Property-Preserving Data Reconstruction *

Nir Ailon Bernard Chazelle Seshadhri Comandur Ding Liu

Abstract

We initiate a new line of investigation into *online property-preserving data reconstruction.* Consider a dataset which is assumed to satisfy various (known) structural properties; eg, it may consist of sorted numbers, or points on a manifold, or vectors in a polyhedral cone, or codewords from an error-correcting code. Because of noise and errors, however, an (unknown) fraction of the data is deemed *unsound*, ie, in violation with the expected structural properties. Can one still query into the dataset in an online fashion and be provided data that is *always* sound? In other words, can one design a filter which, when given a query to any item I in the dataset, returns a *sound* item J that, although not necessarily in the dataset, differs from I as infrequently as possible. No preprocessing should be allowed and queries should be answered online.

We consider the case of a monotone function. Specifically, the dataset encodes a function $f : \{1, \ldots, n\} \mapsto \mathbf{R}$ that is at (unknown) distance ε from monotone, meaning that f can—and must—be modified at εn places to become monotone.

Our main result is a randomized filter that can answer any query in $O(\log^2 n \log\log n)$ time while modifying the function f at only $O(\varepsilon n)$ places. The amortized time over n function evaluations is $O(\log n)$. The filter works as stated with probability arbitrarily close to 1. We provide an alternative filter with $O(\log n)$ worst case query time and $O(\varepsilon n \log n)$ function modifications. For reconstructing d-dimensional monotone functions of the form $f : \{1, \ldots, n\}^d \mapsto \mathbf{R}$, we present a filter that takes $(2^{O(d)}(\log n)^{4d-2} \log\log n)$ time per query and modifies at most $O(\varepsilon n^d)$ function values (for constant d).

1 Introduction

It is a fact of (computing) life that massive datasets often come laden with varying degrees of reliability. Errors might be inherent to the data acquisition itself (faulty sensors, white/bursty noise, aliasing), or to data processing (roundoff errors, numerical instability, coding bugs), or even to intrinsic uncertainty (think of surveys and poll data). Classical error correction postulates the existence of exact data and uses redundancy to provide recovery mechanisms in the presence of errors. Mesh generation in computer graphics, on the other hand, will often deal with reconstruction mostly on the basis of esthetic criteria, while signal processing might filter out noise by relying on frequency domain models.

In the case of geometric datasets, reconstruction must sometimes seek to enforce structural properties. Early work on geometric robustness [9, 15] pointed out the importance of topological consistency. For example, one might want to ensure that the output of an imprecise, error-prone computation of a Voronoi diagram is *still* a Voronoi diagram (albeit that of a slightly perturbed set of points). Geometric algorithm design is notoriously sensitive to structure: dimensionality, convexity, and monotonicity are features that often impact the design and complexity of geometric algorithms. Consider a computation that requires that the input be a set of points in convex position. If the input is noisy, convexity might be violated and the algorithm might crash. Is there

*This work was supported in part by NSF grants CCR-998817, 0306283, ARO Grant DAAH04-96-1-0181.

a filter that can be inserted between the algorithm (the client) and the dataset so that: (i) the client is always provided with a point set in convex position; and (ii) the "filtered" data differs as little as possible from the original (noisy) data? In an offline setting, the filter can always go over the entire dataset, compute the "nearest" convex-position point set, and store it as its filtered dataset. This is unrealistic in the presence of massive input size, however, and only online solutions requiring no preprocessing at all can be considered viable. We call this *online property-preserving data reconstruction.* Besides convexity, other properties we might wish to preserve include low dimensionality and angular constraints:

- Consider a dataset consisting of points on a low-dim manifold embedded in very high dimensional space. Obviously, the slighest noise is enough to make the point set full-dimensional. How to "pull back" points to the (unknown) manifold online can be highly nonobvious.
- Angle constraints are of paramount importance in industrial/architectural design. Opposite walls of a building have a habit of being parallel, and no amount of noise and error should violate that property. Again, the design of a suitable filter to enforce such angular constraints online is an interesting open problem.

In this paper we consider one of the simplest possible instances of online property-preserving reconstruction: monotone functions. Sorted lists of numbers are a requirement for all sorts of operations. A binary search, for example, will easily err if the list is not perfectly sorted. In this case of property-preserving data reconstruction, the filter must be able to return a value that is consistent with a sorted list and differs from the original as little as possible. (An immediate application of such a filter is to provide robustness for binary searching in near-sorted lists.)

We formalize the problem. Let $f : \{1, \dots, n\} \mapsto \mathbf{R}$ be a function at an unknown distance ε from monotonicity, which means that f can (and must) be modified at εn places to become monotone. Figure 1 illustrates the filter in action. To avoid confusion, we use the term "query" to denote interaction between the client and the filter, and "lookup" to denote interaction between the filter and the dataset. Given a query x, the filter generates lookups $a, b, c, \dots$ to the dataset, from which it receives the values $f(a), f(b), f(c), \dots$, and then computes a value $g(x)$ such that the function g is monotone and differs from f in at most $k\varepsilon n$ places, for some k (typically constant, but not necessarily so). We note two things.

1. Once the filter outputs $g(x)$ for some query x, it commits to this value and must output the same value upon future queries.
2. The filter may choose to follow a multi-round protocol and adaptively generate lookups to the dataset depending on previous results. The function $g(x)$ is defined on the fly, and it can depend on both the queries and on random bits. Therefore, after the first few queries, g might only be defined on a small fraction of the domain. At any point in time, if k distinct x_i's have been queried so far, then querying the remaining x_i's (whether the client does it or not) while honoring past commitments leads to a monotone function close enough to f.

It is natural to measure the performance of the filter with respect to two functions. A $(p(n, \varepsilon), q(n))$-filter performs $O(p(n, \varepsilon))$ lookups per query, and returns a function g that is at a distance of at most $q(n)\varepsilon$ from monotonicity, with high probability. The lookup-per-query guarantee can be either amortized or in worst case (the running times are deterministic). Ideally, we would like $p(n, \varepsilon)$ to depend only on n, and $q(n)$ to be constant. There is a natural tradeoff between p and q: we expect q to decrease as p increases. We will see an example of this in this work.

Theorem 1.1 *For any fixed $\delta > 0$ there exists a randomized $(\log^2 n \log\log n, 2+\delta)$-filter with a worst case lookups-per-query guarantee. The amortized lookups-per-query over n function evaluations is $O(\log n)$. The filter behaves as stated with probability arbitrarily close to 1.*

When the filter fails, the failure is in the distance guarantee - the filter returns a monotone function which is much farther than what is guaranteed.

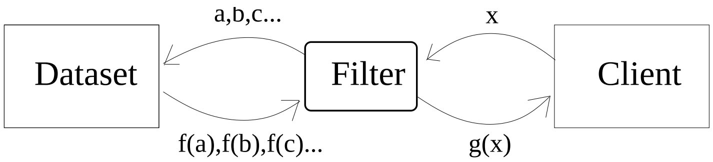

Figure 1: *The property-preserving reconstruction filter: g is sound and differs from f in few places.*

We also provide an alternative filter with a better lookups-per-query guarantee and a worse distance guarantee.

Theorem 1.2 *There exists a $(\log n, O(\log n))$-filter with a worst case lookups-per-query guarantee.*

It is important to note that, in this work, we think of the client as *adversarial*. That is, the filter's guarantees must hold for all sequences of client queries. However, in some cases it might be useful to assume the client's queries are drawn from some known probability distribution. We will see that the filter can take advantage of this.

Theorem 1.3 *Assuming the client draws the queries independently, uniformly at random, a $(1, O(\log n))$-filter can be devised.*

We also extend these techniques to construct filters for higher-dimensional monotonicity.

Theorem 1.4 *There exists a $(2^{O(d)}(\log n)^{4d-2} \log\log n, (2^d + d\delta))$-filter*[1] *for any fixed $\delta > 0$ for d-dimensional monotonicity.*

We are not aware of any previous work on this specific problem. There are many differences between this work and self-correction [4, 14]. Reconstruction deals with data, not just functions. Also, reconstruction is completely error-free but allows the data to be modified upto a constant factor of the distance. Halevy and Kushilevitz [10] define the notion of *property self-correction* (also implicitly used in [14]) which is similar except that their definition requires the existence of a *unique* closest function g which is a reconstruction of f. Their definition was useful in a different context and is not flexible enough to allow the lookups-per-query vs. distance tradeoff used in the definition of a filter.

One of the important features of this model is the importance of *early decisions*. Every time a query is handled, a new constraint is imposed on further query responses - every answer *has* to be consistent with previous answers.

In related areas, property testing is by now a well studied area [8, 14], with many nontrivial results regarding combinatorial, algebraic, and geometric problems [6, 7, 13]. Most notably, linearity testing and low-degree polynomial testing (and correcting) have been vastly studied in the context

[1]For this filter, we actually prove a bound on the running time per query, which happens to be the same as the lookup complexity.

of error-correcting codes, program checking and PCP's. Note that these algebraic problems deal with objects of succint description (coefficients of polynomials). Early wrong decisions of a filter in this case would result in an object g of very large distance from f. Therefore, a "smooth" lookups-per-query vs. distance tradeoff is not possible here. The properties we are interested in (eg monotonicity, convexity of functions) allow a smooth penalty for early wrong decisions as a function of the amount of risk (lookups-per-query) assumed by the filter.

More recent work [2, 11] has provided sublinear algorithms for estimating the distance of a function to monotone. We use ideas from [2] in this work.

2 The $(\log^2 n \log\log n, 2+\delta)$-filter

We use the following notation in what follows. The distance between two functions f_1 and f_2 over the domain $\{1, .., n\}$ is defined as the fractional size of domain points on which they disagree. The function f and the domain size n which are the input to the problem (the dataset in Figure 1) are fixed. We use ε to denote the distance of f from monotonicity, and $\hat{f}$ to denote the monotone function closest to f. So the distance between f and $\hat{f}$ is ε, and $\hat{f}$ minimizes the distance between f and any monotone function. We use g to denote the function output by the filter.

2.1 Preliminaries

Proving Theorem 1.1 requires a few preliminaries, beginning with these definitions:

- *δ-bad and δ-good* : Given $0 < \delta < 1/2$, the integer i is called *δ-bad* if there exists $j > i$ such that
$$\left|\left\{ i \le k \le j \,|\, f(k) < f(i) \right\}\right| \ge (1/2 - \delta)(j - i + 1)$$
or, similarly, $j < i$ such that
$$\left|\left\{ j \le k \le i \,|\, f(k) > f(i) \right\}\right| \ge (1/2 - \delta)(i - j + 1) \ .$$
Otherwise the integer i is called *δ-good*.

- *a-light and a-heavy* : Let $\mathcal{D}$ be the joint distribution of m independent 0/1 random variables $x_1, \ldots, x_m$, which can be sampled independently. If $\mathbf{E}[x_i] \le a$ for all i, then $\mathcal{D}$ is called *a-light*; else it is *a-heavy*.

Roughly speaking, if an integer i belongs to an interval containing many (a constant fraction) violations with i, then i is *bad*. These bad integers appear to the places where the function f needs to be modified.

Lemma 2.1 (Ailon et al. **[2]**) *Given any fixed $a < b$, if $\mathcal{D}$ is either a-light or b-heavy, then with probability 2/3 we can tell which is the case in $O(m)$ time. If $\mathcal{D}$ is neither, the algorithm outputs an arbitrary answer.*

For the sake of completeness, the algorithm of Lemma 2.1 is given in Figure 2. The test is run by calling *light-test*$(\mathcal{D}, a, b, c')$ with $c' = \Theta(b/(b-a)^2)$ (see [2]). Note that in each recursive call of *light-test*, the number of random variables in $\mathcal{D}$ decreases by a factor of 2 - therefore, the running time bound in 2.1 is deterministic. In the following we use the algorithm of Lemma 2.1 to test, with high probability of success, whether a given integer i is δ-bad or 2δ-good, for any fixed $\delta > 0$. Given an interval $[u, v]$, we define two 0/1 random variables $\alpha[u, v]$ and $\beta[u, v]$: given a random integer $j \in [u, v]$, $\alpha[u, v] = 1$ (resp. $\beta[u, v] = 1$) iff $f(u) > f(j)$ (resp. $f(j) > f(v)$). The algorithm *bad-good-test* (Fig. 3) tests if a given integer i is δ-bad or 2δ-good.

```
light-test(𝒟, a, b, k)

 for each x ∈ 𝒟
   sample it k times and compute the average x̂
 form 𝒟' = { x ∈ 𝒟 | x̂ > (a + b)/2 }
 if |𝒟'| = 0 then
   output "a-light"
 if |𝒟'| ≥ |𝒟|/2 then
   output "b-heavy"
 light-test(𝒟', a, b, k + c)
 /* c sufficiently large global constant */
```

Figure 2: **light-test**

Lemma 2.2 *Given any fixed $\delta > 0$ and a parameter k, if i is either δ-bad or 2δ-good, then* bad-good-test *will tell which is the case in time $O(\log n \log k)$ and with probability at least $1 - 1/k$.*

Proof: If integer i is 2δ-good then the expectation of every $x_{2j-1}^{(i)}$ or $x_{2j}^{(i)}$ defined in *bad-good-test* is at most $1/2 - 2\delta$, and so the distribution $\mathcal{D}$ is $(1/2 - 2\delta)$-light. On the other hand, if i is δ-bad, then there exists some $x_{2j-1}^{(i)}$ or $x_{2j}^{(i)}$ with expectation at least $(1/2 - \delta)/(1 + \delta) \geq 1/2 - 3\delta/2$, and so $\mathcal{D}$ is $(1/2 - 3\delta/2)$-heavy. The algorithm from Lemma 2.1 distinguishes between $(1/2 - 2\delta)$-light and $(1/2 - 3\delta/2)$-heavy with probability $2/3$ in $O(\log n)$ time. Since we repeat it $c \log k$ times and take a majority vote, a standard Chernoff bound argument shows that *bad-good-test* fails with probability at most $1/k$. □

Lemma 2.3 *Let $\delta > 0$ be fixed, and ε be the distance of f from monotonicity. There are at most $(2 + O(\delta))\varepsilon n$ δ-bad integers (*Ailon et al. **[2]***). Moreover, the monotone function $\hat{f}$ which is closest to f can be assumed to agree with f on δ-good integers.*

Finally, we let $\delta > 0$ denote an arbitrarily small positive real. Choosing a small enough δ will satisfy the distance guarantee of Theorem 1.1.

2.2 The Algorithm

We now describe the algorithm *monotonize*. Our goal, as described above, is: given a fixed $\delta > 0$, compute a function g online such that: (1) g is monotone; (2) g is $((2+O(\delta))\varepsilon)$-close to f. Specifically, on query i, *monotonize* computes $g(i)$ in time $O(\log^2 n \log \log n)$. Whenever *monotonize* outputs a value $g(i)$, this value must be recorded to ensure consistency. The procedure will therefore hold an internal data structure that will record past commitments. The data structure can be designed to allow efficient retrievals, but we omit the details because we are mainly interested in the number of f-lookups it performs, and not the cost of other operations.

Given a query i, *monotonize* first checks whether i was committed to in the past, and outputs that commitment in that case. If not, more work should be done. In virtue of Lemma 2.3, *monotonize* tries to keep the f values at δ-good integers and change the values for other queries. We will use *bad-good-test* to decide whether i is bad or good.

```
bad-good-test (f, i, δ, k)

  repeat the following c log k times for a big enough constant c
      for each 1 ≤ j ≤ (2/δ) ln n
          define x^{(i)}_{2j-1} = α[i, i + (1 + δ)^j] and x^{(i)}_{2j} = β[i − (1 + δ)^j, i]
      let D be the distribution (x_1, x_2, ···)
      run light-test(D, 1/2 − 2δ, 1/2 − 3δ/2, c'/δ^2) and record output
                /* c' sufficiently large global constant */
           /* This call tells whether D is (1/2 − 2δ)-light or (1/2 − 3δ/2)-heavy.*/
  if majority of light-test outputs are "(1/2 − 2δ)-light"
      then output 2δ-good
      else output δ-bad
```

Figure 3: Testing if an integer i is δ-bad or 2δ-good.

```
find-good-value (f, I, δ)

  set L as an empty list
  randomly select cδ^{-1} log n integers from I for a big enough constant c.
  for each j in random sample
      if bad-good-test (f, j, δ, δ^{-2}) outputs ''2δ-good''
          then append f(j) to L
  output median value of L
```

Figure 4: Finding a good value in an interval.

Suppose now that we decide that i is δ-bad and hence $g(i)$ needs a value that might be different from $f(i)$. Ideally, we would like to find the closest δ-good integers l (to the left of i) and r (to the right of i) and assign $g(i)$ to some value between $f(l)$ and $f(r)$. Because of the sublinear time constraint, we slightly relax this condition. Instead, the idea is to find an interval I_0 around i such that the fraction of 2δ-good integers in I_0 is at least $\Omega(\delta)$, but their fraction in a slightly smaller interval is $O(\delta)$. This ensures that such an interval can be detected through random sampling and that there are not many 2δ-good integers between i and any 2δ-good integer in this interval (a relaxation of the closest condition).

We will search for a good interval within the interval determined by the closest committed values on the left and right of i. Denote this interval by $[l, r]$. Once such a good interval I_0 is found, we try to find a value x that is sandwiched between values of f evaluated at two δ-good points in I_0. Finding x is done in *find-good-value* (Figure 4). We commit to the value x on g restricted to I_0. If no good intervals are found, we spread the value of $g(l)$ on g in the interval $[l + 1, r - 1]$.

Lemma 2.4 *Assume that I contains at least a fraction of δ 2δ-good integers. Then, the procedure* find-good-value *outputs, with probability $1 - 1/n^4$, a value y that is sandwiched between $f(i_1)$ and $f(i_2)$, where $i_1, i_2 \in I$ are δ-good. The running time of* find-good-value *is $O(\log^2 n)$, for fixed δ.*

Proof: The expected number X of δ-bad samples for which *bad-good-test* outputs "2δ-good" is at most $c\delta(1-\delta)\log n$, by Lemma 2.2. The expected total number Y of samples for which *bad-good-test* outputs "2δ-good" is at least $c(1-\delta^2)\log n$. The probability that X exceeds $Y/2$ is at most $1/n^4$ if c is chosen large enough, using Chernoff bounds. Therefore, with probability at least $1-1/n^4$, more than half the values that are appended to the list L are f-values on δ-good points ("good values"). By taking the median of values in L, in such a case, we are guaranteed to get a value sandwiched between two good values. The time bound follows from Lemma 2.2. $\square$

monotonize (f, δ, i)

```
  if g(i) was already committed to then output g(i)
  if bad-good-test(f, i, δ, n^3) outputs ''2δ-good''
→     then commit to g(i) = f(i) and output g(i)

  let l be closest committed index on left of i (0 if none)
  let r be closest committed index on right of i (n+1 if none)
  (*)
  set j_max = ⌊ln(i − l)/ln(1 + δ)⌋ and j_min = 0.
  while j_max − j_min > 1
      set j = ⌊(j_max + j_min)/2⌋, I = [i − (1 + δ)^j, i]
      choose random sample of size cδ^{-1} log n from I, for large c
      for each point i' in sample
          run bad-good-test(f, i', δ, c_1), for large c_1
      if number of "2δ-good" outputs is ≥ (3/2) c log n
          then set j_max = j
          else set j_min = j
  if j_max ≠ ⌊ln(i − l)/ln(1 + δ)⌋
      then set I_l = [i − (1 + δ)^{j_max}, i] and val_l = find-good-value(f, I_l, δ)
      else set val_l = g(l) and I_l = [l + 1, i]
  (**)
  repeat lines (*)...(**) for right side of i, obtaining val_r and I_r

  choose val_l < y < val_r and commit to y on I_l ∪ {i} ∪ I_r
  output y
```

Figure 5: Computing a monotone function online.

To find a good interval, we do a binary search among all the intervals of length $(1+\delta)^j$ ($j = 0, 1, \ldots$) starting or ending at i, that is, $[i, i + (1+\delta)^j]$ and $[i - (1+\delta)^j, i]$. There are $O(\log n)$ such intervals, and thus the running time is $O(\log\log n)$ times the time spent for each interval. The overall algorithm *monotonize* is shown in Figure 5. The following claim together with a suitable rescaling of δ concludes the proof of the first part of Theorem 1.1.

Claim 2.5 *Given any* $0 < \delta < \frac{1}{2}$*, with probability* $1 - 1/n$*,* monotonize *computes a monotone function* g *that is within distance* $(2+\delta)\varepsilon$ *to* f*. Given a query* i*,* $g(i)$ *is computed online in time* $O(\log^2 n \log\log n)$*, when* δ *is assumed to be fixed.*

Proof: First we analyze the running time. The *bad-good-test* in line 3 takes $O(\log^2 n)$ time. If the algorithm determines that i is δ-bad, then the while-loops run $O(\log\log n)$ times. In one iteration of the while-loop, the algorithm calls *bad-good-test* $O(\log n)$ times. Each call takes $O(\log n)$ time by Lemma 2.2. Therefore, the time complexity of the while-loop is $O(\log^2 n \log\log n)$. By Lemma 2.4, the running time of the call to *find-good-value* is $O(\log^2 n)$. The time complexity of the algorithm is therefore $O(\log^2 n \log\log n)$.

Let us first look at the while-loop. If I has more than 2δ-fraction of 2δ-good integers, then the number of "2δ-good" outputs is $< \frac{3}{2}c\log n$ with inverse polynomial probability. This can be shown through Chernoff bounds. On the other hand, if I has less than δ-fraction of 2δ-good integers, then the number of "2δ-good" outputs is $> \frac{3}{2}c\log n$ with inverse polynomial probability. Consider the events -

- The intervals I_l and I_r have at least a δ-fraction of 2δ-good integers and the call to *find-good-value* succeeds.
- The interval $I_{min} = [i - (1+\delta)^{j_{min}}, i]$ has at most 2δ-fraction of 2δ-good integers.

Both these events hold with probability $> 1 - 1/n^4$. The intervals I_l, I_r, and I_{min} are constructed at most $O(n^2)$ times (over all queries). Now consider the event that the call to *bad-good-test* (in line 3) correctly distinguishes between δ-bad and 2δ-good integers. As shown in Lemma 2.2, this happens with probability $> 1 - 1/n^3$. This is totally called at most n times. By a union-bound, all of the above events occur (for every query) with probability $> 1 - 1/n^d$, for some positive constant d. Therefore, we henceforth assume that these events always occur (in other words, the probability of something "bad" happening is polynomially small).

To show that the function g is monotone, we first note that if *bad-good-test* outputs "2δ-good" for i (leading to $g(i)$ being set to $f(i)$), then (by our assumption made above) i is not δ-bad. If i is δ-bad, then val_l lies between the value at two δ-good points in I_l. If val_l is assigned as the g-value of all points in I_l, then g would be monotone with respect to all the values at the δ-good points already committed to. Similarly, val_r can be assigned as the g-value of all points in I_r without disturbing monotonicity. Therefore, since the algorithm assigns some value between val_l and val_r to $I_l \cup \{i\} \cup I_r$, g remains monotone.

Finally we show that g is within distance $(2+\delta)\varepsilon$ to f. We can assume that for $I_{min} = [i - (1+\delta)^{j_{min}}, i]$, the fraction of 2δ-good integers in I_{min} is at most 2δ. Since by the end of the algorithm $j_{max} \leq j_{min} + 1$, the fraction of 2δ-good integers in $I_r = [i, i + (1+\delta)^{j_{max}}]$ (or $I_l = [i - (1+\delta)^{j_{max}}, i]$) is at most 4δ. In other words, each time we make a total of $|I_l \cup \{i\} \cup I_r|$ corrections to f at least a $(1-4\delta)$-fraction of these changes are made on 2δ-bad integers. By Lemma 2.4 in [2], the total number of 2δ-bad integers is at most $(2+10\delta)\varepsilon n$. So the total number of changes we made on f is at most $(2+10\delta)\varepsilon n/(1-4\delta) \leq (2+c\delta)\varepsilon n$ for some constant c. This concludes the proof. □

2.3 Achieving Logarithmic Amortized Query Time

In this section we show how to modify the algorithm to achieve better amortized query time. The worst case query time for a single query remains the same. We need a technical lemma first.

Lemma 2.6 *For any $1/2 > \delta > 0$, let i be a δ-bad integer. Let l, r be two δ-good integers such that $l < i < r$. Then there is a witness to i's badness in the interval $[l, r]$.*

Proof: If $f(i) < f(l)$, then we claim that l is a witness to i's badness. In fact, since $f(l)$ and $f(i)$ is a violating pair, it is immediate that at least one of them is 0-bad with respect to the interval $[l, i]$. Since l is δ-good, i must be 0-bad (and hence δ-bad) with respect to $[l, i]$. In this case, l is a witness to i's badness. Similarly, r will be a witness if $f(i) > f(r)$. In the following we assume that $f(l) < f(i) < f(r)$.

Let w be a witness to i's badness. Without loss of generality, assume that $w < i$. If $w \geq l$ then we are done, so let $w < l$. Since i is δ-bad and l is δ-good, we know that: number of violations in $[w, l]$ with respect to l is $< (1/2 - \delta)(l - w + 1)$; number of violations in $[w, i]$ with respect to i is $\geq (1/2 - \delta)(i - w + 1)$. We also know that each violation in $[w, l]$ with respect to i is also a violation with respect to l, so the number of violations with respect to i in $[l + 1, i]$ is more than $(1/2 - \delta)(i - l) = (1/2 - \delta)(i - (l + 1) + 1)$. This shows that i has a witness to its badness in $[l + 1, i]$. □

The improvement on amortized query time comes from the following strategy: each time the algorithm answers a client query, it also generates a new query by itself and answers that query. This self query is completely independent of all the client queries, and we call it an *oblivious* query.

The oblivious queries are generated based on the balanced binary tree on $[1, n]$. The root of this tree is $\lfloor n/2 \rfloor$. The left subtree of the root corresponds to the interval $[1, \lfloor n/2 \rfloor - 1]$, and similarly the right subtree corresponds to $[\lfloor n/2 \rfloor + 1, n]$. The two subtrees are then defined recursively. This tree is denoted by T.

The oblivious queries are generated according to the following order. We start from the root of T and scan its elements one by one by going down level by level. Within each level we scan from left to right. This defines an ordering of all integers in $[1, n]$ which is the order to make oblivious queries. This ordering ensures that, after the $(2^k - 1)$-th oblivious query, $[1, n]$ is divided by all the oblivious queries into a set of disjoint intervals of length at most $n/2^k$. Each oblivious query is either a δ-good integer itself in which case *monotonize* returns at the line marked by $\rightarrow$, or it causes two δ-good integers being outputted (val_l and val_r in *monotonize*). These two δ-good integers lie on the left and right side of the oblivious query, respectively. This shows that after the $(2^k - 1)$-th oblivious query, $[1, n]$ is divided by some 2δ-good integers into a set of smaller intervals each of length at most $n/2^k$.

Based on Lemma 2.6, whenever we call *bad-good-test* (in *find-good-value* or *monotonize*) to test the badness of an integer i, we only need to search for a witness within a smaller interval $[l, r]$ such that l (resp. r) is the closest δ-good integer on the left (resp. right) of i. As explained above, these δ-good integers come as by-products of oblivious queries. This will reduce the running time of *bad-good-test* to $O(\log n_i \log k)$ (to achieve success probability at least $1 - 1/k$), where $n_i = r - l + 1$. Accordingly, the time spent on binary searching intervals in *monotonize* is reduced to $O(\log \log n_i)$. By the distribution of oblivious queries, for the j-th client query where $2^{k-1} \leq j < 2^k$, the running time of *monotonize* is now $O(\log n \log \frac{n}{2^k} \log \log \frac{n}{2^k})$. The same is true for the j-th oblivious query.

To bound the amortized running time, it suffices to focus on the smallest m such that all n distinct queries appear in the first m queries (including both client and oblivious queries). We can also ignore repetition queries (those that have appeared before) since each one only takes $O(\log n)$ time using standard data structure techniques. Therefore, without loss of generality, we assume that the first n client queries are distinct. The total query time for these n queries is:

$$\sum_{k=1}^{\log n} O(2^{k-1} \log n \log \frac{n}{2^k} \log \log \frac{n}{2^k}) \ .$$

```
build-tree(a, b)

 if a > b then
     output empty tree
 else
     output tree with
       root i chosen uniformly at random in [a, b]
       left subtree build-tree(a, i − 1)
       right subtree build-tree(i + 1, b)
```

It is simple to verify that this sum is $O(n \log n)$. The following claim concludes the proof of the second part of Theorem 1.1.

Claim 2.7 *With probability* $1-1/n$*,* monotonize *computes a monotone function* g *that is within distance* $(2+O(\delta))\varepsilon$ *to* f*. Each single evaluation of* $g(i)$ *is computed online in time* $O(\log^2 n \log\log n)$*. In addition,* monotonize *can be modified slightly to ensure that the amortized query time over the first* $m \geq n$ *client queries is* $O(\log n)$*.*

3 The $(\log n, O(\log n))$- filter

We prove Theorem 1.2. To do this, we define a function g by a random process. The function is determined after some coin flipping done by the algorithm (before handling the client queries). Although the function g is defined after the coin flips, the algorithm doesn't explicitly know it. In order to explicitly calculate g at a point, the algorithm will have to do some f-lookups. Our construction and analysis will upper bound $\mathbf{E}[\mathrm{dist}(f,g)]$ and the amount of work required for explicitly calculating g at a point.

As before, let $\hat{f}$ be a monotone function such that $\mathrm{dist}(f,\hat{f}) = \varepsilon$. Let $B \subseteq [n]$ be the set of points $\{x | f(x) \neq \hat{f}(x)\}$. So $|B| = \varepsilon n$. For simplicity of notation, assume the formal values of $-\infty$ (resp. $+\infty$) of any function on $[n]$ evaluated at 0 (resp. $n+1$).

We build a randomized binary tree $T = \text{build-tree}(1,n)$ with nodes labeled $1..n$, where build-tree(a,b) is defined as follows - after constructing the randomized tree T, the function g at point i is defined as follows. If i is the root of the tree, then $g(i) = f(i)$. Otherwise, Let $p_1, .., p_j, i$ denote the labels of the nodes on the path from the root to node i, where p_1 is the root of the tree and p_j is the parent of i. Assume that g was already defined on $p_1, .., p_j$. Let $l = \max(\{0\} \cup \{p_k | p_k < i\})$ and $r = \min(\{n+1\} \cup \{p_k | p_k > i\})$. If $g(l) \leq f(i) \leq g(r)$, then define $g(i) = f(i)$, otherwise define $g(i)$ as an arbitrary value in $[g(l), g(r)]$. The function g is clearly monotone. The number of f-lookups required for computing $g(i)$ is the length of the path from the root to i. A proof of the following well-known fact can be found in, e.g. [12].

Fact 3.1 *The expected height of* T *is* $O(\log n)$*.*

We show that $\mathbf{E}[\mathrm{dist}(f,g)] = O(\varepsilon \log n)$. We first observe that for any i, if $\{p_1, .., p_j, i\} \bigcap B = \emptyset$, then it is guaranteed that $g(i) = f(i)$. Therefore, any i for which $f(i) \neq g(i)$ can be charged to some $b \in B$ on the path from the root to i. The amount of charge on any $b \in B$ is at most the size of the subtree b in T.

Lemma 3.2 *The expected size of the subtree rooted at node i in T is $O(\log n)$ for any $i \in [n]$.*

Proof: For each $j \in [n]$ such that $j > i$, it is clear that j will be in the subtree rooted by i if and only if i is inserted into the tree before all of $i+1, i+2, \ldots, j$. This happens with probability exactly $1/(j-i+1)$. Similarly, $j < i$ is in the subtree rooted by i with probability exactly $1/(i-j+1)$. We conclude that the expected number of elements in the subtree rooted by i is at most $2\sum_{k=1}^{n} 1/k = O(\log n)$, as required. $\square$

Therefore, the expected total amount of charge is at most $O(|B|\log n) = O(n\varepsilon \log n)$. By Markov's inequality, the total amount of charge is at most $O(n\varepsilon \log n)$ with high probability. The total amount of charge is an upper bound on the distance between f and g. This proves Theorem 1.2, except for the fact that the lookups-per-query guarantee is only on expectation, and not worse case (due to Fact 3.1). However, there is an alternative way to construct T so that we get a worse-case guarantee. We describe the construction and sketch the elementary proof. Assume for ease of notation that $f(i)$ is defined as $-\infty$ for $i < 1$ and as ∞ for $i > n$. Choose the label p of the root uniformly at random in $[n]$. Then set the labels of its left and right children as $\lfloor p - n/2 \rfloor$ and $\lfloor p + n/2 \rfloor$, respectively. Set the labels of the next level (from left to right) as $\lfloor p - 3n/4 \rfloor$, $\lfloor p - n/4 \rfloor$, $\lfloor p + n/4 \rfloor$ and $\lfloor p + 3n/4 \rfloor$, pruning labels that had already been used. Continue until all integers in $[n]$ are a label in the tree. Clearly the height of the tree is $O(\log n)$. The expected size of a subtree rooted at a node of a fixed label is $O(\log n)$ (it is easy to see that this expectation is proportional to the largest divisor of the form 2^t of a random number in $[n]$). This gives a worst case (instead of expected) guarantee of $O(\log n)$ on the length of the path from the root to i (and hence on the number of f-lookups per client query). This concludes the proof of Theorem 1.2.

To prove Theorem 1.3, where the client queries are assumed to be uniformly and independently chosen in $[n]$, we observe that the choices the client makes can be used to build T. More precisely, we can build T on the fly, as follows: The root r of T is the first client query. The left child of r is the first client query in the interval $[1, r-1]$, and the right child of r is the first client query in the interval $[r+1, n]$. In general, the root of any subtree in T is the first client query in the interval corresponding to that subtree. Clearly, this results in a tree T drawn from the same probability distribution as in build-tree$(1, n)$. So we still have Lemma 3.2, guaranteeing the upper bound on the expected distance between g and f. But now we observe that for any new client query i, the path from the root of T to i (excluding i) was already queried, so we need only one more f-lookup, namely $f(i)$. This concludes the proof of Theorem 1.3. $\square$

4 Extension to Higher Dimensions

We extend the $O(\log^2 n \log\log n, 2+\delta)$ filter to higher dimensions. We study functions of the form $f : \{1, \ldots, n\}^d \mapsto \mathbf{R}$. An element I of the domain is referred to as a *point* and described by a d-tuple $I = \langle i_1, \ldots, i_d \rangle$. The ordering on points is defined by : $I \preceq J = \langle j_1, \ldots, j_d \rangle$ if $i_1 \leq j_1, \ldots, i_d \leq j_d$ ($I \prec J$ if $I \preceq J$ and $I \neq J$). For $I \preceq_r J$, the interval $[I, J]$ denotes the set $\{K : I \preceq K \preceq J\}$. A function f is said to have distance ε_f from monotonicity, if $\varepsilon_f n^d$ values of f must be changed to make f monotone.

For $1 \leq r \leq d$, and for $I, J \in \{1, \ldots, n\}^d$, we say that $I \preceq_r J$ if $i_s = j_s$ for all coordinates $s \neq r$ and $i_r \leq j_r$. Similary we define $\prec_r, \succeq_r$ and $\succ_r$. For $I \preceq_r J$, the set $[I, J]$ defined above can be written as $\{K : I \preceq_r K \preceq_r J\}$. For any point I (note that I is always the point $\langle i_1, i_2, \cdots, i_d \rangle$), $1 \leq r \leq d$, and $1 \leq k \leq d$, $I_r^{(k)}$ denotes the point obtained by changing the rth coordinate of I to k, i.e. $I_r^{(k)} = \langle i_1, \cdots, i_{r-1}, k, i_{r+1}, \cdots, i_n \rangle$.

4.1 Preliminaries

Definition 4.1 *Given $\delta > 0$ (arbitrarily small), a point I is said to be* right-δ-bad for dimension 1 *if there exists $J \succeq_1 I$ such that*

$$\left|\left\{K \in [I, J] \,|\, f(K) < f(I)\right\}\right| \geq (1/2 - \delta)(j_1 - i_1 + 1)$$

Inductively define I to be right δ-bad for dimension $r > 1$ *if there exists $J \succeq_r I$ such that*

$$\left|\left\{K \in [I, J] \,|\, K \textit{ is right } \delta\textit{-bad for dimension } r' < r \textit{ or } f(K) < f(I)\right\}\right| \geq (1/2 - \delta)(j_r - i_r + 1)$$

Similarly, we define left-δ-bad *by considering $J \preceq_r I$ in the above definitions. A point is* δ-bad *if it is either left δ-bad or right δ-bad for any dimension. Otherwise it is δ-good.*

It is easy to see that if I is right δ-bad for any dimension r, then it is right δ-bad for all dimensions $r' > r$ (by taking $J = I$, satisfying $J \succeq_{r'} I$ trivially). Therefore, we can equivalently define I to be right δ-bad for dimension $r > 1$ if there exists $J \succeq_r I$ such that

$$\left|\left\{K \in [I, J] \,|\, K \text{ is right } \delta\text{-bad for dimension } r - 1 \text{ or } f(K) < f(I)\right\}\right| \geq (1/2 - \delta)(j_r - i_r + 1)$$

We give a more intuitive picture to explain this definition. Let us focus on the 2-dimensional case, where the domain is a 2D-grid. Suppose we marked out a set of points, say S, such that f restricted to the *complement* of S is monotone. Now, we go through a row and mark every point that is contained in an (horizontal) interval that contains a constant fraction of marked points. After doing this for all rows, we do this for all columns. We now have a new set of marked points, which (we show later) is at most a constant factor larger than the original set of marked points. An approximation of the new set can be found in polylogarithmic time, as we later show.

We prove the following lemma (compare with Lemma 2.3):

Lemma 4.2 *Let ε_f be the distance of f from monotonicity. Then*

1. *The function f is monotone on all the 0-good points (and therefore on all δ-good points).*

2. *No more than $(2 + 4\delta/(1 - 2\delta))^d \varepsilon_f n$ points are δ-bad.*

Proof: First we prove the following statement : If $I \prec J$ and $f(I) > f(J)$, then either I is right 0-bad or J is left 0-bad. This is proved by induction on the dimension d. When $d = 1$, the proof trivially follows from the fact that $<$ is transitive. Assume this is true up to dimension $d - 1$. We will show that if $I \prec J$ violate monotonicity (that is, $f(I) > f(J)$), then either I or J is 0-bad. If $i_d = j_d$, then we can simply use the inductive hypothesis.

Otherwise, consider the points $I_d^{(k)}, J_d^{(k)}$ for k ranging in $[I_d, J_d]$. Clearly, $I \preceq I_d^{(k)} \preceq J_d^{(k)} \preceq J$. Suppose $f(I) \leq f(I_d^{(k)})$ and $f(J_d^{(k)}) \leq f(J)$. This implies the violation $f(I_d^{(k)}) > f(J_d^{(k)})$ (since $f(I) > f(J)$). By induction, either $I_d^{(k)}$ is right 0-bad or $J_d^{(k)}$ is left 0-bad (for dimension $d - 1$). Therefore for every $i_d \leq k \leq j_d$, either of the following happen -

- $f(I_d^{(k)}) < f(I)$ or $I_d^{(k)}$ is right 0-bad (for dimension $d - 1$)
- $f(J_d^{(k)}) > f(J)$ or $J_d^{(k)}$ is left 0-bad (for dimension $d - 1$)

This implies that either I is right 0-bad or J is left 0-bad, completing the proof of the first part of the lemma.

To prove the second part, we start by choosing some set $\mathcal{B} \subseteq [n]^d$ of size $\varepsilon_f n^d$ such that f is monotone on $[n]^d \backslash \mathcal{B}$ (such a set exists by definition of distance of f from monotonicity). We partition this set into the set $\mathcal{B}^\ell$ of *lower*, and the set $\mathcal{B}^u$ of *upper* points - a point $B \in \mathcal{B}$ is lower if there exists a point $C \prec B$, $C \in [n]^d \backslash \mathcal{B}$, such that $f(C) > f(B)$. On the other hand, B is upper if there exists $C \succ B$, $C \in [n]^d \backslash \mathcal{B}$, such that $f(C) < f(B)$. Because of the definition of $\mathcal{B}$, $\mathcal{B}^\ell \cup \mathcal{B}^u = \mathcal{B}$. Also, $\mathcal{B}^\ell \cap \mathcal{B}^u = \phi$. Suppose the contrary, i.e., there is a point I that belongs to both $\mathcal{B}^\ell$ and $\mathcal{B}^u$. There exists a point $C^\ell \notin \mathcal{B}$ such that $C^\ell \prec I$ and $f(C^\ell) > f(I)$. There also must exist a point $C^u \notin \mathcal{B}$ such that $C^u \succ I$ and $f(C^u) < f(I)$. Therefore, $C^\ell \prec C^u$ but $f(C^\ell) > f(C^u)$, which contradicts the fact that both C^ℓ and C^u are not in $\mathcal{B}$. Our aim is to bound the number of δ-bad points. We now define the L set and R set. The L set contains all points in $\mathcal{B}^u$ and any point $I \notin \mathcal{B}$ which is *left* δ-bad. The R set, on the other hand, contains all points in $\mathcal{B}^\ell$ and any point $I \notin \mathcal{B}$ which is *right* δ-bad. For a point $I \notin \mathcal{B}$ that is both left and right δ-bad, we arbitrarily put it in one of the sets. Note that any δ-bad point belongs to either of these sets.

Let $\mathcal{B}_1$ consist of all points I such that there exists some $J_I \preceq_1 I$ such that for at least $|[J_I, I]|(1/2 - \delta)$ points $K \in [J_I, I]$, $K \in \mathcal{B}^u$. Note that by definition all the points in $\mathcal{B}^u$ belong to $\mathcal{B}_1$. We claim that $\mathcal{B}_1$ contains every point in L that is left δ-bad for dimension 1. Let I be such a point. First, suppose $I \in \mathcal{B}$. Then $I \in \mathcal{B}^u$, and it trivially belongs to $\mathcal{B}_1$. Otherwise, for some $J_I \preceq_1 I$ there exist at least $|[J_I, I]|(1/2 - \delta)$ points $K \in [J_I, I]$ such that $f(K) > f(I)$. It must be the case that all such K's are in $\mathcal{B}^u$ (since $I \notin \mathcal{B}$), implying $I \in \mathcal{B}_1$.

We now claim that $|\mathcal{B}_1| \leq (1/2 - \delta)^{-1} |\mathcal{B}^u| = (2 + 4\delta/(1 - 2\delta))|\mathcal{B}^u|$. This claim follows from a charging scheme similar to the one used in [2] (Lemma 2.3). The following is taken directly from there - it is reproduced for completeness. Let us take a *line* - a set of points that differ *only* in the first coordinate (such a set would be represented by $\{\langle k, i_2, i_3, \cdots, i_n\rangle | 1 \leq k \leq n\}$). We assign a charge of one unit of credit to each point in $\mathcal{B}^u$. Now, we move in increasing order in the line (from lower first coordinate to higher) and for each point $I \notin \mathcal{B}^u$ that belongs to $\mathcal{B}_1$, we "spread" one unit of credit among all points $K \in [J_I, I]$ such that $f(K) > f(I)$. This is done by adding one unit of credit so that at the end of spreading, *all* such points K end up with the same amount of credit. We show that no point ever receives more than $(2 + 4\delta/(1 - 2\delta))$ units of credit. Suppose for contradiction that this did happen. Let I be the point in $\mathcal{B}_1$ that causes K to have more than $(2 + 4\delta/(1 - 2\delta))$ units of credit. All points in $\mathcal{B}_1$ that are present in $[J_I, I]$ must have more than $(2 + 4\delta/(1 - 2\delta))$ units of credit. The total amount of credit among these points is $> (2 + 4\delta/(1 - 2\delta)) \times (1/2 - \delta)|[J_I, I]| = |[J_I, I]|$. But, these points could not have received credit from points smaller than J_I and the total credit accumulated so far must be $\leq [J_I, I]$ (contradiction). This proves that (within a line) the total number of points in $\mathcal{B}_1$ is at most $(2 + 4\delta/(1 - 2\delta))$ times the number of points in $\mathcal{B}^u$. Applying this argument for all lines, we prove that $|\mathcal{B}_1| \leq (2 + 4\delta/(1 - 2\delta))|\mathcal{B}^u|$.

To bound the number of left δ-bad points for dimension $r > 1$, we consider the (inductively defined) set $\mathcal{B}_{r-1}$ and define $\mathcal{B}_r$ to be the set of all points $I \in [n]^d$ such that there exists some $J \preceq_r I$ such that for at least $|[J, I]|(1/2 - \delta)$ points $K \in [J, I]$, $K \in \mathcal{B}_{r-1}$. The set $\mathcal{B}_r$ contains all points in L which are left δ-bad for dimension r. To see this, consider $I \notin \mathcal{B}$ which is left δ-bad for dimension r. There exists $J_I \preceq_r I$ such that for at least $(1/2 - \delta)[J_I, I]$ points $K \in [J_I, I]$, $f(K) > f(I)$ or K is left δ-bad for dimension $(r - 1)$. Using induction and arguments given above, we can show that $I \in \mathcal{B}_r$. By applying the charging argument used for $\mathcal{B}_1$, we can also claim that $|\mathcal{B}_r| \leq (2 + 4\delta/(1 - 2\delta))|\mathcal{B}_{r-1}|$ (here, the charging argument will charge along lines which contain points that differ only in the rth coordinate). By induction, $|\mathcal{B}_d| \leq (2 + 4\delta/(1 - 2\delta))^d |\mathcal{B}^u|$ and $\mathcal{B}_d$ contains all of L. Similarly we can bound the number of points in R in terms of $|\mathcal{B}^\ell|$. Taking the sum of these two bounds, we prove that the total number of δ-bad points is as stated in the lemma. $\square$

```
                         bad-good-test (f, I, δ, k, r)

repeat the following c log k times for a big enough constant c
    for each 1 ≤ j ≤ (2/δ) ln n
        define I_j = (i_1, i_2, ··· , i_r + (1 + δ)^j, ···)
        define x_j such that x_j = 1 iff for random I' ∈ [I, I_j], check(f, I, I', δ, r) is true
    let D be the distribution (x_1, x_2, ···)
    test if D is (1/2 − 2δ)-light or (1/2 − 3δ/2)-heavy with probability 2/3

if a majority of the above tests output (1/2 − 2δ)-light
    output "2δ-good"
else
    output "δ-bad"

                         check (f, I, I', δ, r)
if f(I') < f(I)
    output "true"
if r > 1 and bad-good-test(f, I', δ, n^2 k, r − 1) outputs "δ-bad"
    output "true"
else
    output "false"
```

Figure 6: Testing if a point I is right-δ-bad or right-2δ-good for dimension r.

We now extend the procedure *bad-good-test* to higher dimensions (Figure 6). For simplicity of presentation, *bad-good-test* as described will only distinguish between right-2δ-good and right-δ-bad. Extending this to check left-2δ-good vs left-δ-bad will be obvious.

The procedure *bad-good-test*(f, I, δ, k, r) will output whether I is 2δ-good or δ-bad for dimension r with error probability $1/k$. The x_j's represent 0/1 random variables. To separate 2δ-good points from δ-bad ones, *bad-good-test* is called with dimension d (last argument). This procedure is almost a direct extension of the one-dimensional procedure. It goes through every dimension and checks for violations along one-dimensional intervals. Of course, here "violation" can refer to a standard violation or to a point that is bad for a lower dimension. The procedure is simply used recursively for finding bad points for lower dimensions.

Lemma 4.3 *Given any fixed $\delta > 0$ and parameter $0 < k < n^{6d}$, if I is either 2δ-good or δ-bad, then* bad-good-test *will tell which is the case in time $(2d)^{O(d)}(\log n)^{2d-1}\log k$ and with probability $1 - 1/k$.*

Proof: Correctness is proved by induction on the dimension. When $r = 1$, *bad-good-test*(f, I, δ, k, r) gives the right output with error probability $1/k$ (Lemma 2.2). Assume up to r. For $r + 1$, the probability that any of the calls to *bad-good-test*(f, I', δ, n^2k, r) errs is $< (nk)^{-1}$. The value c can be chosen large enough to ensure that the total probability of error is $1/k$ (note that this c is independent of any parameter).

We prove by induction on the dimension that *bad-good-test*(f, I, δ, k, r) runs in time $(Cd)^r(\log n)^{2r-1}\log k$ (for some large constant C). The case $r = 1$ is proven in Lemma 2.2. Assume up to r. For $r+1$, each sampling of the random variables x_j requires a call to *bad-good-test*(f, I', δ, n^2k, r), which takes $(Cd)^r(\log n)^{2r-1}\log(n^2k) \leq (Cd)^{r+1}(\log n)^{2r}$ time (since $k < n^{6d}$). To test if $\mathcal{D}$ is $(1/2-2\delta)$-light or $(1/2-3\delta/2)$-heavy with probability 2/3 requires $O(\log n)$ samples of the x_j's (Lemma 2 of [2]). Since this whole procedure is repeated $O(\log k)$ times, the total running time is $(Cd)^{r+1}(\log n)^{2r+1}\log k$. □

4.2 Reconstruction

In this section, we discuss how to correct f by assigning appropriate function values to points that are δ-bad (wlog, we will assume that the initial function values of f are all distinct). The aim of this section is to prove the following theorem.

Theorem 4.4 *For any $0 < \delta < 1/2$, there exists a $(2^{O(d)}(\log n)^{4d-2}\log\log n, (2^d + d\delta))$-filter for d-dimensional monotonicity.*

We will now refer to 2δ-good points as good, and δ-bad points as bad. Let the number of bad points be γn^d (by Lemma 4.2, $\gamma = (2^d + O(d\delta))\varepsilon_f$). We assume that all committed points (with their function values) are stored in a data structure $\mathcal{C}$ that can support range search queries. Namely, given a point I, the data structure can determine the values $\max\{f(J)|J \prec I, J \in \mathcal{C}\}$ and $\min\{f(J)|J \succ I, J \in \mathcal{C}\}$. Note that these ranges are orthogonal. There are data structures [3] that take $O((\log n)^{d+2})$ for queries and updates.

The reconstruction procedure becomes much more complicated in higher dimensions. The essential difference is that the domain we are now focussing on is a partial order, not a complete order as in the one dimensional case. In one dimension, finding a safe replacement value for a bad point I can be done by finding the f value of the largest good point less than I. A sublinear procedure finds a point that is close enough. Adding a dimension complicates matters considerably. Indeed, there could be a large (possibly linear) set of good points all less than I which are mutually incomparable. Instead of just a single point, now a *set* of points defines the replacement value. Consider the two-dimensional domain given in Figure 7. All points below the slanted line are good, and all those above are bad. A safe replacement value for I would have to be larger than the f values of all (or, at the very least, most) the points on the slanted line to prevent destruction of good points. Note how the issue of early decisions becomes very crucial here. A choice of the replacement value for I would somehow have to consider all these f values. The main challenge is do this in sublinear time.

The procedure *monotonize* for higher dimensions is quite similar to the one-dimensional case. A small difference is that commitment is done pointwise, and not in intervals as was done for one dimension. First, *monotonize* checks the input point for goodness. It also checks whether the value $f(I)$ is consistent with previously committed values. If either of these fails (and the value has to be changed), a recursive procedure *get-value* determines a replacement for $f(I)$. This procedure highlights the main difference for the higher dimensional case. The base case for this procedure is *get-value*$_1(f, I, \delta)$ - this refers to a procedure that, with probability $1 - 1/n^{6d}$ and in time $O(d\log^2 n\log\log n)$, outputs a value in $[f(I'), f(I'')]$ where I', I'' are good, $I' \preceq_1 I'' \preceq_1 I$, and such that

$$\left|\left\{J \in [I', I] | J \text{ good}\right\}\right| \leq \delta \left|[I', I]\right|$$

Note that this is just finding a "close" good value along a one-dimensional line containing I, and therefore has essentially been discussed in Section 2 (more specifically, it is the code between

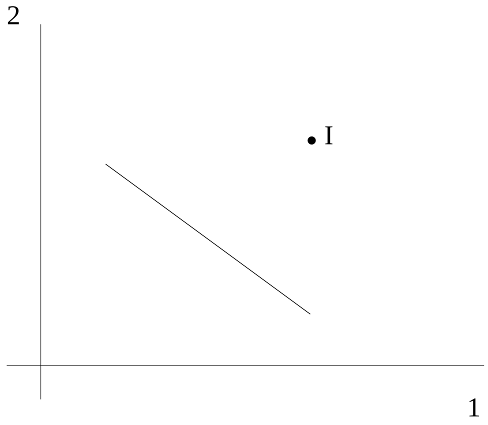

Figure 7: Issues in higher dimensions

(*) and (**) in Figure 5). We will assume that such a value always exists (this can be ensured by padding with dummy values). As a result, *get-value* always outputs a value sandwiched between the f-values of two good points less than I. To show the correctness of *monotonize*, we need to prove that not too many good points are destroyed by the values output by *get-value*.

Before we give a detailed description of how this procedure works, we first provide some intuition. Again let us focus on the 2-dimensional case and think of the first dimension as horizontal and the second as vertical. For the sake of simplicity, let us assume that given a point I, we can find the largest good point J such that $J \preceq_1 I$ - in other words, the closest good point in the same row as I and less than it. This point is referred to as $frontier(I)$ - we later define this in an approximate manner which allows us to find frontiers in sublinear time. A possible polynomial time reconstruction procedure would be to go to every point $K \preceq_2 I$ (we are looking at points in the same *column* as I). For each K, we find $frontier(K)$. In Figure 8, the points K_1, K_2, K_3 have the same first coordinate as I. For each such K_i, J_i is its frontier. We end up with a collection of $\Theta(n)$ points (consisting of all frontier points), and we take the maximum f value among these points and use this as the replacement value. Note that in the example of Figure 7, this procedure would look at the function value of every point on the slanted line.

We now intuitively explain how this process is approximated by a sublinear procedure in a sense that will be made precise later. Let us take all linear intervals along the second dimension (vertical intervals) having I as their right endpoint and have a length of a power of 2 - the set of exponentially increasing intervals. In each such an interval, we choose a small random sample of points. Let K be such a point. We then find $J = frontier(K)$. This leaves us with a polylogarithmic sized set of points J, which we use to reconstruct the value at I. Naturally, this is insufficient to capture all points less than I and this replacement value for I can violate some *good* points less than I. The maximum function value of all these points is (roughly speaking) denoted by $bound(I)$[2]. This value tells us that for any good $I' \prec I$, if $f(I') \leq bound(I)$, then this sublinear reconstruction procedure will *not* create violation with I. These points are *safe*. The point I' might create a violation if I' is *unsafe* (if $f(I') > bound(I)$). In this case, we will be forced to change the value at I', even though it is good. We show that the number of such unsafe points is very small.

[2] Of course, this process is randomized, so such a value cannot be determined purely by I. We give an exact definition later which does not have this problem, but the essence is still the same.

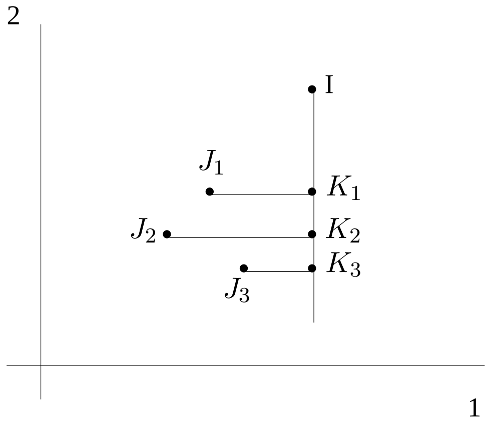

Figure 8: Reconstruction

This idea is used recursively for reconstruction in higher dimensions. Suppose we are now in a three-dimensional domain. Again, we take all exponentially increasing linear intervals along the *third* dimension, and choose a small random sample within each set. For each such point K, we cannot uniquely assign the closest good point. Instead, we run the *two*-dimensional procedure described above on K. This will look at the two-dimensional plane of points having the same third coordinate as K and return a polylogarithmic set of points. We collect all points obtained after running the two-dimensional reconstruction procedure for all such K, and use these points to reconstruct the value at I.

We now describe the above intuition precisely.

Definition 4.5 *Define $frontier(I)$ to be the smallest I' such that $I' \prec_1 I$, I' is good, and*

$$\left|\left\{J \in [I', I] | J \text{ good}\right\}\right| \leq \delta |[I', I]|$$

The value $f(frontier(I))$ is denoted by $bound_1(I)$. If I is good and for some J, $I \in (frontier(J), J]$, then I is called 1-unsafe.

The value output by *get-value*$(f, I, 1, \delta)$ is (with high probability) greater than or equal to $bound_1(I)$.

Definition 4.6 *For $1 < r \leq d$, $bound_r(I)$ is defined recursively. Take any interval of the form $[I', I]$, where $I' \prec_r I$. Consider the set $S = \{bound_{r-1}(J) | J \in [I', I]\}$. The smallest value greater than or equal to $(1-\delta)|S|$ elements of S is denoted $bound_r(I)$.*

I is r-unsafe *if I is good and there exists a $J = \langle j_1, \cdots, j_r, i_{r+1}, \cdots, i_d \rangle \succ I$ (called the witness) such that $bound_r(J) < f(I)$.*

Points which are good and not r-unsafe (for all $1 \leq r \leq d$) are referred to as *safe* points. First, we will bound the number of unsafe points in terms of γn^d. Then, we will show (with high probability) that *monotonize* does not change f values at safe points.

get-value (f, I, r, δ)

```
if r = 1, then output get-value_1(f, I, δ)
let L be an empty list
for each 1 ≤ j ≤ (2/δ) ln n
    for each 1 ≤ k ≤ (32/δ)d ln n
        randomly select l ∈ [i_r − (1 + δ/2)^j, i_r]
        append get-value(f, I_r^(l), r − 1, δ) to L
output maximum value of L
```

Figure 9: Finding an appropriate value for $f(I)$.

Lemma 4.7 *For any $1 \leq r \leq d$, the number of r-unsafe points is bounded by $5\delta\gamma n^d$.*

Proof: The proof is done by induction on r. First, we prove the base case. Consider a 1-unsafe point I. It lies in the interval $[frontier(J), J]$ for some $J = \langle j, i_2 \cdots, i_n \rangle$. Among the intervals $[frontier(J), I]$ and $[I, J]$, one of them must have $\leq 2\delta$ fraction of good points. If the former occurs, call I *downward*, otherwise call it *upward* (if both happen, we assign any name).

Choose all points of the form $\langle k, i_2, \cdots, i_d \rangle$, $1 \leq k \leq n$. We will use a charging argument over this set of points to show that the number of upward points is at most a $2\delta/(1-2\delta)$ fraction of bad points. We go in reverse order $\langle n, i_2 \cdots, i_n \rangle, \cdots, \langle 1, i_2 \cdots, i_n \rangle$, and for each unsafe point I, one unit of charge is spread onto the bad points in the interval $[I, J]$. At some stage, suppose some bad point K get a charge $> 2\delta/(1-2\delta)$ while some I is being processed. Because we spread charge uniformly, every bad point in $[I, J]$ has charge $> 2\delta/(1-2\delta)$. Since the number of bad points in this interval is $\geq (1-2\delta)$ fraction of the whole interval, the total charge is $> 2\delta(j - i_1 + 1)$. This charge could only have come from all the good points in $[I, J]$ (since unsafe points are always good), which are $\leq 2\delta(j - i_1 + 1)$ in number. That leads to a contradiction.

Similarly, we bound the number of downward points, by traversing the column in the order $\langle 1, i_2 \cdots, i_n \rangle, \cdots, \langle n, i_2 \cdots, i_n \rangle$. Therefore, the total number of unsafe points is bounded by $(4\delta/(1-2\delta))\gamma n^d < 5\delta\gamma n^d$.

Now for the induction step - assume that the statement is true for all $r' < r$. For any point I, let R_I denote the set $\{I_r^{(l)} | 1 \leq l \leq n\}$. A charging argument (as above) will be applied to each such set of points. First, we mark all bad and $(r-1)$-unsafe points in R_I. Then going in reverse order $I^{(n)}, \cdots, I^{(1)}$ (for clarity, the subscripts have been dropped), we process each r-unsafe point $I^{(k)} \in R_I$. Let its witness be $J = \langle j_1, \cdots, j_r, i_{r+1}, \cdots, i_n \rangle \succ I^{(k)}$. We begin by proving the following claim.

Claim 4.8 *The interval $[I^{(k)}, I^{(j_r)}]$ can have at most a δ-fraction of unmarked points.*

Every unmarked point $I^{(l)}$ in the interval is good, and therefore $f(I^{(k)}) \leq f(I^{(l)})$. Since $I^{(l)}$ is not $(r-1)$-unsafe, $f(I^{(l)}) \leq bound_{r-1}(J^{(l)})$, showing that for any unmarked point $I^{(l)} \in [I^{(k)}, I^{(j_r)}]$, $f(I^{(k)}) \leq bound_{r-1}(J^{(l)})$. Since J is a witness, $f(I^{(k)}) > bound_r(J)$. This means that $f(I^{(k)})$ is greater than a $(1-\delta)$-fraction of values in the set $\{bound_{r-1}(J^{(l)}) | k \leq l \leq j_r\}$, proving the claim.

For processing $I^{(k)}$, we take one unit of charge and spread it over all marked points in the interval $[I^{(k)}, I^{(j_r)}]$. Using an argument similar to the one used above, we can show that each marked point

monotonize (f, I, δ)

```
if ĝ(I) was already committed
    output ĝ(I)
let l = max{f(J)|J < I, J ∈ C} and r = min{f(J)|J > I, J ∈ C}
if l ≤ f(I) ≤ r and bad-good-test(f, I, δ, n^{5d}, 2) outputs "2δ-good"
    commit to ĝ(I) = f(I) by inserting in C
    output ĝ(I)
let val = get-value(f, I, d, δ)
if val < l
    val = l
if val > r
    val = r
commit to ĝ(I) = val by inserting in C
output ĝ(I)
```

Figure 10: The filter

ends up with a charge of at most $\delta/(1-\delta)$. The total number of marked points is $\leq (1+5\delta)\gamma n^d$. Therefore, the number of r-unsafe points is at most $2\delta\gamma n^d < 5\delta n^d$. □

Lemma 4.9 *With probability $> 1 - n^{-5d}$ and in time $2^{O(d)}(\log n)^{4d-2} \log\log n$,* get-value$(f, I, d, \delta)$ *outputs a value which is consistent with all safe points i.e. for all safe J, if $J \succ I$, $f(J) \geq I$ and if $J \prec I$, $f(J) \leq I$.*

Proof: First we prove the following statement : With probability $> 1 - n^{-5d}$, the output of *get-value*(f, I, d, δ) is larger than *bound*$_d(I)$.

We show by induction over r that, with probability of error $< (\log n)^{cr} n^{-6d}$ for some sufficiently large constant c, the output of *get-value*(f, I, r, δ) is $\geq$ *bound*$_r(I)$. Let us denote by I_r^j the point $I_r^{(i_r-(1+\delta)^j)}$ (just to reduce clutter). For $r = 1$, we know that *get-value*$_1$ outputs a value larger than *bound*$_1(I)$ with error probability $< 1/n^{6d}$, proving the base case. Assume inductively upto r. We now prove for $r+1$. For every $1 \leq j \leq (2/\delta)\ln n$ (refer to Figure 9), consider the interval $[I_{r+1}^j, I]$. Let L be the list as defined in Figure 9. Note that the maximum of L is the output of *get-value*$(f, I, r+1, \delta)$. By the induction hypothesis and a union bound (over the $O(\log n)$ values of k in Figure 9), the probability that any call to *get-value*$(f, I_r^{(l)}, r, \delta)$ errors is $O((\log n)^{cr+1} n^{-6d})$. This fact, combined with a standard Chernoff bound argument, tells us that L contains a value larger than a $(1-\delta/2)$-fraction of values from the set $\{\mathit{bound}_r(I') | I' \in [I_{r+1}^j, I]\}$ with probability of error $< (\log n)^{cr+2} n^{-6d}$. Taking a union bound over all j, we prove that the maximum of L is larger than *bound*$_{r+1}(I)$ with probability of error $< (\log n)^{c(r+1)} n^{-6d}$. This completes the proof of the statement.

Let us denote the output of *get-value*(f, I, d, δ) by v. Note that v is between the values of two good points both smaller than I. Therefore, for any safe point $S \succ I$, $f(S) \geq v$. Suppose $S \prec I$. By definition, $f(S) \leq \mathit{bound}_d(I) \leq v$, and the values are consistent.

For the running time, note that *get-value* recurses on the dimension parameter - we can inductively show that the running time of *get-value*(f, I, r, δ) is $2^{O(d)}(\log n)^{2(d+r-1)} \log\log n$, proving the stated bound. $\square$

Finally, to prove Theorem 4.4, we prove the following :

Lemma 4.10 *Given* $0 < \delta < 1/2$, monotonize *computes a monotone function* $\hat{g}$ *such that* $dist(f, \hat{g}) < (2^d + O(d\delta))\varepsilon_f$ *with probability* $> 1 - n^{-2d}$. *The running time of* monotonize *is* $2^{O(d)}(\log n)^{4d-2} \log\log n$ *per query.*

Proof: The call to *bad-good-test* takes $(2d)^{O(d)}(\log n)^{2d-1}$ time. The running time for the call to *get-value* is $2^{O(d)}(\log n)^{4d-2} \log\log n$, which is more expensive (assuming of course, that n is sufficiently large). Henceforth, a *sound* value for a point that is one that when assigned to it will be consistent (with regard to monotonicity) with f-values at safe points. If, after some queries, the output is *sound*, we mean that no violations with f-values at safe points are present.

It is easy to see that *monotonize* always outputs a monotone function. Since we know that each call to *bad-good-test* and *get-value* errs with probability $< n^{-5d}$ (Lemmas 4.3, 4.9), the probability that any such call errs is certainly $< n^{-2d}$. Assuming now that these calls will not error, we will prove that *monotonize* will always output values that keep it sound. Let us prove this by induction on the number of queries processed by *monotonize*. For the base case, any good point will be committed to its f-value. Note than the f-value at *any* good point is sound. For a bad point, by Lemma 4.9, we will commit to a value that is sound.

Assume up to t queries. (In the following, the variables l, r, val are as defined in Figure 10.) For the $(t+1)$th query, if the point I is good and $f(I) \in [l, r]$, soundness will be maintained. Otherwise, we will have to run *get-value*. If $val \in [l, r]$, we are done (again by Lemma 4.9). Suppose $val > r$. Consider the committed point $J \succ I$ such that $\hat{g}(J) = r$. We know by induction that $f(K) \leq r$ for all safe $K \prec I \prec J$. For any safe $K \succ I$, $f(K) \geq val > r$. Therefore, r is a sound value for I. The case where $val < l$ is handled similarly.

This shows that f is modified only at points that are unsafe. The number of those (by Lemmas 4.2, 4.7) is $(2^d + O(d\delta))\varepsilon_f n^d$. Rescaling δ gives us the result. $\square$

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] Agarwal, P., Erickson, J. *Geometric Range Searching and its Relatives*, Advances in Discrete and Computational Geometry, 1999, 1–56.

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

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

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

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

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

[9] Hoffmann, C.M., Hopcroft, J.E., Karasick, M.S. *Towards implementing robust geometric computations*, Proc. 4th SOCG, 1988, 106–117.

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

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

[12] Reed, B. *The Height of a Random Binary Search Tree.* Journal of the ACM (50), 2003, 306–332.

[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.

[15] Salesin, S., Stolfi, J., Guibas, L.J. *Epsilon geometry: building robust algorithms from imprecise computations*, Proc. 5th SOCG, 1988, 208–217.