Parallel Monotonicity Reconstruction

Michael Saks*
saks@math.rutgers.edu
Dept. of Mathematics
Rutgers University

C. Seshadhri
csesha@cs.princeton.edu
Dept. of Computer Science
Princeton University

Abstract

We investigate the problem of monotonicity reconstruction, as defined in [3], in a parallel setting. We have oracle access to a nonnegative real-valued function f defined on domain $[n]^d = \{1, \ldots, n\}^d$. We would like to closely approximate f by a monotone function g. This should be done by a procedure (a *filter*) that given as input a point $x \in [n]^d$ outputs the value of $g(x)$, and runs in time that is highly sublinear in n. The procedure can (indeed must) be randomized, but we require that all of the randomness be specified in advance by a single short random seed. We construct such an implementation where the the time and space per query is $(\log n)^{O(1)}$ and the size of the seed is polynomial in $\log n$ and d. Furthermore the distance of the approximating function g from f is at most a constant multiple of the minimum distance of any monotone function from f.

This implementation allows for parallelization: one can initialize many copies of the filter with the same short random seed, and they can autonomously handle queries, while producing outputs that are consistent with the same approximating function g.

*Supported in part by NSF grant CCR-0515201.

1 Introduction

1.1 Online property reconstruction The process of assembling large data sets is prone to varied sources of error, such as measurement error, replication error, and communication noise. Error correction techniques (i.e. coding) can be used to reduce or eliminate the affects of some sources of error, but often some residual errors may be unavoidable. Despite the presence of such inherent error, the data set may still be very useful.

One problem in using such a data set is that even small amounts of error can significantly change the behavior of algorithms that act on the data. For example, if we do a binary search on an array that is supposed to be sorted, a few erroneous entries may lead to behavior that deviates significantly from the "correct" behavior.

This is an example of a more general situation. We have a data set that ideally should have some specified structural property, i.e., a list of numbers that should be sorted, a set of points that should be in convex position, or a graph that should be a tree. Algorithms that run on the data set may rely on this property. A small amount of error may destroy the property, and result in the algorithm producing wildly unexpected results, or even crashing. In these situations, a small amount of error may be tolerable but only if the structural property is maintained.

These considerations motivated the formulation of the *online property reconstruction* model, which was introduced in [3]. We are given a data set, which we think of as a function f defined on some domain Γ. Ideally, f should have a specified structural property $\mathcal{P}$, but this property may not hold due to unavoidable errors. We wish to construct *online* a new data set g such that (1) g has property $\mathcal{P}$ and (2) $d(g, f)$ is small, where $d(g, f)$ is the fraction of values $x \in \Gamma$ for which $g(x) \neq f(x)$.

How small should $d(g, f)$ be in condition (2)? Define $\varepsilon_f = \varepsilon_f(\mathcal{P})$ to be the minimum of $d(h, f)$ over all h that satisfy $\mathcal{P}$. Of course, ε_f is a lower bound on the

deviation of g from f. The *error blow-up* of g is the ratio $d(g,f)/\varepsilon_f$. This error blow-up can be viewed as the price that is paid in order to restore the property $\mathcal{P}$, and we want this to be a not too large constant.

An offline reconstruction algorithm explicitly outputs such a g on input f. In the context of large data sets, the explicit construction of g from f requires a considerable amount of computational overhead (at least linear in the size of the data set). For this reason, [3] considered online reconstruction algorithms. Such an algorithm, called a *filter*, gets as input $x_1, x_2, \ldots$ of elements of Γ presented one at a time and must output the sequence of values $g(x_1), g(x_2), \ldots$ where $g(x_i)$ is produced in response to x_i, before knowing x_{i+1}. The filter can access the function f via an oracle which given $y \in \Gamma$ answers $f(y)$. The aim is to design a filter which, given any permutation of Γ, outputs a function g satisfying (1) and (2) above and furthermore produces each successive $g(x_i)$ quickly, i.e., in time much smaller than $O(|\Gamma|)$.

In [3], a filter for the *monotonicity property* was given. In this setting, the domain Γ is the set $[n]^d = \{(j_1, \ldots, j_d) : j_i \in [n]\}$, where $[n]$ denotes the set $\{1, 2, \ldots, n\}$. $[n]^d$ is considered to be partially ordered under the component-wise (product) order. A function f defined on Γ is *monotone* if $x \leq y$ implies $f(x) \leq f(y)$. The filter they constructed satisfies condition (1), has error blow-up that is bounded above by $2^{O(d)}$ (independent of n), and answers each successive query in time $(\log n)^{O(d)}$.

1.2 Parallel property reconstruction The filter for monotonicity proposed in [3] has the following general structure. For each successive query x_j, the filter performs a randomized algorithm to compute $g(x_j)$. This algorithm accesses f, and also needs to access the answers $g(x_i)$ for $i < j$ to the queries asked previously. In particular, the function g produced may depend on both the order of the queries as well as the random bits used by the algorithm.

This general structure for filters has two potential drawbacks: (1) It requires the storage of all previous queries and answers, thus incurring possibly significant space overhead for the algorithm, (2) It does not support a parallel implementation in which multiple copies of the filter, having read-only access to f are able to handle queries independently while maintaining mutual consistency.

In this paper, we propose the following strengthened requirements for a filter. A *parallel filter* for reconstructing property $\mathcal{P}$ is an algorithm A that has oracle access to a function f on domain Γ (the "data set") and to an auxiliary random string ρ (the "random seed"), and takes as input $x \in \Gamma$. For fixed f and ρ, A runs deterministically on input x to product an output $A_{f,\rho}(x)$. We want A to satisfy the following properties:

1. For each f, with high probability (with respect to the choice of ρ), the function $A_{f,\rho}$ should satisfy $\mathcal{P}$.
2. For each f, with high probability (with respect to the choice of ρ), the function $A_{f,\rho}$ should be "suitably close" to f.
3. For each x, $A_{f,\rho}$ on x can be computed very quickly.
4. ρ should be "much smaller" than $|\Gamma|$.

A parallel filter can be used, trivially, as an online filter. In addition, such a filter affords an obvious parallel implementation: generate one random seed, and give the same random seed to each of the copies of the filter. Since $A_{f,\rho}$ is deterministic all of the parallel copies will behave identically.

For a parallel filter, there are three parameters of interest: the error blow-up, the time per query and the number of random bits needed for ρ, that is, to initialize the filter. The memory used by a parallel filter is bounded by the sum of the length of ρ and the maximum time of a single query. By keeping these both small (e.g., much smaller than $|\Gamma|$) we obtain an online filter which uses little auxiliary space.

Having formulated the notion of a parallel filter, one natural question that arises is: Is there a parallel filter for monotonicity (or even one that uses sublinear space overall) with performance comparable to that of the online filter given in [3]?

1.3 Our results In this paper, we construct a parallel filter for monotonicity for functions defined on $[n]^d$ with the following performance:

- The time per query is $(\log n)^{O(d)}$.
- The error blow-up is $2^{O(d^2)}$, independent of n.
- The number of random bits needed to initialize the filter is $(d \log n)^{O(1)}$.

Comparing our filter to the online filter for monotonicity of [3], we see that our filter achieves parallelism while having query time and error blow-up that are similar to (but not quite as good) as those obtained by [3]. We have not tried to optimize the exact constants for error blow-up and the exponents in running time.

1.4 Related Work The idea of reconstruction is connected to the issue of *distance* of an object to a given property. There has been a large amount of work done on property testing [10, 14], which deals with this distance parameter. The problem of monotonicity in the context of property testing has been studied in [5, 6, 9]. Many testers have been given for a wide variety of combinatorial, algebraic, and geometric problems [7, 8, 13, 14]. More closely related to reconstruction is the notion of tolerant property testing [12] which deals with actually estimating the distance in sublinear time. Sublinear algorithms for determining the distance of a function to monotonicity have been given in [2, 12]. The tester in [12] uses a data structure of intervals, something that we also use (although the data structure that we use is completely different).

1.5 Preliminaries As stated in the introduction, we consider data sets as functions defined on a fixed finite domain Γ. A property $\mathcal{P}$ is a set of functions defined on Γ. The distance between two functions f and g, denoted $d(f, g)$ is the fraction of $x \in \Gamma$ for which $f(x) \neq g(x)$. For a function f and a property $\mathcal{P}$, the distance of f to $\mathcal{P}$, $\varepsilon_f = \varepsilon_f(\mathcal{P})$ is the minimum of $d(f, h)$ for $h \in \mathcal{P}$.

For a positive integer m, $[m]$ denotes the set $\{1, 2, \ldots, m\}$. Throughout this paper, $\Gamma = [n]^d = \{(x_1, \ldots, x_d) : \forall i \in [d], x_i \in [n]\}$ for some integers n and d. We fix n and d, and assume, without (much) loss of generality that $n = 2^k$ where k is a positive integer. Γ is partially ordered with respect to the product relation: $x \leq y$ if and only if $x_i \leq y_i$ for all $i \in [d]$.

We consider functions mapping Γ to the nonnegative reals (for simplicity of presentation). Such a function f is *monotone* if $f(x) \leq f(y)$ whenever $x \leq y$.

Elements of Γ are called *points*. Points are generally denoted by lower case letters, sets of points are denoted by upper case letters and sets of sets of points by caligraphic letters.

As defined in the introduction, a parallel filter uses randomness only in the choice of the string ρ that initializes the filter. All probability statements are made with respect to the choice of this string ρ. In general, when we say that an event occurs with *low probability* we mean that its probability is $1/|\Gamma|^{\omega(1)}$, i.e. superpolynomially small in $|\Gamma|$. Similarly, a high probability event is one having probability $1 - (1/|\Gamma|^{\omega(1)})$.

2 A high level view of the filter

Given a function f on $[n]^d$, we say that a subset $S \subseteq [n]^d$ is *f-admissible* if the restriction of f to S is monotone.

The first component of the filter is a subroutine, *Sift* which takes as input a point $x \in [n]^d$ and returns *accept* or *reject*. With high probability it satisfies the following properties -

S1 : The set of accepted points is f-admissible.

S2 : The set of rejected points has size at most $C_1(d)\varepsilon_f n^d$ where $C_1(d)$ is independent of n. (In our algorithm $C_1(d) = 2^{O(d^2)}$.)

S3 : *Sift* runs in time $(\log n)^{O(d)}$.

The second part of the filter is a function *Build* that takes as input a point $x \in [n]^d$ and (using the suboutine *Sift*) returns a set $Rep(x)$ of *representative points* for x. This subroutine satisfies -

B1 : Every point in $Rep(x)$ is less than or equal to x.

B2 : Every point in $Rep(x)$ is accepted by *Sift*.

B3 : For all x, y with $x \leq y$, for each $z \in Rep(x)$ there is a point $z' \in Rep(y)$ such that $z \leq z'$.

B4 : With high probability, the number of points x for which $x \notin Rep(x)$ is at most $C_2(d)$ times the number of points rejected by *Sift*, where $C_2(d)$ is independent of n. (In our algorithm, $C_2(d) = 2^{O(d^2)}$.)

B5 : *Build* runs in time $(\log n)^{O(d)}$.

Our filter is then defined from *Build* as follows:

> Given x, $g(x) = \max\{f(z) : z \in Rep(x)\}$. If $Rep(x)$ is empty, then $g(x) = 0$.

Let us see that properties [S1]-[S3] and [B1]-[B5] ensure that the filter has the required properties.

To see that g is monotone, let $x \leq y$. By the definition of g, $g(x) = f(z)$ for some $z \in Rep(x)$. By property [B3], there is a $z' \in Rep(y)$ such that $z \leq z'$. By [B2], both z and z' are accepted by *Sift* so by [S1], $f(z) \leq f(z')$. By the definition of $g(y)$ we have $g(y) \geq f(z')$, so $g(y) \geq f(z') \geq f(z) = g(x)$ as required.

To get an upper bound on the number of points for which $g(x) \neq f(x)$, note that $x \in Rep(x)$ implies $g(x) = f(x)$ and so by [B4], the number of points with $g(x) \neq f(x)$ is at most $C_2(d)$ times the number of points rejected by *Sift* which, by [S2] is at most $C_2(d)C_1(d)\varepsilon_f n^d$ as required.

Finally, the desired bound on the running time of the filter follows from [B5] and [S3].

3 The one-dimensional case

In this section, we describe our parallel monotonicity filter for the case $d = 1$.

3.1 The DAG $D(k)$ of intervals For $x, y \in [n]$ we define the interval $[x, y]$ to be the set $\{z \in [n] : x \leq z \leq y\}$. If I is an interval we write $\min(I)$ for its smallest element and $\max(I)$ for its largest element; $\min(I)$ and $\max(I)$ are the *endpoints* of I and $I - \{\min(I), \max(I)\}$ is the *interior* of I. If $\min(I) = 1$, we say I is *left extreme* and if $\max(I) = n$ we say I is *right extreme*.

Our filter makes use of a special set of intervals, which we now define. For integers $i \geq 1$ and $j \geq 0$, we define the interval

$$I_i^j = [j2^{i-1} + 1, (j + 2)2^{i-1}].$$

The set $\{I_i^j : 1 \leq i \leq k, 0 \leq j \leq \frac{n}{2^{i-1}} - 2\}$ is denoted $\mathcal{I} = \mathcal{I}(k)$. The set $\mathcal{I}_i = \mathcal{I}_i(k)$, called the *ith level*, is the set of intervals $\{I_i^j \,|j \geq 0\}$. The ith level contains $\frac{n}{2^{i-1}} - 1$ intervals each of size 2^i.

An interval I_i^j is said to be *even* if j is even and *odd* if j is odd. Notice that the set of even intervals at level i comprise the natural partition of $[1, n]$ into $\frac{n}{2^i}$ intervals of size 2^i, while the odd intervals at level i partition the interval $[2^{i-1}, n - 2^{i-1}]$ into $\frac{n}{2^i} - 1$ intervals of size 2^i.

We define a DAG $D = D(k)$ with vertex set $\mathcal{I}$, with an edge from interval I to interval J if $J \subseteq I$ and they belong to adjacent levels. The DAG $D(k)$ has k levels and $2^{k+1} - k - 2$ vertices ($2^{k+1-i} - 1$ at level i). The interval $I_n^0 = [1, n]$ is the unique interval of in-degree 0 and is called the *root* of D, and the intervals in level 1 (those having size 2) have out-degree 0 and are called *leaves*.

Figure 1 shows the the DAG $D(3)$ (with edges pointing downwards). Note that for $r < s$ the sub-DAG consisting of the first r levels of $D(s)$ is isomorphic to $D(r)$.

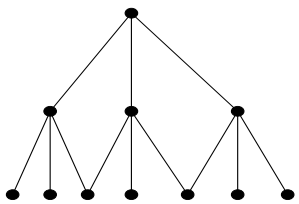

Figure 1: *The DAG* $D(3)$

Suppose I and J are intervals and there is an edge from I to J. We say that I is the *parent* of J and J is a *child* of I. We must have $|I| = 2|J|$. Furthermore, parent-child relationships fall into three categories:

- $\min(I) = \min(J)$. Here we say that J is the *left child* of I and I is the *right parent* of J.
- $\max(I) = \max(J)$. Here we say that J is the *right child* of I and I is the *left parent* of J.
- $\min(J) = \min(I) + |J|/2$ and $\max(J) = \max(I) - |J|/2$. Here we say that J is the *central child* of I and I is the *central parent* of J.

Every interval at level $i \geq 2$ has exactly 3 children, one left child, one central child and one right child.

Every non-root interval I has either one or two parents. If I is an odd interval, then it has one parent, and that parent is a central parent. If I is an even interval and neither left-extreme nor right-extreme then it has one left parent and one right parent. If I is left-extreme it has only a right parent, and if it is right-extreme it has only a left parent.

We now give some definitions that will be heavily used in the following sections. Let I be an interval and x be a point.

- **Non-left parent of I:** We say that interval J is a *non-left parent* of I if it is either a right parent or a central parent of I. We observe that if I is right-extreme, then it has no non-left parent, but otherwise I has a *unique non-left parent*, which we denote by $nonleftpar(I)$.
- **Non-left path of I:** We define the *non-left path of* I, $path(I)$, to be the sequence $I = I_1, I_2, \ldots, I_t$ where for $1 \leq j < t$, $I_{j+1} = nonleftpar(I_j)$ and I_t is right-extreme.
- **Non-left ancestor of I:** We say that J is a *non-left ancestor* of I if $J \in path(I)$.
- **I is to the left of x:**We say that interval I is *to the left* of point x if $y \leq x$ for all $y \in I$.
- **I is left-maximal for x:** We say that I is *left-maximal* for x if I is left of x and x belongs to the interior of every non-left ancestor of I. For each x there are at most two intervals at each level that are left maximal for x. The intervals that are left-maximal for n are precisely those that are right-extreme.
- *upper*(I)**:** For I of size at least 4, we define *upper*(I) to be the subinterval of I consisting of the greatest $|I|/4$ points of I. Note that if I is a non-left parent of an interval J, then every point in J is less than every point in *upper*(I).

Some useful properties of $\mathcal{I}$ are stated below.

I1: Every interval in $\mathcal{I}_i$ has size 2^i.
I2: Every node of D has at most two ancestors at each level; in particular each point of $[n]$ belongs to at most two intervals at each level. It follows that every node has $O(\log n)$ ancestors.

I3: If $I_1 \cap I_2 \neq \emptyset$ and $|I_1| < |I_2|$ then $I_1 \subset I_2$.
I4: For any two points $x < y$. there is an interval of $\mathcal{I}$ containing both of size at most $4|[x, y]|$.
I5: If $\mathcal{A}$ is a containment-free subset of $\mathcal{I}$ (i.e., no member of $\mathcal{A}$ contains another), then each point x belongs to at most 2 intervals of $\mathcal{A}$.

Properties [I1], [I2] and [I3] are obvious. For property [I4], let t be the integer such that $2^t \leq |[x, y]| < 2^{t+1}$, and let j be the largest integer such that $j2^{t+1} \leq x$. then $I^j_{t+2} = [j2^{t+1}, (j + 2)2^{t+1}]$ contains x and y and has length $2^{t+2} \leq 4|[x, y]|$.

For property [I5], note that [I3] and $\mathcal{A}$ being containment-free, imply that all sets in $\mathcal{A}$ that contain x must have the same size; hence there are at most 2 of them.

We introduce another important definition.

- θ-dense : If S is a subset of $[n]$ and $\theta \in [0, 1]$, we say that S is θ-dense in I if $|S \cap I| \geq \theta|I|$. For $S \subseteq [n]$, define $\Lambda_\theta(S)$ to be the union of all $I \in \mathcal{I}$ that are θ-dense.

LEMMA 3.1. *For any subset S, $\Lambda_\theta(S) \leq \frac{2}{\theta}|S|$.*

Proof. [of Lemma 3.1] Let $\mathcal{J}$ be the subset of all $I \in \mathcal{I}$ for which S is θ-dense in I. Let $\mathcal{A}$ be the set of maximal members of $\mathcal{J}$. Then

$$|\bigcup_{I\in\mathcal{J}} I| = |\bigcup_{I\in\mathcal{A}} I| \leq \sum_{I\in\mathcal{A}} |I| \leq \frac{1}{\theta}\sum_{I\in\mathcal{A}} |S \cap I|.$$

Since $\mathcal{A}$ is a containment-free subcollection of $\mathcal{I}$, by property [I5], each element of S belongs to at most 2 members of $\mathcal{A}$. The final sum is at most $2|S|$, giving the upper bound of $\frac{2}{\theta}|S|$ on $|\bigcup_{I\in\mathcal{J}} I|$. □

3.2 The random seed Our random seed will be interpreted as an n-ary string, i.e. a sequence selected independently from the set $[n]$. In the one-dimensional case, the seed is viewed as a pair of sequences $(s(1), \ldots, s(t))$ and $(r(1), \ldots, r(t))$ of length $t = c \log^2 n$ elements of $[n]$. Thus we use $2t \log^2 n$ bits of randomness.

Using these strings, we define for each interval I the sequences $s \mod I$ and $r \mod I$ in I^t, respectively to be $(s(1) \mod I, \ldots, s(t) \mod I)$ and $(r(1) \mod I, \ldots, r(t) \mod I)$.

3.3 The subroutine *Sift* in one dimension The subroutine *Sift* takes as input a point x and outputs *accept* or *reject* in such a way that the set of accepted points is f-admissible. Anticipating what we need in the multidimensional case, we will define *Sift* in greater generality than we need for the one-dimensional case. We assume that the set of points is initially partitioned into sets *Eligible* and *Ineligible* points. This partitioning is provided by a subroutine which, on input x, tests whether $x \in$ *Eligible*. We will show that *Sift* satisfy the following conditions:

S1′: Only eligible points are accepted and, with high probability, the set of accepted points is f-admissible.

S2′: The set of rejected points has size at most $C_1(\varepsilon_f n^d + |Ineligible|)$ for some constant C_1.

S3′: *Sift* runs in time $(\log n)^{O(1)}T$, where T is an upper bound on the time to test membership in *Eligible*.

When we use this subroutine for the one-dimensional filter we'll take *Eligible* $= [n]$ and *Ineligible* $= \emptyset$, but for the higher dimensional filter, we'll need the more general version.

Let us define a *violation* to be a pair (x, y) such that $x < y$ and $f(x) > f(y)$, or $x \in$ *Ineligible* or $y \in$ *Ineligible*. We also say *x is a violation with y* (and vice versa).

DEFINITION 3.1. *For a point x and $A \subseteq [n]$, define violations(x, A) to be the fraction of points in A that are in violation with x.*

For $\mu > 0$, we say that x is μ-sound *if $x \in$ Eligible and for all $I \in \mathcal{I}$ such that $x \in I$, violations$(x, I) < \mu$. Otherwise we say that x is μ-unsound.*

The following lemma is similar to a result from [3].

LEMMA 3.2. *1. Every μ-sound point belongs to Eligible.*

2. For $\mu \leq 1/8$, the set of μ-sound points is f-admissible.

3. For $\mu \leq 1/2$, the number of μ-unsound points is at most $2\mu^{-1}(\varepsilon_f n + |Ineligible|)$.

Proof. The first part is immediate from the definition of μ-sound.

For the second part, it suffices to show that for each violation (x, y) $(x < y)$ at least one of x and y is $1/8$-unsound. This is immediate if either x or y belongs to *Ineligible*, so assume that $x, y \in$ *Eligible*. Since $f(x) > f(y)$, for each $z \in [x, y]$ at least one of (x, z) and (y, z) is a violation. Thus *violations*$(x, [x, y])$ + *violiations*$(y, [x, y]) \geq 1$. By property [I4], there is an interval $I \in \mathcal{I}$ of size at most $4|[x, y]|$ that contains $[x, y]$,

so $violations(x, I) + violations(y, I) \geq 1/4$, and so one of them is at least 1/8.

For the third part, by definition of ε_f, there is a subset S of size $\varepsilon_f n$ such that $[n] - S$ is f-admissible. By Lemma 3.1, $\Lambda_\mu(S \cup Ineligible)$ has size at most $2\mu^{-1}(\varepsilon_f n + |Ineligible|)$, so it suffices to show that every μ-unsound point x belongs $\Lambda_\mu(S \cup Ineligible)$. This is true for $x \in S \cup Ineligible$ since x belongs to an interval $I \in \mathcal{I}$ of size 2, and $S \cup Ineligible$ is μ-dense in I. If $x \notin S \cup Ineligible$, then the set of points violating x is a subset of $S \cup Ineligible$. Since x is μ-unsound there is an interval I containing x that contains at least $\mu|I|$ points violating x. Therefore $S \cup Ineligible$ is μ-dense in I and $x \in \Lambda_\mu(S \cup Ineligible)$. □

We define a subroutine **Violations** which takes as input a point x and interval $I \in \mathcal{I}$ and returns an estimate of $violations(x, I)$, which is obtained by computing the fraction of points in the sample $s \mod I$ (as described in Section 3.2) that are violations with x. Clearly **Violations** runs in time $(\log n)^{O(1)}$. We say that **Violations** *fails* for x and I if $|\textbf{Violations}(x, I) - violations(x, I)| > .01$. We say that **Violations** *fails* if it fails for some x, I where $I \in \mathcal{I}$ and $x \in I$, and we say it succeeds otherwise.

The subroutine *Sift* works as follows. On input x, sift rejects x if for some $I \in \mathcal{I}$ such that $x \in I$, $\textbf{Violations}(x, I) \geq .11$, and accepts x otherwise. Since there are $O(\log n)$ intervals of $\mathcal{I}$ containing x, and the sequence s of samples has length $t = (\log n)^{O(1)}$, we conclude that the running time of *Sift* is $(\log n)^{O(1)}$.

We say that *Sift* succeeds provided that:

- Every .1-sound point is accepted
- Every 1/8-unsound point is rejected.

From Lemma 3.2, we deduce that if *Sift* succeeds, then [S1],[S2],[S3] stated in Section 2 hold.

Corollary 3.1. *If* Sift *succeeds then:*

1. *The set of points accepted by* Sift *is f-admissible.*
2. *The number of points rejected by* Sift *is at most* $20(\varepsilon_f n + |Ineligible|)$.

Finally, we observe that the probability that *Sift* fails is very small.

Lemma 3.3. *The probability that* Sift *fails is* $(n \log n)2^{-\Omega(t)}$, *where t is the length of the random sequence s.*

Note that for the given choice of t, this is $n^{-\omega(1)}$.

This is proven using a straightforward Chernoff bound argument (proof in full version).

3.4 The procedure *Build*: 1-dimensional case

We now turn to the description of the procedure *Build* (for the 1-dimensional case). We will use the procedure *Sift* with the trivial subroutine *Eligible* that accepts every point.

Recall that *Build* is supposed to take as input a point x and return a set $Rep(x)$ consisting of (some) eligible points less than or equal to x. The main ingredients are a pair of procedures $Sample$ and $Refine$. Each takes as input an interval I and returns a small subset of points of I.

$Sample(I)$ is defined as follows. Consider t, as defined in Section 3.2. If $|I| \leq t$, $Sample(I) = I$. If $|I| > t$, $Sample(I)$ is the set of points in the sequence $r \mod I = (r(1) \mod I, \ldots, r(t) \mod I)$, as described in Section 3.2.

Before defining $Refine$, we need a few definitions. For a set S, let $Sift(S)$ denote the set of points of S accepted by *Sift*.

Now we define $Refine$. On input I, determine $Sample(J)$ for each $J \in path(I)$. If $Sift(Sample(J)) \cap upper(J) \neq \emptyset$ for all $J \in path(I)$ then $Refine(I) = Sift(Sample(I))$. Otherwise, $Refine(I) = \emptyset$. An easy reverse induction on the level of J shows that if $Refine(I)$ is nonempty then $Refine(J) = Sift(Sample(J))$ for all non-left ancestors J of I, and hence $Refine(J) \cap upper(J) \neq \emptyset$ for all non-left ancestors of I.

Finally we define the procedure *Build*. On input x, the output $Rep(x)$ is defined to be the union over all intervals I that are left-maximal with respect to x, of $Refine(I)$. (For the sake of definition and implementation, we can redefine $Refine(I)$ to simply be the largest point in the present $Refine(I)$. We describe $Refine(I)$ as a set (1) to make the proof of [B3] more convenient, (2) to prepare for the extension to the multi-dimensional case where $Refine(I)$ can not be taken to be a single point.

We now verify properties [B1]-[B5] of Section 2. Properties [B1] and [B2], that all points in $Rep(x)$ are less than or equal to x, and that each point in $Rep(x)$ is accepted by *Sift*, are immediate from the definition of $Rep(x)$.

To prove [B3], let x, y be arbitrary points in $[n]$ with $x \leq y$ and let $z \in Rep(x)$. We must show that there is a $z' \in Rep(y)$ with $z \leq z'$. Since $z \in Rep(x)$, there is an interval I that is left-maximal for x such that $z \in Refine(I)$. If I is left-maximal for y then we take $z' = z$. Otherwise, let J be the largest interval in $path(I)$ that is to the left of y ; by definition J is left maximal for y. Since $Refine(I) \neq \emptyset$, $Refine(J) \cap upper(J) \neq \emptyset$ so we can select $z' \in Refine(J) \cap upper(J)$. Since J is a non-left ancestor of I, every point in I is less than every

point in $upper(J)$ (this follows by an easy induction on $|J|$), so $z \leq z'$.

Next we prove [B4], bounding the number of points for which $x \notin Rep(x)$. We say that *Sample* fails for interval I if at least half the points of $upper(I)$ are accepted by *Sift*, but no points in $Sample(I) \cap upper(I)$ are accepted by *Sift*. We say that *Sample* fails if *Sample* fails for some interval I of size at least 4, and *Sample* succeeds if it succeeds for all intervals I of size at least 4, that is, if for each such interval I, if at least $|I|/8$ points of $upper(I)$ are accepted by *Sift*, then $Sample(I) \cap upper(I) \neq \emptyset$.

PROPOSITION 3.1. *Sample succeeds with high probability.*

The proof is given in the full version of the paper.

By Proposition 3.1, *Sample* succeeds with high probability. Assuming that *Sample* succeeds, we prove an upper bound on the number of points x for which $x \notin Rep(x)$. For each $x \in [2, n]$, the interval $[x-1, x]$ is left-maximal for x. If $x \notin Rep(x)$ then $x \notin Refine([x-1, x])$. Therefore either *Sift* rejects x or there is a non-left ancestor J of $[x-1, x]$ such that $Sift(Sample(J)) \cap upper(J) = \emptyset$. Since *Sample* succeeds, fewer than half the points in $upper(J)$ are accepted by *Sift*, which means at least $1/8$ of the points of J are rejected by *Sift*. Letting *Reject* be the set of points rejected by *Sift*, then the set of points for which $x \notin Rep(x)$ is a subset of $\Lambda_{1/8}(Reject)$. By Lemma 3.1, the number of such points is at most $16|Reject|$.

The running time of *Build* is bounded as follows. For any x, there are at most $2 \log n$ intervals (at most 2 on each level) that are left-maximal with respect to x. For each such I, we compute $Refine(I)$ which involves computing $Sample(J)$ for each of the at most $\log n$ non-left ancestors J of I and calling *Sift* for each of the points in $Sample(J)$. Thus the running time of *Build* is essentially the cost of $O((\log n)^5)$ calls to *Sift*, so the overall cost is $(\log n)^{O(1)}$. (No attempt has been made to optimize the exponent of $\log n$ in this description.)

This completes the description and proof of correctness for the filter in the 1-dimensional case.

4 A filter for multidimensional data

4.1 Boxes and lines For $x, y \in [n]^d$, $[x, y]$ denotes the set $\{z : x \leq z \leq y\}$. This set is a product of (1-dimensional) intervals $[x_1, y_1] \times \cdots \times [x_d, y_d]$ and is called a *box*. For a box $B = [x, y]$, we write $B = B_1 \cdots \times \cdots B_d$ where $B_r = [x_r, y_r]$ is the interval obtained by projecting B onto the rth coordinate axis.

A box B is *degenerate* in direction r if $|B_r| = 1$, *non-degenerate* in direction r if $|B_r| > 1$, and *spanning* in direction r if $B_r = [1, n]$.

An *r-line* is a box that is spanning in dimension r and degenerate in every other dimension. The r-lines partition $[n]^d$ into n^{d-1} sets, each of size n. We say that $x \leq_r y$ if x, y lie in the same r-line and $x \leq y$. The r-line passing through x is denoted by $x^{(r)}$.

There is a natural bijection between an r-line L and the set $[n]$ given by $x \in L \leftrightarrow x_r$. For $j \in [n]$ we write j_L for the corresponding point on L, and for $S \subseteq [n]$ we write S_L for the corresponding subset of L. Define $\mathcal{I}_L$ to be the set of I_L for $I \in \mathcal{I}$, $\mathcal{I} = \mathcal{I}(k)$ was defined in Subsection 3.1.

4.2 The random seed The random seed (which consists of independent uniformly random elements selected from $[n]$) is divided into $2d$ sequences $s^1, \ldots, s^d, r^1, \ldots, r^d$, each of length $t(d) = cd \log^2 n$ (for some constant c).

4.3 The function *Sift*, multi-dimensional case Define, for $j \in [d]$ a subroutine $Linesift_j$ as follows: On input x, $Linesift_j$ runs the one-dimensional *Sift* on x with respect to the j-line $x^{(j)}$, using the random sample s^j. As with *Sift*, $Linesift_j$ requires an auxiliary procedure $Eligible_j$, which we assume does not use the random sample s^j.

For each j-line L, the analysis of the one-dimensional *Sift* applies to $Linesift_j$. For each line L, one defines the notion of μ-soundness of a point $x \in L$ with respect to the line L in the obvious way. We say that $Linesift_j$ succeeds for a j-line L if -

- Every point that is .1-sound point with respect to L is accepted
- Every point that is 1/8-unsound with respect to L is rejected.

We say that $Linesift_j$ succeeds if it succeeds on every j-line. By Lemma 3.3, the probability that $Linesift_j$ fails for a particular j-line is bounded above by $n \log n 2^{-\Omega(t)}$. Therefore the probability that $Linesift_j$ fails on some j-line is bounded above by $n^d \log n 2^{-\Omega(t)}$. For the selected t this is $n^{-\omega(d)}$.

This holds for any choice of the auxiliary procedure $Eligible_j$, provided that $Eligible_j$ does not depend on the random string s^j.

Next we define $Sift_j$ for $0 \leq j \leq d$. $Sift_0$ accepts every input point x. For $0 \leq j \leq d$, $Sift_j$ is obtained by running $Linesift_j$, taking $Eligible_j$ to be $Sift_{j-1}$.

Finally, *Sift* is defined to be $Sift_d$.

Let $Accepted_j$ (resp., $Rejected_j$) be the set of points accepted (resp., rejected) by $Sift_j$. Let $\mathcal{C}_j$ be the partition of $[n]^d$ into n^{d-j} classes, where points

are assigned to classes according to their last $d-j$ coordinates. Two points x, y in the same j-line with $x <_j y$ is a j-violation if $f(x) > f(y)$ or one of them is in $Ineligible_j = Rejected_{j-1}$.

LEMMA 4.1. *Assume that* Linesift$_1$, Linesift$_2$, ..., Linesift$_d$ *all succeed.*

For each $j \in \{0, \ldots, d\}$,

1. *For each* $C \in \mathcal{C}_j$, *the set* $C \cap Accepted_j$ *is* f*-admissible.*
2. $|Rejected_j| \leq (20^{j+1} - 20)/(19)\varepsilon_f n^d$,
3. *Let* T_j *be the running time of* Sift$_j$. *Then* $T_j \leq (\log n)^{O(j)}$.

The proof is given in the full version of the paper.

COROLLARY 4.1. *With high probability :*

1. *The function* f *restricted to the set of points accepted by* Sift *is monotone.*
2. *The number of points rejected by* Sift *is at most* $20^{d+1}\varepsilon_f n^d$.

4.4 The DAG $\Delta^d(k)$ Before describing the function *Build*, we need some additional definitions.

We consider the set $\mathcal{B} = \mathcal{B}(k)_d$ to be the set of of all boxes of the form $B = B_1 \times \cdots B_d$ where each $B_j \in \mathcal{I}(k)$ (the set of intervals defined for the one-dimensional case). For each $r \in [d]$, we define an equivalence relation on $\mathcal{B}$: for $B, C \in \mathcal{B}$, $B \sim_r C$ if $B_j = C_j$ for all $j \neq r$. For each r-equivalence class $\mathcal{C}$, the mapping taking $B \in \mathcal{C}$ to B_r is a bijection between $\mathcal{C}$ and $\mathcal{I}(k)$.

We define a dag $\Delta = \Delta^d(k)$ on vertex set $\mathcal{B}$ as follows: (B, C) is in Δ if and only if for some $r \in [d]$ $B \sim_r C$ and $(B_r, C_r) \in D$, where D is the DAG defined for the one-dimensional case. In this case we say that B is an r-parent of C. We adapt the terminology from the one-dimensional case, If B is an r-parent of C we say that C is the (left,right,central) r-child of B if C_r is the (left,right,central) child of B_r, and we say that B is the (left,right,central,non-left) r-parent of C if B_r is the (left,right,central,non-left) parent of C_r.

For a point x and box B we say that B is to the left of x if for each $j \in [d]$, B_j is to the left of x_j. We say that B is left-maximal for x if for each $j \in [d]$, B_j is left-maximal for x_j.

We say that a a box C is a non-left ancestor of B if for all $j \in [d]$, C_j is a non-left ancestor of B_j with respect to the one-dimensional dag D. Intuitively, a non-left ancestor of B is obtained by repeatedly taking a non-left parent along some direction. The number of non-left ancestors of B is $(\log n)^{O(d)}$, since any interval in $\mathcal{I}$ has at most $O(\log n)$ non-left ancestors.

4.5 The function *Build* : multi-dimensional case We now turn to the description of the procedure *Build*. We will make use of the multi-dimensional version of *Sift*.

Recall that *Build* takes as input a point x and returns a set $Rep(x)$ of points. Similar to the one-dimensional case, *Build* will use a pair of procedures *Sample* and $Refine$, which take as input a box $B \in \mathcal{B}$ and returns a small subset of points of B. The procedure $Refine$ only returns points that are accepted by *Sift*.

For a segment S of size at least 4, define $upper(S)$ to be the set consisting of the largest $|S|/4$ points in S.

$Sample(B)$ is defined as follows. Remember $t = cd\log^2 n$, for sufficiently large constant c. If $|B| \leq t$, $Sample(B) = B$. If $|B| > t$, $Sample(B)$ is defined to be the product set $Sample(B) = \prod_{i=1}^{d} Sample_i(B)$ where $Sample_i(B)$ is the set of points of the form $r^j(i)$ mod B_i where r^j ranges over points in the sequence $r^1, \ldots, r^t$. We also define the set $Lines(B)$ consisting of all lines L (in one of the d coordinate directions) such that $|L \cap B| \geq 4$ and $L \cap Sample(B) \neq \emptyset$.

Next we define $Refine$. On input B, the algorithm considers all non-left ancestors C of B in decreasing order of size. For each such C, the algorithm constructs $Refine(C) \subseteq Sample(C)$, and a set of lines $BadLines(C) \subseteq Lines(C)$.

$Refine(C)$ is defined to be the set of $x \in Sample(C)$ that are (i) accepted by *Sift*, and (ii) not contained in any line L that is in $BadLines(P)$ for some non-left parent P of C.

$BadLines(C)$ is the set of those lines $L \in Lines(C)$ of size at least 4 such that $upper(L \cap C) \cap Refine(C)$ is empty. Note that $BadLines(C)$ contains $BadLines(P)$, for any non-left parent of C. Let us again explain how $Refine(B)$ is determined. Let C be the largest non-left ancestor of B. The set $Refine(C)$ can be easily constructed by running *Sift* on every point in $Sample(C)$. This is used to determine $BadLines(C)$. Now consider the child C' of C which is a non-left ancestor of B. We throw away any point of $Sample(C')$ that lies in a line of $BadLines(C)$ or is rejected by *Sift*. After finding $BadLines(C')$, we move to the relevant child of C' and so on until we reach B.

We can now define the procedure *Build*. On input x, the output $Rep(x)$ is defined to be the union of $Refine(B)$ over all boxes that are left-maximal for x.

We now verify properties [B1]-[B5] of Section 2. Properties [B1] and [B2], that all points in $Rep(x)$ are less than or equal to x, and that each point in $Rep(x)$ is accepted by *Sift*, are immediate from the definition of $Rep(x)$.

To prove [B3], let x, y be arbitrary points in $[n]$ with $x \leq y$ and let $z \in Rep(x)$. We must show that there is

a $z' \in Rep(y)$ with $z \leq z'$. Since $z \in Rep(x)$, there is an interval I that is left-maximal for x such that $z \in Refine(I)$. If I is left-maximal for y then we take $z' = z$. Otherwise, let J be the largest interval in $path(I)$ that is to the left of y ; by definition J is left maximal for y. Since $Refine(I) \neq \emptyset$, $Refine(J) \cap upper(J) \neq \emptyset$ so we can select $z' \in Refine(J) \cap upper(J)$. Since J is a non-left ancestor of I, every point in I is less than every point in $upper(J)$ (this follows by an easy induction on $|J|$), so $z \leq z'$.

To prove [B3], let x, y be arbitrary points in $[n]^d$ with $x \leq y$ and let $z \in Rep(x)$. We must show that there is a $z' \in Rep(y)$ with $z \leq z'$. It suffices to consider the case that x and y differ only in one coordinate, say coordinate j, since the general case will then follow by an easy induction on the number of coordinates in which x and y differ.

Since $z \in Rep(x)$, there is a box B with $z \in Refine(B)$, such that B is a left-maximal for x. Define the box C such that $C_i = B_i$ for $i \neq j$ and C_j is equal to the largest non-left ancestor of B_j that is left of y_j. It follows that C is left-maximal for y.

Let L be the j-line through z. We claim that $upper(L \cap C) \cap Refine(C) \neq \emptyset$; if so we can select z' to be any element of $upper(L \cap C) \cap Refined(C)$ and then $z \leq z'$ and $z' \in Rep(y)$. Suppose for contradiction, $upper(L \cap C) \cap Refine(C) = \emptyset$, then by definition L belongs to $BadLines(C)$. A simple induction shows that for every j-descendant C' of C, $L \cap Refine(C') = \emptyset$, but this contradicts that $z \in B$ (since B is descendant of C).

Finally we come to the proof of [B4]. This is the hardest part and we break it into a separate subsection.

4.6 Proof that *Build* satisfies [B4] We want to get an upper bound on the number of points x for which $x \notin Rep(x)$ that holds with high probability. We assume throughout this analysis that the random strings $s^1, \ldots, s^d$ used for *Sift* are fixed in such a way that *Sift* succeeds (in the sense defined earlier.)

Let $A(x)$ be the box $[x_1 - 1, x_1] \times [x_2 - 1, x_2] \times \cdots [x_d - 1, x_d]$. $A(x)$ is left-maximal for x and so $Refine(A(x)) \subseteq Rep(x)$. So it suffices to prove an upper bound on the number of x for which $x \notin Refine(A(x))$.

For each box C, we define $Lines'(C)$, $Refine'(C)$ and $BadLines'(C)$ in a way that parallels $Lines(C)$, $Refine(C)$ and $BadLines(C)$, the key difference being that we look over all points in C not just those in $Sample(C)$. $Lines'(C)$ is the set of all lines L such that $|L \cap C| \geq 4$. $Refine'(C)$ is defined to be the set of $x \in C$ that are (i) accepted by *Sift*, and (ii) not contained in any line L that is in $BadLines'(P)$ for some non-left parent P of C. $BadLines'(C)$ is the set of those lines $L \in Lines'(C)$ of size at least 4 such that $|L \cap (C - Refine'(C))| \geq |L \cap C|/8$.

Observe that, with the random strings $s^1, \ldots, s^d$ being fixed, the functions $Refine'$, $Lines'$ and $BadLines'(P)$ are deterministic.

We say that *Build* succeeds if for all boxes B, $BadLines(B) \subseteq BadLines'(B)$.

LEMMA 4.2. *The probability that* Build *fails is at most* $n^{O(d)}2^{-\Omega(t)}$.

The proof is given in the full version of the paper.

PROPOSITION 4.1. *If* Build *succeeds then for each point* x, $Refine'(A(x)) \subseteq Refine(A(x))$.

Proof. $Sample(A(x)) = A(x)$, and therefore $y \notin Refine(A(x))$ means that y is rejected by *Sift* or y belongs to $BadLines(P)$ for some non-left parent of $A(x)$. Since *Build* succeeds, $BadLines(P) \subseteq BadLines'(P)$ and so $y \notin Refine'(A(x))$. □

So now it remains to find an upper bound on the size of $\cup_x (A(x) - Refine(A(x))$.

In Section 3.1, we defined $\Lambda_\theta(S)$ to be the union of all intervals $I \in \mathcal{I}$ for which $|S \cap I|/|I| \geq \theta$. We adapt this definition to the multidimensional setting. For $j \in [d]$, let $\Lambda^j_\theta(S)$ be the union over all j-lines L, of the union of all $I \in \mathcal{I}(L)$, for which $|S \cap I|/|I| \geq \theta$.

Let $\Upsilon_\theta(S) = \bigcup_{j \in [d]} \Lambda^j_\theta(S)$, and $\Upsilon^k_\theta(S)$ is defined by $\Upsilon^1_\theta(S) = \Upsilon_\theta(S)$ and $\Upsilon^k_\theta(S) = \Upsilon(\Upsilon^{k-1}_\theta(S))$.

We will show:

LEMMA 4.3. $\bigcup_x A(x) - Refine'(x) \subseteq \Upsilon^d_\theta(S)$ *where* S *is the set of points rejected by* Sift *and* $\theta = 2^{-(d+2)}$.

Assuming the lemma, we finish the proof of property [B4]. By Lemma 3.1, $\Lambda^j_\theta(S) \leq (2/\theta)|S|$, and therefore $\Upsilon_\theta(S) \leq (2d/\theta)|S|$. Thus $\Upsilon^d_\theta(S) \leq (2d/\theta)^d|S|$ which for the given value of θ is $2^{O(d^2)}$ times the number of points rejected by *Sift*.

So it remains to prove Lemma 4.3.

Proof of Lemma 4.3. We need some additional definitions. A cylinder is a box that for each direction j, is either degenerate or full. We say that C is a J-cylinder if J is the set of full directions.

For $j \in [d]$, an $([n] - \{j\})$-cylinder is also called a *j-hyperplane.* For $r \in [n]$ we write $H_j(r)$ for the j-hyperplane consisting of all points x with $x_j = r$.

Note that there is a natural bijection between a J-cylinder C and the set $[n]^{|J|}$. Using this bijection, we can define a family of boxes $\mathcal{B}(C)$ contained in C and a digraph $\Delta(C)$. Thus each box $B \in \mathcal{B}(C)$ has the form

$B_1 \times \cdots \times B_d$ where $B_j = C_j$ is a singleton for $j \notin J$ and $B_j \in \mathcal{I}$ for $j \in J$. We write $Cyl(B)$ for the unique cylinder C such that $B \in \mathcal{B}(C)$, this is also the unique smallest cylinder containing B. We call C the *cylinder-type* of B. The *dimensionality* of a box B, $dim(B)$ is the number of non-degenerate directions.

We will now define a set $BadBoxes$ of boxes. The set $BadBoxes$ is defined as follows. Consider boxes in increasing order of dimensionality. In other words, take boxes B and B' such that $Cyl(B)$ is a J-cylinder and $Cyl(B')$ is a J'-cylinder. If $|J| < |J'|$, then B is considered before B'. For boxes of the same dimensionality, we consider them in decreasing order of size. Initially we consider 0-dimensional boxes (points) and declare such a box to be bad if and only if the point is rejected by *Sift*. For a box B of dimension at least 1, we put $B \in BadBoxes$ if (1) There is a box B' of the same cylinder type as B that contains B and is in $BadBoxes$. (Note that B' is considered before B.) or (2) There is a non-degenerate direction j of B such that for at least $1/2^{d+2}$ of the values $r \in B_j$, $H_j(r) \cap B \in BadBoxes$. (Note that each box $H_j(r) \cap B$ has lower dimension than B and is considered before B).

The lemma will follow from:

P1 For any full-dimensional box $B \in \mathcal{B}$, if $x \in B - Refine'(B)$ there is a cylinder C containing x such that that $B \cap C \in BadBoxes$.

P2 $\bigcup_{B \in BadBoxes} B \subseteq \Upsilon^d_\theta(S)$

Proof of [P1] : We prove [P1] by reverse induction on $|B|$. If $B = [n]^d$, then $x \in B - Refine'(B)$ means that x is rejected by *Sift*, in which case we take C to be the singleton cylinder $\{x\}$. By definition, $\{x\} = B \cap C$ belongs to $BadBoxes$.

Suppose $|B| < n^d$. If x is rejected by *Sift* we again take $C = \{x\}$. Otherwise, since $x \in B - Refine'(B)$, there is a parent P of B and line $L \in BadLines'(P)$ with $x \in P \cap L$. Let j be the direction of L. Let $S = L \cap (P - Refined'(P))$. Since $L \in BadLines'(P)$, $|S| \geq |L \cap P|/8$. By induction, for each $y \in S$ there is a cylinder $C(y)$ such that $C(y) \cap P \in BadBoxes$. Let $J(y)$ be the set of directions for which $C(y)$ is non-degenerate. If there exists a $y \in S$ such that $j \in J(y)$, then $C(y) \cap B$ is also in $BadBoxes$ and $x \in C(y) \cap B$ as required. So assume $j \notin J(y)$ for all $y \in S$. For $J \subseteq [d] - \{j\}$, let $S(J) = \{y \in S : J(y) = J\}$. By a simple averaging argument, there must be a $J^* \subseteq [d] - \{j\}$ such that $|S(J^*)| \geq |S|/2^{d-1} \geq |L \cap P|/2^{d+2}$. Extending $C(y)$ in direction j gives the same cylinder C for all $y \in S(J^*)$. We then have $P \cap C \in BadBoxes$ since for $1/2^{d+2}$ of $r \in P_j$, $P \cap C \cap H_j(r) \in BadBoxes$. But then $B \cap C$ has the same cylinder type as $P \cap C$ and so is also in $BadBoxes$. This completes the proof of [P1].

Proof of [P2] : We claim that for all $B \in BadBoxes$, we have $B \subseteq \Upsilon^j_\theta(S)$ where $j = dim(B)$. The proof is a straightforward induction on $j = dim(B)$, and is given in the full version of the paper.

References

[1] Ailon, N., Chazelle, B. *Information Theory in Property Testing and Monotonicity Testing in Higher Dimension*, Proc. 22nd STACS, 2005, 434–447.

[2] Ailon, N., Chazelle, B., Comandur, S., Liu, D. *Estimating the distance to a monotone Function*, Proc. 8th RANDOM, 2004, 229–236.

[3] Ailon, N., Chazelle, B., Comandur, S., Liu, D. *Property Preserving Data Reconstruction*, Proc. ISAAC (2004).

[4] Blum, M., Luby, M., Rubinfeld, R. *Self-testing/correcting with applications to numerical problems*, Proc. 22nd STOC, 1990, 73–83.

[5] Dodis, Y., Goldreich, O., Lehman, E., Raskhodnikova, S., Ron, D., Samorodnitsky, A., *Improved Testing Algorithms for Monotonicity* Proc. 2nd Random, 1999, 97–108.

[6] Ergun, F., Kannan, S., Kumar, S. Ravi, Rubinfeld, R., Viswanathan, M. *Spot-checkers*, Proc. 30th STOC, 1998, 259–268.

[7] Fischer, E. *The art of uninformed decisions: A primer to property testing*, Bulletin of EATCS (75), 2001, 97–126.

[8] Goldreich, O., *Combinatorial property testing - A survey*, Randomization Methods in Algorithm Design, 1998, 45–60.

[9] Goldreich, O., Goldwasser, S., Lehman E., Ron, D., Samorodnitsky, A., *Testing Monotonicity*, Combinatorica (20), 2000, 301–337.

[10] Goldreich, O., Goldwasser, S., Ron, D. *Property testing and its connection to learning and approximation*, Journal of the ACM (45), 1998, 653–750.

[11] Halevy, S., Kushilevitz, E. *Distribution-Free Property Testing*, RANDOM-APPROX, 2003, 302–317.

[12] Parnas, M., Ron, D., Rubinfeld, R. *Tolerant property testing and distance approximation*, ECCC TR04-010, 2004

[13] Ron, D. *Property testing*, Handbook on Randomization, Volume II, 2001, 597-649.

[14] Rubinfeld, R., Sudan, M. *Robust characterization of polynomials with applications to program testing*, SIAM Journal of Computing (25), 1996, 647–668.