen and points chosen, so that $\Phi(h')$ will be anomalously large. Namely we plant a region containing .02 fraction of the points, where in that region points are in the measured set with probability of .7 and baseline set of .3 and outside with probability .5 and .5 respectively. In Figure 6 we plot Discrepancy Error $= |\Phi(\hat{h}) - \Phi(h')|$ as a function of the overall runtime of the algorithms. Note that $\Phi(h^*) \geq \Phi(h')$, so it is possible to find a $\Phi(\hat{h}) \geq \Phi(h')$, but $\Phi(h')$ serves as a

useful proxy. We find that Ham Tree Sample generally outperforms Random Sample; for instance for 0.003 error, Ham Tree Sample takes 10 seconds to Random Sample's 50 seconds. Mat Poly Points also usually performs better than Ham Tree Sample, while Chan and Chan Simple perform comparably to random sampling, albeit with high variance, even though their sampling procedure is hundreds of times slower.

Conclusion. Overall we recommend Ham Tree Sample for computing ε-samples if moderate computing beyond random sampling can be tolerated. This method significantly reduces the size and error versus random sampling, and is not difficult to implement.

References

1   Crimes in Chicago. `https://www.kaggle.com/currie32/crimes-in-chicago`, 2017.
2   Huseyin Akcan, Herve Bronnimann, and Robert Marini. Practical and efficient geometric ϵ-approximations. *Proceedings of the 18th Canadian Conference on Computational Geometry*, pages 120–125, 2006.
3   J. Ralph Alexander. Geometric methods in thge theory of uniform distribution. *Combinatorica*, 10:115–136, 1990.
4   Amitabha Bagchi, Amitabh Chaudhary, David Eppstein, and Michael T. Goodrich. Deterministic sampling and range counting in geometric data streams. *ACM Transactions on Algorithms*, 3(A16), 2007.
5   Nikhil Bansal. Constructive algorithms for discrepancy minimization. In *Proceedings 51st Annual IEEE Symposium on Foundations of Computer Science*, pages 407–414, 2010.
6   Christopher Barker. Pep 485 – a function for testing approximate equality. `https://www.python.org/dev/peps/pep-0485/`, Jan 2015.
7   Timothy M. Chan. Optimal partition trees. In *In: Proc. 26th Annu. ACM Sympos. Comput. Geom*, pages 1–10, 2010.
8   Bernard Chazelle. *The Discrepancy Method*. Cambridge, 2000.
9   Bernard Chazelle and Joel Friedman. A deterministic view of random sampling and its use in geometry. *Combinatorica*, 10:229–249, 1990.
10 Bernard Chazelle and Jiri Matousek. On linear-time deterministic algorithms for optimization problems in fixed dimensions. *Journal of Algorithms*, 21:579–597, 1996.
11 Herbert Edelsbrunner and Emo Welzl. Halfplanar range search in linear space and $o(n^{0.695})$ query time. In 23, editor, *Information Processing Letters*, pages 289–293, 1986.
12 Pavlos S. Efraimidis and Paul G. Spirakis. Weighted random sampling with a reservoir. *Information Processing Letters*, 97(5):181–185, 2006.
13 S. Har-Peled. Constructing planar cuttings in theory and practice. *SIAM J. Comput.*, 29(6):2016–2039, 2000.
14 Martin Kulldorff. A spatial scan statistic. *Communications in Statistics: Theory and Methods*, 26:1481–1496, 1997.
15 Yi Li, Philip M. Long, and Aravind Srinivasan. Improved bounds on the samples complexity of learning. *J. Comp. and Sys. Sci.*, 62:516–527, 2001.
16 Sachar Lovett and Raghu Meka. Constructive discrepancy minimization by walking on the edges. *SIAM Journal on Computing*, 44:1573–1582, 2015.
17 Michael Matheny and Jeff M. Phillips. Computing approximate statistical discrepancy. *CoRR*, abs/1804.11287, 2018. `arXiv:1804.11287`.
18 Michael Matheny, Raghvendra Singh, Liang Zhang, Kaiqiang Wang, and Jeff M. Phillips. Scalable spatial scan statistics through sampling. In *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2016.

19   Jiri Matoušek. Approximations and optimal geometric divide-and-conquer. In *Proceedings 23rd Symposium on Theory of Computing*, pages 505–511, 1991.

20   Jiri Matoušek. Efficient partition trees. *Discrete & Computational Geometry*, 8:315–334, 1992.

21   Jiri Matoušek. Tight upper bounds for the discrepancy of halfspaces. *Discrete and Computational Geometry*, 13:593–601, 1995.

22   Jiri Matoušek. *Geometric Discrepancy*. Springer, 2009.

23   Jiří Matoušek, Chi-Yuan Lo, and William Steiger. Ham-sandwich cuts in rd. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Theory of Computing*, STOC '92, pages 539–545, New York, NY, USA, 1992. ACM.

24   Nimrod Megiddo. Partitioning with two lines in the plane. *Journal of Algorithms*, 6(3):430–433, 1985.

25   Raimund Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Computational Geometry*, 1:51–64, 1991.

26   Subhash Suri, Csaba D. Tóth, and Yunhong Zhou. Range counting over multidimensional data streams. In *Proceedings 20th Symposium on Computational Geometry*, pages 160–169, 2004.

27   Vladimir Vapnik and Alexey Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theo. of Prob and App*, 16:264–280, 1971.

28   Ron Wein, Eric Berberich, Efi Fogel, Dan Halperin, Michael Hemmer, Oren Salzman, and Baruch Zukerman. 2D arrangements. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.12 edition, 2018.

29   D. E. Willard. Polygon retrieval. In 11, editor, *SIAM Journal of Computing*, pages 149–165, 1982.