Optimal No-Regret Learning for One-Sided Lipschitz Functions

Paul Dütting [*1] Guru Guruganesh [*2] Jon Schneider [*3] Joshua R. Wang [*2]

Abstract

Inspired by applications in pricing and contract design, we study the maximization of one-sided Lipschitz functions, which only provide the (weaker) guarantee that they do not grow too quickly in one direction. We show that it is possible to learn a maximizer for such a function while incurring $O(\log \log T)$ total regret (with a universal constant independent of the number of discontinuities / complexity of the function). This regret bound is asymptotically optimal in T due to a lower bound of Kleinberg and Leighton. By applying this algorithm, we show that one can sell digital goods to multiple buyers and learn the optimal linear contract in the principal-agent setting while incurring at most $O(\log \log T)$ regret.

1. Introduction

Traditionally, the fields of online learning and optimization have primarily been concerned with optimizing continuous, well-behaved functions (e.g., Lipschitz continuous functions, or even bounded convex functions). However, many practical problems (especially those stemming from economics) involve optimizing a function with *asymmetric discontinuities*.

For example, consider the problem of setting a price x for selling a single item to a buyer with an (unknown) fixed value p for the item. The goal of the seller is to maximize their revenue as a function of x (by setting multiple different prices over time, and observing their revenue in each case). This function is *not continuous* in x: as x exceeds p, their revenue sharply drops from p down to 0. However, this function is not entirely pathological either – we still have the nice property that increasing the price x by some amount δ increases our revenue by at most δ. In other words, the function we wish to optimize is "Lipschitz continuous in one direction" (however, note that decreasing the price x by a small amount δ might cause the revenue to increase dramatically).

Formally, if a function $f : [0, 1] \to \mathbb{R}$ has the property that $f(x + \delta) \leq f(x) + L\delta$ for all $x \in [0, 1]$ and $\delta \geq 0$, we say it is *L-right-Lipschitz continuous*. In the scenario above, the revenue function $f(x)$ representing the revenue at a price x is a particularly simple example of a (1-)right-Lipschitz continuous function, being piece-wise linear with two pieces. More complex one-sided Lipschitz continuous functions arise in other economics contexts; see Section 1.1 for two motivating examples related to setting prices for digital goods and learning linear contracts, where the relevant functions can have arbitrarily many points of discontinuity.

In this paper, we study one of the simplest and most fundamental questions related to (online) optimization of such functions: how to design low-regret (compared to always querying the optimal point) algorithms for optimizing a single one-sided Lipschitz continuous function f from *determinstic, noise-free* feedback. Specfically, we consider the following game that takes place over T rounds. In each round t, our algorithm can choose an input $x^{(t)}$ and learns the value of $f(x^{(t)})$. The goal of the algorithm is to maximize the total sum $\sum_{t=1}^{T} f(x^{(t)})$ (with the intent that the algorithm eventually learns the maximizer of f). The *regret* of the algorithm over these T rounds is given by the difference $T(\max f) - \left(\sum_{t=1}^{T} f(x^{(t)})\right)$, where $(\max f)$ is the maximum value of f.

Our primary contribution is the construction of an efficient algorithm for this problem with an asymptotically optimal regret bound. In particular, we show it is possible to optimize *any* L-left(/right)-Lipschitz continuous function from $[0, 1]$ to $[0, L]$ while incurring at most $O(L \log \log T)$ regret:

Theorem 1.1 (Restatement of Corollary 3.2 and Theorem 5.1)**.** *There exists a deterministic and efficient online algorithm (Algorithm 2) for maximizing an L-left-Lipschitz continuous function mapping* $[0, 1] \to [0, L]$ *over* T *rounds that incurs at most* $O(L \log \log T)$ *regret. Moreover, any learning algorithm must incur at least* $\Omega(L \log \log T)$ *for this problem.*

[*]Equal contribution [1]Google Research, Zürich, Switzerland [2]Google Research, Mountain View, USA [3]Google Research, New York City, USA. Correspondence to: Joshua R. Wang <joshuawang@google.com>.

Proceedings of the 40th International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023. Copyright 2023 by the author(s).

The lower bound of $\Omega(L \log\log T)$ can be shown by considering the selling a single item problem mentioned above, and in fact quickly follows from a similar lower bound proven by (Kleinberg & Leighton, 2003). However, matching this regret bound is technically challenging and requires synthesizing several different ideas from the analysis of online algorithms. In particular, note that the naive strategy of simply querying every point along an equally spaced grid achieves a regret of at best $O(L\sqrt{T})$.

At a high level, our algorithm maintains a collection of candidate intervals (possibly containing the maximizer of f). Our algorithm strives to reduce the maximum width among all intervals by taking the widest interval and making queries to divide it further. The crux of the analysis involves noticing that to efficiently divide an interval without incurring too much regret, we should essentially switch between running one of two different optimization algorithms based on which "regime" the interval fits into: in one case we should run a standard (bisection-based) binary search, but in the second case we should run an *asymmetric binary search* (influenced by the asymmetric binary search procedure of (Kleinberg & Leighton, 2003)) where we bias our queries towards one end of the interval. In order to reconcile the different rates at which these two algorithms shrink intervals, the analysis of our algorithm hinges on a single potential function that both algorithms are charged against.

Organization. The remainder of this paper is organized as follows. In the remainder of this section we discuss (the previously mentioned) two motivating applications for learning to optimize one-sided Lipschitz functions, and provide a brief survey of existing work on learning one-sided Lipschitz functions and loss functions with discontinuities. Then, after setting up the problem in Section 2, we begin by presenting the $\Omega(L \log\log T)$ lower bound in Section 3. In Section 4, we present two learning algorithms for very specific subcases of this problem (where either f is a single spike, or we know the maximum value of f ahead of time) whose ideas play an important role in our analysis of the main algorithm. Finally, in Section 5, we present our $O(L \log\log T)$ regret algorithm along with a sketch of its proof.

1.1. Applications

We discuss two motivating examples in more detail below (where the associated one-sided Lipschitz function can be far more complex), one from mechanism design and one from contract design:

Example 1: Selling digital goods to multiple buyers. Consider the generalization of the problem above where instead of selling a single item to a single buyer, the seller is selling copies of some digital good (i.e., one where they can produce arbitrarily many copies at no cost) to an audience of n buyers (see e.g. (Goldberg et al., 2001)). Each buyer i has a threshold value p_i (where they will buy the item at prices below p_i and not otherwise). The seller must set a single price x for the item. Upon doing this, they receive an average revenue per buyer of $f(x) = (x \cdot \#\{i \mid p_i \geq x\})/n$. Of course, the seller may not know these p_i (or even the value of n) in advance, but instead may have the opportunity to set different prices over time and observe the values of $f(x)$, hoping to maximize $f(x)$ in an online fashion. Similar to the original example, this function f is 1-left-Lipschitz continuous, however the function itself is considerably more complex (containing n discontinuities instead of just one).

Example 2: Learning the optimal linear contract. In the classical hidden-action principal-agent problem (Holmström, 1979; Grossman & Hart, 1983), an agent must choose one of n different actions, each of which has some cost to the agent and results in a distribution over outcomes with differing utilities for the principal. The principal would like to incentivize the agent to choose an action which is beneficial to the principal – however, the principal cannot directly observe the action chosen by the agent, only the eventual outcome.

Payment mechanisms the principal can use to incentivize the agent are known as *contracts* and are the main focus of the area of contract theory. One particularly significant subclass of contracts, notable for their simplicity and robustness (see e.g. (Carroll, 2015; Dütting et al., 2019)) are *linear contracts*. In a linear contract, the principal promises to transfer an α fraction (for some $\alpha \in [0, 1]$) of their received utility to the agent. The agent chooses their action in response to this contract and the principal receives some expected net utility $U(\alpha)$ in response.

Existing papers in the contract theory literature (e.g. (Dütting et al., 2019)) point out that directly optimizing $U(\alpha)$ (and in fact, finding the *optimal non-linear contract*) can often be done efficiently, but this assumes knowledge of all of the agent's actions, their costs, and their likelihoods of inducing certain outcomes – information which the principal may not possess. Instead, the principal may wish to learn the optimal linear contract over time (by offering different linear contracts α and seeing how they perform on average). We can show (see Appendix A) that $U(\alpha)$ is a left-Lipschitz continuous function, and therefore this learning problem fits directly into our framework. We additionally briefly remark that the problem of learning the optimal linear contract in more complex principal agent settings, such as those with combinatorial action spaces (Dütting et al., 2021) or multiple types of agent (Guruganesh et al., 2021) also result in a right-Lipschitz continuous utility function for the principal, and hence also lie within our framework. In these applications the number of spikes can be exponential in the input size.

1.2. Related Work

The most important predecessor to our work is (Kleinberg & Leighton, 2003), which considers the problem of selling a single indivisible good. As discussed above their approach for *deterministic, noise-free* feedback does not work for our more complex setting, and a number of additional ideas are required for our $O(\log\log T)$ bound. We do rely on this work by showing how we inherit their $\Omega(\log\log T)$ lower bound. A main new insight from our work is that the complexity of the function (in particular the number of discontinuities) does *not* enter the regret bound.

There is a very broad literature that studies the application of learning to problems in pricing and mechanism design and some of this literature also contends with the one-sided asymmetry of the functions that arise. (Cesa-Bianchi et al., 2017) study a bandit setting that models (among other things) setting reserve prices in repeated auctions, and one of their regret benchmarks involves competing with the best one-sided Lipschitz function (extending previous work on Lipschitz bandits, e.g. (Slivkins, 2014)) . Another relevant line of work is the recent literature on (noise-free) contextual pricing (Cohen et al., 2016; Lobel et al., 2017; Paes Leme & Schneider, 2018; Mao et al., 2018; Liu et al., 2021), which considers the problem of repeatedly selling heterogeneous goods with multidimensional features (where the value of the buyer for a single good is given by some fixed, unknown linear function of the feature vector of that good); this work also requires adapting the asymmetric binary search algorithm of (Kleinberg & Leighton, 2003) to a more complex setting. Of these papers, perhaps the most similar to our setting is the work of (Emamjomeh-Zadeh et al., 2021), which studies a variant of contextual pricing where the seller must pick one of several products and set a price. For the pricing problem in Example 1, it is possible to apply the results of (Emamjomeh-Zadeh et al., 2021) to obtain an $O(n\log\log T)$ regret algorithm for n buyers; note that our result removes the dependence on n entirely. To the best of our knowledge, there has been no prior work directly focused on the main topic of this paper (learning to maximize a fixed one-sided Lipschitz continuous function).

Over the past few years, contract theory has received a surge of attention from an algorithmic point of view (Alon et al., 2021; Castiglioni et al., 2021; 2022; Dütting et al., 2019; 2020; 2021; 2023; Guruganesh et al., 2021). The majority of these papers make the assumption that the principal already knows all the relevant details of the underlying problem. (Ho et al., 2014) study an adaptive-discretization style algorithm for learning the best (non-linear) contract under stochastic feedback. More recently, (Zhu et al., 2022) prove tight bounds on the regret possible in the setting of Ho et al., and exhibit a separation between the problem of learning linear contracts (where in their setting it is possible to achieve $O(T^{2/3})$ regret) and the problem of learning general contracts (where the regret must scale as $\Omega(T^{1-\varepsilon})$, with $\varepsilon \to 0$ as the number of outcomes increases). We note here that both papers imply very little about what is possible in the noise-free setting, and the algorithms we present are very different in techniques from the algorithms present in these works (e.g., the optimal $O(T^{2/3})$ regret algorithm for learning linear contracts in a stochastic setting simply involves running UCB over an appropriately chosen subset of equally spaced linear contracts).

2. Preliminaries

Our main object of study is one-dimensional functions that are only Lipschitz continuous in one direction.

Definition 2.1 (One-Sided Lipschitz Functions)**.** A one-dimensional function f is L-left-Lipschitz continuous if for all $x, y \in \text{dom}(f)$ where $x \le y$:

$$f(x) - f(y) \le L \cdot (y - x)$$

Similarly, f is L-right-Lipschitz continuous if for all $x, y \in \text{dom}(f)$ where $x \le y$:

$$f(y) - f(x) \le L \cdot (y - x)$$

Intuitively, right-Lipschitz functions cannot increase too quickly if you start at a point and move to the right (i.e. increase x); left-Lipschitz functions, move to the left. By applying linear transformations, we can easily manipulate the direction, domain, range, and Lipschitz constant of a function. To manipulate direction, if $f(x)$ with domain $[D_\ell, D_r]$ is L-right-Lipschitz continuous, then $g(x) \triangleq f(D_\ell + D_r - x)$ has the same domain and range and is L-left-Lipschitz continuous. To manipulate the domain: if $f(x)$ has domain $[D_\ell, D_r]$ and is L-left-Lipschitz continuous (resp. L-right-Lipschitz continuous), then $g(x) \triangleq f(D_\ell + x \cdot (D_r - D_\ell))$ has the same range and domain $[0, 1]$ and is $L(D_r - D_\ell)$-left-Lipschitz continuous (resp. $L(D_r - D_\ell)$-right-Lipschitz continuous). To manipulate range, if $f(x)$ is L-left-Lipschitz continuous (resp. L-right-Lipschitz continuous) with range $[R_\ell, R_r]$, then $g(x) \triangleq R \cdot \frac{f(x)-R_\ell}{R_r-R_\ell}$ has the same domain and range $[0, R]$ and is $\frac{LR}{R_r-R_\ell}$-left-Lipschitz continuous (resp. $\frac{LR}{R_r-R_\ell}$-right-Lipschitz continuous).

By using the first two transformations, we can focus our attention on L-left-Lipschitz functions with domain $[0, 1]$. However, the final transformation scales the range and Lipschitz constant by the same factor and hence preserves their ratio. The regret of our algorithm scales with the larger of the Lipschitz constant and the range size (after normalizing the domain size), so we round up the size of the smaller quantity and assume they are equal for the remainder of the paper (it is free to pretend that the range or Lipschitz constant is larger than its actual value). Putting everything together, we can transform any $f(x)$ with domain $[D_\ell, D_r]$, range $[R_\ell, R_r]$, and left-Lipschitz constant L (or right-Lipschitz constant L) into a $g(x)$ with domain $[0, 1]$, range $[0, 1]$ and left-Lipschitz constant 1. Since regret is an additive notion of error and we needed to scale the range/Lipschitz constant by $\min\left\{(R_r - R_\ell)^{-1}, (L(D_r - D_\ell))^{-1}\right\}$, the regret on the original function gets scaled up by $\max\left\{(R_r - R_\ell), L(D_r - D_\ell)\right\}$. With that factor in mind, we can now focus on the following problem.

In the LEFTLIPSCHITZMAXIMIZATION problem, we are given a 1-left-Lipschitz function $f : [0, 1] \to [0, 1]$ and the number of rounds T. In each round t, an (online) algorithm chooses an $x^{(t)} \in [0, 1]$ and then is told $f(x) \in [0, 1]$. The goal is minimize the total regret, defined to be the difference between the value of the points that the algorithm chose and the value of playing the single best point repeatedly:

$$R(\text{ALG}, f) \triangleq \sum_{t=1}^{T} \Big((\max f) - f(x^{(t)})\Big)$$

Remark 2.2. It is a standard trick to remove an online algorithm's dependence on knowing T by guessing T and gradually increasing the guess (resetting the algorithm in the process) when it is violated. For an algorithm with $O(\log\log T)$ dependence on T, the appropriate guess scaling is $T = 2\hat{}2\hat{}2\hat{}i$, where for each incorrect guess we increment i. This makes our final guess incur regret within a factor two of the correct guess. Furthermore, the sum of regret from all guesses (incorrect or correct) telescopes, incurring another factor two.

An online algorithm playing this game learns the function's value at points of its choosing, which restricts the function's value at other points due to its left-Lipschitz continuity. Not only do we know that f cannot increase too quickly as we move to the left, we also know that f cannot decrease too quickly as we move to the right.

Suppose then that we know the value of f at two points $\ell, r \in [0, 1]$ with $\ell < r$. Then for any $x \in [\ell, r]$, we can deduce the following from left-Lipschitz continuity:

$$f(r) + (r - x) \geq f(x) \geq f(\ell) - (x - \ell).$$

In other words, the value of f is sandwiched between $[f(\ell) - \ell] - x$ and $[f(r) + r] - x$. This restricts f to a parallelogram-shaped region of space. It will be useful for our algorithm to work with such parallelograms, so we define some additional notation for them.

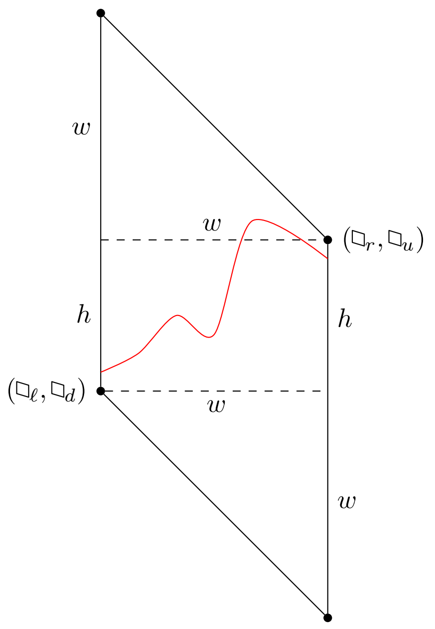
w
w
$(\varparallelogram_r, \varparallelogram_u)$
h
h
$(\varparallelogram_\ell, \varparallelogram_d)$
w
w

Figure 1. Anatomy of $\varparallelogram = \text{PARALLELOGRAM}(\ell, r, d, u)$. The function f restricted to $(\varparallelogram_\ell, \varparallelogram_r]$ must remain inside the parallelogram. One possible consistent f is depicted as a red curve.

Our analysis above assumed we knew the exact values of $f(\ell)$ and $f(r)$, but actually it suffices to have a lower bound on $f(\ell)$ and an upper bound on $f(r)$. Suppose we had bounds $d \leq f(\ell)$ and $f(r) \leq u$ (short for "down" and "up"). Then for $x \in [\ell, r]$ we know that $f(x) \in [d - \ell - x, u + r - x]$. This leads to the following definition.

Definition 2.3. $\text{PARALLELOGRAM}(\ell, r, d, u)$ represents the interval $(\ell, r]$[1] along with the guarantees $d \leq f(\ell)$ and $f(r) \leq u$ (and less importantly $\ell \leq r$). We use the symbol $\varparallelogram$ to represent such a parallelogram, and we can index such a variable using $\varparallelogram_\ell, \varparallelogram_r, \varparallelogram_d, \varparallelogram_u$ to refer to its ℓ, r, d, u values respectively. The "width" of parallelogram $\varparallelogram$ is $\varparallelogram_w \triangleq \varparallelogram_r - \varparallelogram_\ell$ and the "height" is $\varparallelogram_h \triangleq \varparallelogram_u - \varparallelogram_d$.

In our algorithm, $\varparallelogram_d$ will be the value of some point we have queried previously whereas $\varparallelogram_u$ will be the value of the *best* point we have queried previously, so we will know that $\varparallelogram_d \leq \varparallelogram_u$ and hence heights $\varparallelogram_h$ will be nonnegative.

Figure 1 depicts such a parallelogram and one possible function consistent with its guarantees.

Our additional definitions of width and height help us characterize how large a parallelogram is. Width equals the size of the underlying interval $(\varparallelogram_\ell, \varparallelogram_r]$ and also bounds how much better the best value f attains in this interval can

[1]We use half-closed intervals so that all intervals under consideration are disjoint, but this is just to help the proof avoid unnecessary edge cases.

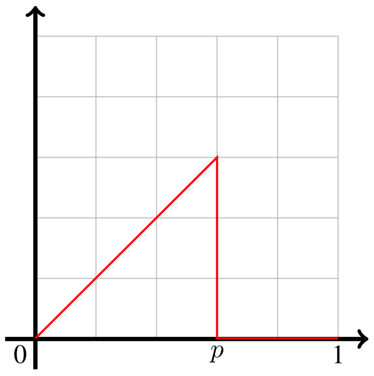

Figure 2. Revenue function f_p induced by the IDENTICALBUYERSPOSTEDPRICE problem. The function is piece-wise linear, with the first piece having a slope of one and then falling at p to the second piece, which has a slope of zero. As a result, this function is 1-right-Lipschitz for any value of p.

be compared to the previously best seen point (which has value $▱_u$). On the other hand, height quantifies how risky it is to make queries in this interval. Intuitively, wherever we choose to query in this interval, we risk the function value being $▱_h$ lower than the previously best seen point, incurring at least that much regret.

3. An $\Omega(\log\log T)$ Lower Bound

Kleinberg and Leighton studied an IDENTICALBUYERSPOSTEDPRICE problem, which is as follows (some notation changed from the original paper to match our general problem) (Kleinberg & Leighton, 2003). We have an infinite supply of identical goods, and face a sequence of T identical buyers that have the same valuation $p \in [0, 1]$ for these goods. We know T but not p. In each round t our (online) algorithm can choose a price $x^{(t)} \in [0, 1]$. If our price is lower than the valuation ($x^{(t)} \leq p$), then we sell the item for $x^{(t)}$ revenue. Otherwise, we do not sell the item and generate zero revenue. The goal is to minimize the total regret, defined to be the difference between the revenue the algorithm generated and the revenue from repeatedly picking p.

One convenient way for us to view this problem is to define a function that expresses how much revenue we generate when we choose a price x.

$$f_p(x) = \begin{cases} x & \text{if } x \leq p \\ 0 & \text{otherwise} \end{cases}$$

This function is plotted in Figure 2. Observe that this function is 1-right-Lipschitz and always outputs a point in $[0, 1]$. In other words, we can use an algorithm for LEFTLIPSCHITZMAXIMIZATION to solve this problem. The reduction is as follows. The secret function is $g(x) \triangleq f_p(1 - x)$. Whenever our LEFTLIPSCHITZMAXIMIZATION algorithm queries point $x^{(t)}$, we choose the price $1 - x^{(t)}$. If the item sells, then we report a function value of $(1 - x^{(t)})$. If the item does not sell, then we report a function value of zero.

Theorem 3.1 (c.f. Theorems 1.1, 2.1, 2.2 from (Kleinberg & Leighton, 2003))**.** *There is a deterministic algorithm that solves the* IDENTICALBUYERSPOSTEDPRICE *problem with at most $O(\log\log T)$ regret. When p is uniformly sampled from $[0, 1]$, any (i.e. possibly randomized) algorithm that solves the* IDENTICALBUYERSPOSTEDPRICE *problem incurs at least $\Omega(\log\log T)$ regret.*

By the reduction argument above, this immediately implies a lower bound for our problem.

Corollary 3.2. *There is a distribution over L-left-Lipschitz functions such that any algorithm that solves the* LEFTLIPSCHITZMAXIMIZATION *problem incurs at least $\Omega(\log\log T)$ regret.*

If we hope to match this lower bound for our problem, it will be instructive to discuss how Kleinberg and Leighton produced a matching upper bound for their problem. The key idea is as follows (again, some notation changed from the original paper to match). At the beginning of a phase, the algorithm has narrowed down the price p to somewhere in the range $[\ell, r]$ of width $w \triangleq r - \ell$. The algorithm divides this interval into $1/w$ equally-sized subintervals of width w^2: $[\ell, \ell + w^2], [\ell + w^2, \ell + 2w^2], ..., [r - w^2, r]$. It does so by querying the breakpoints from left-to-right (i.e. $\ell + w^2, \ell + 2w^2, ...$) and stops immediately when the item does not sell. There are at most $1/w$ prices that sell the item, each incurring at most w regret, for a total of $O(1)$ regret. There is at most one price that does not sell the item, incurring at most $O(1)$ regret. Since each phase refines the width from w to w^2, we only need $\log\log T$ phases before we reach width $1/T$. When the width is $1/T$, we can safely expend all remaining queries and incur at most $O(1)$ additional regret.

This strategy will be useful when considering parallelograms that have more height than width. For such parallelograms, any query risks incurring a large amount of regret, akin to not selling the item. To mitigate this risk, we will also make multiple queries; note that our queries will need to be right-to-left instead of left-to-right since we flipped the domain to interchange right-Lipschitz with left-Lipschitz.

4. Algorithmic Warmup

In this section, we give two more technical ideas that will go into our final algorithm. Similar to how considering the IDENTICALBUYERSPOSTEDPRICE problem in Section 3 gave us intuition about how to handle tall parallelograms, we will be considering other restricted versions of

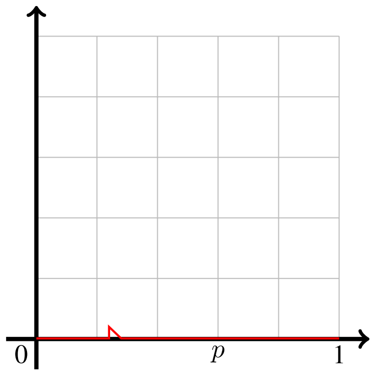

Figure 3. A spike function f hides a small spike of size $1/\sqrt{T}$ but is otherwise zero everywhere. It is 1-left-Lipschitz.

the LEFTLIPSCHITZMAXIMIZATION problem to build further intuition.

4.1. Binary Search

The first warmup exercise is to consider a function f which is zero except for a small spike of size $1/\sqrt{T}$. This function is depicted in Figure 3. In particular, this spike makes the maximum value of f equal to $1/\sqrt{T}$, so we must find a point on it if we want to do better than $\Theta(\sqrt{T})$ regret. Since querying a point on this spike is a prerequisite to achieving low regret, let us consider just the regret incurred in the process of finding a point on the spike.

First off, observe that KL search does not quite work for this situation because the hidden spike is not guaranteed to begin at the boundary of the search space (we reversed left and right, so specifically the boundary $x = 1$). One consequence of this is that if we have made several queries that all returned zero function value, the set of possible maximizers looks like a disjoint union of intervals and not a single contiguous interval.

Our queries need to roughly look like a $1/\sqrt{T}$-net before we expect to find a nonzero function value and hence the hidden spike. One possible query pattern that avoids overengineering to the spike's width being $1/\sqrt{T}$ is to perform a binary search that does not discard either half. This search process might proceed as follows. We initially know that the spike is somewhere in the range $[0, 1]$ and first query $1/2$. If we do not find the spike, we break our original range into two ranges $[0, 1/2]$ and $[1/2, 1]$. We next query $1/4$ and $3/4$, further refining the ranges into $[0, 1/4]$, $[1/4, 1/2]$, $[1/2, 3/4]$, and $[3/4, 1]$. If we divide this search into phases, with a single phase refining all ranges into two equal subranges, then in $O(\log T)$ phases our ranges will be small enough to guarantee we have found the spike. The key question is: how much regret does this incur? In particular, does it incur less than $O(\log \log T)$ regret?

Algorithm 1 Sweep Using Maximum Value Hint

```
1: Initalize $x \leftarrow 1$.
2: loop
3:     Query $f(x)$.
4:     Update $x \leftarrow x - [(\max f) - f(x)]$.
5: end loop
```

In fact, if we are careful with our argument, we can arrive at a $O(1)$ regret bound. In the phases up to phase $\log \left[1/\sqrt{T}\right]$, we have queried exactly a $1/\sqrt{T}$-net, so we have made a total of $\sqrt{T}$ queries. Each query found a value of (at least) zero and incurred (at most) $1/\sqrt{T}$ regret. This adds up to total regret $O(1)$.

Contrast this with the following more naive analysis attempt. In the first phase, we query one point, $1/2$, and try to bound how much regret it could incur only using knowledge we have after the first phase. Since the ranges after this phase are $[0, 1/2]$ and $[1/2, 1]$, it is plausible that the function actually has a spike of size $1/2$ and we incurred $1/2$ regret, for a total of $1/2$ regret in the first phase. Repeating this reasoning for the second phase, we query two more points, have four intervals, risk a spike of size $1/4$, and we risked incurring $2 \cdot 1/4 = 1/2$ total regret in this phase. If we continue like this, each phase will result in constant regret; we have a logarithmic number of phases producing a bound of $O(\log T)$ overall regret. This is too much for our $O(\log \log T)$ goal!

The upshot is that we must be careful not to be too eager when determining how much regret a particular query incurs. The difference between these two analyses was that the precise one used the final value of the function maximizer, while the loose one used a bound we had on the value of the function maximizer that only took into account the queries/results we had made so far. In our final algorithm, our analysis will sometimes need to only pay for the difference between the best seen point so far and the queried point, making additional payments as the best point seen so far improves.

4.2. Maximum Value Hint

The second warmup exercise is to suppose our algorithm is told the maximium value of f, $(\max f)$, but not where it is. It turns out this hint is quite powerful, since it lets us cut a log log factor from the regret:

Theorem 4.1. *There is an online algorithm that solves the* LEFTLIPSCHITZMAXIMIZATION *when additionally told* $(\max f)$ *with at most* $O(1)$ *regret.*

We defer the full proof to Appendix B, but the algorithm that does so is Algorithm 1. The main proof idea is that we know the maximizer is in the region $[0, x]$, and this region shrinks by exactly the amount of regret we incur.

Remark 4.2. Although our algorithm here is tailored to knowing the maximum value precisely, there is a more general idea to be found. If we ever query a point and it is lower than the value of the best point we have seen so far, we can immediately eliminate a region to its left. The size of this region is equal to the amount that our point was below the best point we have seen so far. This will let us charge the immediately incurred regret to elimination of feasible region. There is some additional regret incurred, namely the difference between the actual maximum value and the value of the best point seen so far, but this will fall under the binary search technique from Subsection 4.1.

5. A Hybrid Algorithm

In this section, we explain how to synthesize the technical ideas from Sections 3 and 4 to prove our main theorem.

Theorem 5.1. *There is an online algorithm that solves the* LeftLipschitzMaximization *problem with at most* $O(\log\log T)$ *regret.*

We plan to show that Algorithm 2 incurs at most $O(\log\log T)$ regret for 1-left-Lipschitz functions.

Since Algorithm 2 is complicated, here is a quick summary of what it is doing. Initially, query endpoints $f(0)$ and $f(1)$ to constuct a starting parallelogram. This is added to a set of parallelograms that tracks all intervals that might have a maximizer. The algorithm then repeatedly removes the widest parallelogram from this set and either performs a binary search step or a KL search step on it, depending on whether it is wider (binary search) or taller (KL search). During this entire process, the algorithm also tracks the best point seen and where it occurred. When it sees a new best point, the algorithm also needs to adjust all parallelograms in its set by increasing their height and decreasing their width.

We prove the regret bound for our algorithm using a potential argument. In each iteration i of the algorithm's while loop, the algorithm modifies one or more parallelograms from S. We define a potential function ϕ on S and will show that from round-to-round, this potential function on the state of S is monotone nonincreasing and the total regret that the algorithm incurs can be charged to the total drop in potential.

Our function ϕ is defined on parallelograms, and the potential of S is just the sum of potentials of all parallelograms it contains. We steal some notation from our algorithm, namely that the width of parallelogram ⏢ is $w \triangleq ⏢_r - ⏢_\ell$ and its height is $h \triangleq ⏢_u - ⏢_d$. With these definitions, the potential of one parallelogram ⏢ is:

$$\phi(⏢) \triangleq \phi_{binary}(⏢) + \phi_{KL}(⏢)$$

$$\phi_{binary}(⏢) \triangleq ⏢_w$$

$$\phi_{KL}(⏢) \triangleq -⏢_h \log\log\left[\min\left(2⏢_h/⏢_w + 2, (2T+2)^2\right)\right]$$

Our potential function is the sum of two potential subfunctions, ϕ_{binary} and ϕ_{KL}. The former function is nonnegative and will be used to execute the ideas from Section 4. The latter is nonpositive (our logs are base two, so $\log\log 2 = 0$) and will be used to execute the ideas from Section 3.

We abuse notation slightly and allow these functions to accept the entire set of parallelograms S as input:

$$\phi(S) \triangleq \sum_{⏢\in S} \phi(⏢)$$

$$\phi_{binary}(S) \triangleq \sum_{⏢\in S} \phi_{binary}(⏢)$$

$$\phi_{KL}(S) \triangleq \sum_{⏢\in S} \phi_{KL}(⏢)$$

Theorem 5.1 follows directly from combining the following three technical theorems.

Theorem 5.2. *When evaluated on the state of S before/after iterations of Algorithm 2's while loop on line 9, both $\phi_{binary}(S)$ and $\phi_{KL}(s)$ are monotone nonincreasing, so $\phi(S)$ is as well.*

Theorem 5.3. *The initial potential satisfies $\phi(S) \le O(1)$ and the final potential satisfies $\phi(S) \ge -O(\log\log T)$.*

Theorem 5.4. *The total regret of Algorithm 2 is at most a constant times the total drop in the potential function $\phi(S)$ plus a constant.*

Theorems 5.2 and 5.3 together inform us that the sum of all drops of the potential function is $O(1) + O(\log\log T) = O(\log\log T)$ in magnitude. Theorem 5.4 lets us charge our regret to these potential drops, meaning our regret is also $O(\log\log T)$ as desired.

Due to space constraints, the proofs of these three technical theorems can be found in Appendix C. In their place, we include some quick proof sketches here.

Proof Sketch of Theorem 5.2. The monotonicity of $\phi_{binary}(S)$, the sum of widths of parallelograms, is due to our algorithm only dividing up parallelogram intervals into smaller parallelogram intervals. The monotonicity of ϕ_{KL} is due to each iteration of the main while loop of our algorithm producing a "final" parallelogram (on one of lines 21, 26, 39) that has at least as much height as the original parallelogram ⏢. $\square$

Proof Sketch of Theorem 5.3. The initial potential is bounded above because ϕ_{binary} is at most the total width,

Algorithm 2 Hybrid Binary and KL Search

```
1: Query $f(0)$ and $f(1)$.
2: if $f(0) < f(1)$ then
3:     Initialize our search set $S \leftarrow \{\textsc{Parallelogram}(\ell = 0, r = 1, d = f(0), u = f(1))\}$.
4:     Initialize our best $f$-value found so far $b \leftarrow f(1)$ and where it was found $x \leftarrow 1$.
5: else
6:     Initialize our search set $S \leftarrow \{\textsc{Parallelogram}(\ell = 0, r = 1 - (f(0) - f(1)), d = f(0), u = f(0))\}$.
7:     Initialize our best $f$-value found so far $b \leftarrow f(0)$ and where it was found: $x \leftarrow 0$.
8: end if
9: while $S$ contains a $\parallelogram$ with $\parallelogram_w > \frac{1}{T}$ do
10:     Let $\parallelogram$ be some parallelogram in $S$ of maximum width; remove $\parallelogram$ from $S$.
11:     if $\parallelogram_w \geq \parallelogram_h$ then {binary search case, see Subsection 4.1 for intuition}
12:         We make at most $\hat{q} \leftarrow 1$ query to subdivide $(\parallelogram_\ell, \parallelogram_r]$.
13:     else {KL case, see Section 3 for intuition}
14:         We make at most $\hat{q} \leftarrow \lfloor 2\parallelogram_h/\parallelogram_w + 4 + \parallelogram_w/\parallelogram_h \rfloor$ queries to subdivide $(\parallelogram_\ell, \parallelogram_r]$.
15:     end if
16:     Our queries will be at least $\delta_q \triangleq \frac{\parallelogram_w}{\hat{q}+1}$ apart.
17:     We use $m^i$ to denote the $i^{th}$ middle point that we query and $\parallelogram^i$ to denote the $i^{th}$ parallelogram we add to $S$; the
        first query point is $m^1 \leftarrow \parallelogram_r - \delta_q$ and the first parallelogram is $\parallelogram^1 \leftarrow \textsc{Parallelogram}(\ell = m_1, r = \parallelogram_r, d =$
        $null, u = b)$ {We fill in the $d$-value later.}
18:     for $i = 1, 2, ..., \hat{q}$ do
19:         Query $f(m_i)$.
20:         if $f(m_i) + (m_i - \parallelogram_\ell) < b$ then {entire remaining interval $(\parallelogram_\ell, m_i]$ cannot beat $b$}
21:             Update $\parallelogram^i_d \leftarrow \min\{f(m_i), \parallelogram_d\}$ and insert the final $\parallelogram^i$ into $S$ and break out of the enclosing for loop.
22:         end if
23:         Update $\parallelogram^i_d \leftarrow f(m_i)$ and insert $\parallelogram^i$ into $S$.
24:         if $f(m_i) \leq b$ then {the interval $(m_i - (b - f(m_i)), m_i]$ cannot beat $b$}
25:             The next query point is $m_{i+1} \leftarrow m_i - (b - f(m_i)) - \delta_q$.
26:             If the next query point is not inside our original interval (i.e. $m_{i+1} \leq \parallelogram_\ell$), we add the final
                $\textsc{Parallelogram}(\ell = \parallelogram_\ell, r = m_i - (b - f(m_i), d = \parallelogram_d, u = b)$ to $S$ and break out of the enclosing
                for loop.
27:             Otherwise, the next parallelogram is $\parallelogram^{i+1} \leftarrow \textsc{Parallelogram}(\ell = m_{i+1}, r = m_i - (b - f(m_i), d =$
                $null, u = b)$.
28:         else {just witnessed a new best $f$-value}
29:             Compute the improvement in $f$-value $\delta_f = f(m_i) - b$.
30:             Update the best $f$-value found so far $b \leftarrow f(m_i)$ and where it was found $x \leftarrow m_i$.
31:             for each parallelogram $\parallelogram'$ in $S$ do
32:                 if $\parallelogram'_\ell \leq \parallelogram'_r - \delta_f$ then
33:                     Update $\parallelogram'$ in $S$ to $\textsc{Parallelogram}(\ell = \parallelogram'_\ell, r = \parallelogram'_r - \delta_f, d = \parallelogram'_d, u = b)$.
34:                 else {update to an unused width-zero parallelogram for proof-accounting}
35:                     Update $\parallelogram'$ in $S$ to $\textsc{Parallelogram}(\ell = \parallelogram'_\ell, r = \parallelogram'_\ell, d = \parallelogram'_d, u = \parallelogram'_u + (\parallelogram'_r - \parallelogram'_\ell))$.
36:                 end if
37:             end for
38:             The next query point is $m_{i+1} \triangleq m_i - \delta_q$.
39:             If the next query point is not inside our original interval (i.e. $m_{i+1} \leq \parallelogram_\ell$), we add the final
                $\textsc{Parallelogram}(\ell = \parallelogram_\ell, r = m_i, d = \parallelogram_d, u = b)$ to $S$ and break out of the enclosing for loop Other-
                wise, the next parallelogram is $\parallelogram^{i+1} \leftarrow \textsc{Parallelogram}(\ell = m_{i+1}, r = m_i, d = null, u = b)$.
40:         end if
41:     end for
42: end while
43: Repeatedly query the best $f$-value found so far for any remaining rounds, i.e. repeatedly query $f(x)$.
```

which is initially one, and ϕ_{KL} is nonpositive. The final potential is bounded below because ϕ_{binary} is nonnegative and ϕ_{KL} is at most $-O(\log\log T)$ times the sum of heights; an additional charging argument shows that the sum of heights is bounded by a constant. □

Proof Sketch of Theorem 5.4. We execute a semi-eager, semi-lazy charging argument. Our two key subcases come from algorithm lines 11 and 13, which correspond to the binary search from Subsection 4.1 and the KL search from Section 3, respectively. For binary search, we know we need to do lazy charging, and we can charge any immediate regret incurred from a suboptimal query value to loss in total width (ϕ_{binary}). For KL search, we use eager charging, and no matter whether all planned queries execute or we fail-fast in the middle, we can show we incur at most $O(\triangleright_h)$ regret, which can be charged to ϕ_{KL} due to the log log factor increasing by one. □

References

Alon, T., Dütting, P., and Talgam-Cohen, I. Contracts with private cost per unit-of-effort. In *EC*, pp. 52–69, 2021.

Carroll, G. Robustness and linear contracts. *Am. Econ. Rev.*, 105(2):536–563, 2015.

Castiglioni, M., Marchesi, A., and Gatti, N. Bayesian agency: Linear versus tractable contracts. In *EC*, pp. 285–286, 2021.

Castiglioni, M., Marchesi, A., and Gatti, N. Designing menus of contracts efficiently: The power of randomization. In *EC*, pp. 705–735, 2022.

Cesa-Bianchi, N., Gaillard, P., Gentile, C., and Gerchinovitz, S. Algorithmic chaining and the role of partial feedback in online nonparametric learning. In *COLT*, pp. 465–481, 2017.

Cohen, M. C., Lobel, I., and Paes Leme, R. Feature-based dynamic pricing. In *EC*, pp. 817–817, 2016.

Dütting, P., Roughgarden, T., and Talgam-Cohen, I. Simple versus optimal contracts. In *EC*, pp. 369–387, 2019.

Dütting, P., Roughgarden, T., and Talgam-Cohen, I. The complexity of contracts. In *SODA*, pp. 2688–2707, 2020.

Dütting, P., Ezra, T., Feldman, M., and Kesselheim, T. Combinatorial contracts. In *FOCS*, pp. 815–826, 2021.

Dütting, P., Ezra, T., Feldman, M., and Kesselheim, T. Multi-agent contracts. In *STOC*, pp. 1311–1324, 2023.

Emamjomeh-Zadeh, E., Paes Leme, R., Schneider, J., and Sivan, B. Jointly learning prices and product features. In *IJCAI*, pp. 2360–2366, 2021.

Goldberg, A. V., Hartline, J. D., and Wright, A. Competitive auctions and digital goods. In *SODA*, pp. 735–744, 2001.

Grossman, S. J. and Hart, O. D. An analysis of the principal-agent problem. *Econometrica*, 51:7–45, 1983.

Guruganesh, G., Schneider, J., and Wang, J. R. Contracts under moral hazard and adverse selection. In *EC*, pp. 563–582, 2021.

Ho, C.-J., Slivkins, A., and Vaughan, J. W. Adaptive contract design for crowdsourcing markets: Bandit algorithms for repeated principal-agent problems. In *EC*, pp. 359–376, 2014.

Holmström, B. Moral hazard and observability. *Bell J. Econ.*, 10:74–91, 1979.

Kleinberg, R. and Leighton, T. The value of knowing a demand curve: Bounds on regret for online posted-price auctions. In *FOCS*, pp. 594–605, 2003.

Liu, A., Paes Leme, R., and Schneider, J. Optimal contextual pricing and extensions. In *SODA*, pp. 1059–1078, 2021.

Lobel, I., Paes Leme, R., and Vladu, A. Multidimensional binary search for contextual decision-making. *Oper. Res.*, 2017.

Mao, J., Paes Leme, R., and Schneider, J. Contextual pricing for lipschitz buyers. *NeurIPS*, pp. 5648–5656, 2018.

Paes Leme, R. and Schneider, J. Contextual search via intrinsic volumes. In *FOCS*, pp. 268–282, 2018.

Slivkins, A. Contextual bandits with similarity information. *J. Mach. Learn. Res.*, 15:2533–2568, 2014.

Zhu, B., Bates, S., Yang, Z., Wang, Y., Jiao, J., and Jordan, M. I. The sample complexity of online contract design. *CoRR*, abs/2211.05732, 2022.

A. Linear Contract Profit is One-Sided Lipschitz

In this appendix, we consider the application of learning the optimal linear contract. After defining some necessary contract theory notation, we will prove that the principal's utility, as a function of the linear contract they choose, is one-sided Lipschitz.

In the basic hidden-action principal-agent problem, an agent chooses between n different actions, resulting in one of m different outcomes. Action i incurs a cost of c_i to the agent, has probablity $F_{i,j}$ of resulting in outcome j, which would result in a reward of r_j for the principal. To incentivize the agent, the principal may present a contract where they promise to transfer x_j in the event of outcome j. Faced with such a contract, the agent chooses an action i[2] that maximizes their utility, $\sum_j F_{i,j}x_j - c_i$. The principal makes a net profit of $\sum_j F_{i,j}(r_j - x_j)$ (also referred to as the principal's utility).

A contract is linear if it can be written as $x_j = \alpha r_j$ for all outcomes j and some $\alpha \in [0, 1]$. We are now ready to state and prove our desired result.

Lemma A.1. *$U_{principal}(\alpha)$ is L-left-Lipschitz continuous, where the Lipschitz constant is $L = \max_i \sum_j F_{i,j}r_j$, i.e. the maximum expected reward of any action.*

Proof. The first step is to use the definition of a linear contract to simplify the agent and principal utility functions:

$$\begin{aligned}
U_{agent}(\alpha) &= \max_i \left[\sum_j F_{i,j}\alpha r_j\right] - c_i \\
&= \max_i \alpha \left[\sum_j F_{i,j}r_j\right] - c_i \\
U_{principal}(\alpha) &= \sum_j F_{i_\star,j}(r_j - \alpha r_j) && (i_\star \text{ is the agent chosen action}) \\
&= (1-\alpha)\left[\sum_j F_{i_\star,j}r_j\right]
\end{aligned}$$

This highlights $\left[\sum_j F_{i,j}r_j\right]$ as a key term for action i. This is equal to the expected reward the principal gets when action i is played.

As we increase α, U_{agent} jumps to actions with higher and higher expected reward. As a result, $U_{principal}$ sometimes jumps up because it has this expected reward as a factor, and otherwise steadily decreases due to the $(1-\alpha)$ factor. This is enough to show that $U_{principal}(\alpha)$ is $\left[\max_i \sum_j F_{i,j}r_j\right]$-left-Lipschitz continuous. More formally, consider two linear contracts $\alpha \le \alpha'$.

$$\begin{aligned}
U_{principal}(\alpha) - U_{principal}(\alpha') &= (1-\alpha)\left[\sum_j F_{i_\star,j}r_j\right] - (1-\alpha')\left[\sum_j F_{i'_\star,j}r_j\right] \\
&\le (1-\alpha)\left[\sum_j F_{i_\star,j}r_j\right] - (1-\alpha')\left[\sum_j F_{i_\star,j}r_j\right] && (i'_\star \text{ has higher expected reward}) \\
&\le \left[\sum_j F_{i_\star,j}r_j\right](\alpha' - \alpha)
\end{aligned}$$

[2]It is standard to tiebreak for actions that generate more profit for the principal, since giving the agent an ϵ fraction of the reward would induce that behavior.

$$\leq \underbrace{\left[\max_i \sum_j F_{i,j} r_j\right]}_{\text{Lipschitz constant}} (\alpha' - \alpha)$$

This completes the proof of Lemma A.1. □

B. Missing proofs from Section 4

In this appendix, we provide the missing proof of Theorem 4.1, which is restated for convenience.

Theorem 4.1. *There is an online algorithm that solves the* LEFTLIPSCHITZMAXIMIZATION *when additionally told* $(\max f)$ *with at most* $O(1)$ *regret.*

Proof. We will show that Algorithm 1 incurs at most $O(1)$ regret for 1-left-Lipschitz functions. This proves the result via scaling f appropriately.

We maintain the invariant that the algorithm knows $(\max f)$ is attained somewhere in the interval $[0, x]$. Initially, $x = 1$ and so this invariant is trivially true due to the domain of f being $[0, 1]$.

Next, consider some iteration of the loop with some initial value of x. We query $f(x)$ and update to $x' \triangleq x - [(\max f) - f(x)]$. Consider some point $z \in (x', x]$, and consider what 1-left-Lipschitz continuity implies about this point.

$$\begin{aligned}
z &\leq x \\
f(z) - f(x) &\leq x - z && \text{(left-Lipschitz)} \\
f(z) &\leq f(x) + x - z \\
&< f(x) + (\max f) - f(x) && (x' < z) \\
&= (\max f)
\end{aligned}$$

Hence from the query result we know that $(\max f)$ is not attained in the interval $(x', x]$. Since we knew it was attained in $[0, x]$ by our inductive hypothesis this implies it is attained in $[0, x']$, completing the induction.

We finish up the proof from here using a potential argument. We define our potential function to be $\phi(x) = x$. Initially this potential is one. Each round, the potential goes down by $x - x'$ and we incur regret $(\max f) - f(x)$. A slight rearranging of the update rule for x shows the change in potential and the regret are equal. Since the potential is always nonnegative (the max must be attained somewhere so the interval cannot become empty), the total error is bounded by one. This completes the proof. □

C. Missing Proofs from Section 5

In this appendix, we provide the missing proofs of Theorems 5.2, 5.3, and 5.4 from Section 5. Each theorem is restated before its corresponding proof for convenience.

Theorem 5.2. *When evaluated on the state of* S *before/after iterations of Algorithm 2's while loop on line 9, both* $\phi_{binary}(S)$ *and* $\phi_{KL}(s)$ *are monotone nonincreasing, so* $\phi(S)$ *is as well.*

Proof. We first show that our potential function is monotone between iterations of our algorithm's main while loop, found on line 9. To do so, we observe that the algorithm edits the parallelograms in S for two reasons: (i) the algorithm removes a parallelogram from S at the start of the iteration and makes queries to divide its interval into smaller disjoint intervals that it inserts back into S and (ii) when the algorithm witnesses a new best f-value, it updates all non-zero-width parallelograms in S so that their u-values equal that best value b. We will argue about the effects of these two reasons separately, and we also argue separately about ϕ_{binary} and ϕ_{KL}, showing that each individually is monotone nonincreasing.

The simpler subfunction is ϕ_{binary}. For the edits due to reason (i), we update a parallelogram that covers $(▱_\ell, ▱_r]$ with parallelograms that cover disjoint subintervals, so the total width is nonincreasing. For the edits due to reason (ii), we only shrink the widths of paralellograms in S, so the total width is again nonincreasing. As a result, ϕ_{binary} is nonincreasing over iterations of our algorithm.

The more complex subfunction is ϕ_{KL}. For the edits due to reason (i), we claim that the iteration that removes a parallelogram $\square$ always inserts another parallelogram that has lower width and at least as much height. In particular, this is the final parallelogram inserted by this iteration, which occurs on one of algorithm lines 21, 26, or 39. Importantly, we always trigger such a final parallelogram, because we subtract δ_q once before the for loop begins and $\hat{q}$ additional times during the for loop if it keeps running, for a total subtraction of $[\hat{q}+1]\delta_q = \square_w$. Observe all final parallelogram possibilities have a d-value of at most $\square_d$ and a u-value of b which is at least $\square_u$ because b is nondecreasing over the algorithm, so they have height at least the original parallelogram's height. They trivially have less width because their corresponding interval is a subset of that of the original parallelogram. To finish this claim, all we need to observe is that ϕ_{KL} is decreasing in $\square_h$ and increasing in $\square_w$. Recall the definition of ϕ_{KL}:

$$\phi_{KL} \triangleq -\square_h \log\log\left[\min\left(2\square_h/\square_w + 2, (2T+2)^2\right)\right]$$

ϕ_{KL} is decreasing in $\square_h$ because increasing $\square_h$ increases the coefficient in front of the $\log\log$ and possibly the quantity inside the $\log\log$ (keep in mind there is a leading negative). It is increasing in $\square_w$ because increasing $\square_w$ possibly decreases the quantity inside the $\log\log$ (again keep in mind there is a leading negative). Finally, although reason (i) may introduce additional non-final parallelograms, ϕ_{KL} is nonpositive so this can only decrease the total potential further.

Next, we consider edits due to reason (ii). It is simpler to see that each parallelogram updated to a parallelogram that has less width and more height. As we have already argued for reason (i), this only decreases their potential value under ϕ_{KL}.

We have now shown that $\phi(S)$ is monotone nonincreasing between iterations. $\square$

Theorem 5.3. *The initial potential satisfies $\phi(S) \leq O(1)$ and the final potential satisfies $\phi(S) \geq -O(\log\log T)$.*

Proof. To understand the range of ϕ, we now need to upper-bound its initial value and lower-bound its final value. The bound on initial value comes from observing the state of the initial parallelogram. The potential of the initial parallelogram is at most $O(1)$, since its width is at most one and ϕ_{KL} is nonpositive. The bound on the final value is a little trickier, but we will show that ϕ_{KL} is at least $-O(\log\log T)$ and note that ϕ_{binary} is nonnegative.

Observe that due to the min in ϕ_{KL}:

$$\begin{aligned}
&\sum_{\square\in S} \phi_{KL}(\square) \\
&= \sum_{\square\in S} \left[-\square_h \log\log\left[\min\left(2\square_h/\square_w + 2, (2T+2)^2\right)\right]\right] \\
&\geq -\sum_{\square\in S} \square_h \log\log\left[(2T+2)^2\right] \\
&\geq -\sum_{\square\in S} \square_h \cdot O(\log\log T) \\
&\geq -O(\log\log T) \cdot \sum_{\square\in S} \square_h
\end{aligned}$$

We want to show that $\sum_{\square\in S} \square_h$ is at most $O(1)$ to complete this part of the proof. The height of the initial parallelogram is indeed $O(1)$, since by assumption f has range $[0,1]$. How much can the total height grow over the duration of the algorithm? We claim that increases to the total height can be charged to either decreases to the total width or increases to the best f-value found so far (i.e. b). Since total width can drop by at most $O(1)$ and b can increase by at most $O(1)$ due to the range of f being $[0,1]$, this claim implies[3] the desired lower bound on the final value of ϕ_{KL}. To prove this claim,

[3]Interestingly, b actually can only increase by the size of the f's range less the height of the original rectangle, so a more precise bound on final total height is domain size plus range size. This is also the point in the analysis where the dependence on the larger of range size and Lipschitz constant emerges. This is easiest to see when considering the size of the range after we have scaled to achieve unit domain size and unit Lipschitz constant. If we shrink the range size below one, then the unit Lipschitz constant dominates the range size and correspondingly the unit domain size dominates the total height analysis. If we grow the range size above one, then the range size dominates the Lipschitz constant and correspondingly the range size dominates the total height analysis.

it will be useful to again divide parallelogram changes into these two reasons: (i) the algorithm removes a parallelogram from S at the start of the iteration and makes queries to divide its interval into smaller disjoint intervals that it inserts back into S and (ii) when the algorithm witnesses a new best f-value, it updates all non-zero-width parallelograms in S so that their u-values equal that best value b.

Let's consider parallelogram changes due to reason (i). Recall that we have shown that the height of the original parallelogram is primarily inherited by the final parallelogram inserted in this iteration. The final parallelogram's height can be a little larger, though, due to having a lower d-value and/or having a higher u-value. The only way to get a lower d-value is if we query a bad point ($f(m_i) + (m_i - ▱_\ell) < b$) and skip over the entire remaining interval; this makes the algorithm set $▱^i_d \leftarrow \min\{f(m_i), ▱_d\}$. For this case:

$$\begin{aligned} ▱^i_d &= \min\{f(m_i), ▱_d\} \\ &\geq f(m_i) \\ &\geq f(▱_\ell) - (m_i - ▱_\ell) && \text{(Lipschitzness)} \end{aligned}$$

In other words, the final parallelogram can only have a lower d-value than the original parallelogram by the width of the interval skipped over, $(▱_\ell, m_i]$. If this does not occur, then the final parallelogram receives a d-value of $▱_d$. The final parallelogram can also have a higher u-value. Its u-value is set to b, which may be higher than $▱_u$. We charge this against the increase to b (indeed, this is the only thing we charge to increases to b).

That just leaves all the non-final parallelograms inserted by reason (i). Since they are nonfinal, we know that at the time of insertion into S, their d-values are $▱^i_d \leftarrow f(m_i)$ and their u-values are b. If such a parallelogram has nonzero height, then right after insertion we wind up in the $f(m_i) \leq b$ case and proceed to skip exactly $b - f(m_i)$ in width before making the next query point.

We now move on to reason parallelogram updates made by reason (ii). It is straightfoward to observe that these updates deduct exactly as much width as they increase height. This completes the proof. □

Theorem 5.4. *The total regret of Algorithm 2 is at most a constant times the total drop in the potential function $\phi(S)$ plus a constant.*

Proof. Before S is initialized and our potential is defined, our algorithm queries $f(0)$ and $f(1)$, incurring at most $O(1)$ regret since by assumption the range of f is $[0, 1]$. After the while loop finishes and we repeatedly query $f(x)$, we know that any f-value that is higher than b must occur inside the interval corresponding to some parallelogram in S. We also know that the width of every parallelogram in S is at most $\frac{1}{T}$ and that the u-value of every parallelogram in S is at most b, so the optimal f-value is at most $b + \frac{1}{T}$ due to Lipschitzness. Since we make at most T queries, this incurs at most $\frac{1}{T} \cdot T = O(1)$ regret. Both of these $O(1)$ regret amounts are accounted for by the "plus a constant" in the theorem statement.

As Subsection 4.1 describes, it is necessary for our charging scheme to be partially lazy to handle the example there. One missing detail until now is how to pay for increases in the best seen f-value. When b increases, we can use the fact that our parallelogram update logic is shrinking widths to charge the regret we just learned about to ϕ_{binary}. There is a small issue with this idea: if a parallelogram runs out of width, then we will not have enough potential drop to charge against. As a result, we cannot lazily charge all parallelograms; we need to eagerly pay for some queries that might have been associated with small-width parallelograms upfront.

Formally, all of the while loop queries are either eagerly paid for or lazily associated with a parallelogram. For an eagerly paid for query, we charge the difference between $f(m_i)$ and the best possible f-value consistent with the queries so far. For a lazily associated query, we charge the difference between $f(m_i)$ and the best f-value seen so far (i.e. b), then associate the query with a parallelogram being inserted into S. When this parallelogram is extracted from S to be processed by the while loop, we charge the total improvement in best f-value (i.e. the sum of δ_f) to the total potential changes to this parallelogram from updates. Furthermore, this while loop must either eagerly pay for the difference between its $▱_u$ and the best possible f-value consistent with the queries so far or associate it with yet another parallelogram being inserted into S. When the while loop terminates, we must also take all the lazily associated parallelograms in S and charge their total improvement in f-value. We then pay an additional $\frac{1}{T}$ per such parallelogram to account for the fact that the best possible f-value might be $\frac{1}{T}$ better than b, but since our algorithm only makes T queries total, there are at most T such lazily associated queries for an additional uncharged $O(1)$ regret, which also goes to the "plus a constant" in the theorem statement.

Then, when this parallelogram is either removed from S to be processed by the while loop, or the while loop terminates, we charge the total improvement in best f-value (i.e. the sum of δ_f) to the total potential changes to this parallelogram. We impose the condition that a parallelogram can only be lazily associated with a query if, at the moment of its insertion into S, its width is at least a quarter of the maximum width in S.

Consider some iteration of the while loop, which extracts a parallelogram $\square$ to refine. An extracted parallelogram must have at least $\frac{1}{T}$ width, i.e. strictly positive width, so the updates performed on it while it was sitting in S did not zero its width. Since each increase in b was successfully matched by a decrease in the width of this parallelogram, we can charge the sum of b improvements to the drop in this parallelogram's ϕ_{binary}. Note that while this parallelogram actually may or may not be lazily associated with a query, our proof assumes the worst (that it is lazily associated) and explains how to perform the accounting to handle this.

Our analysis now splits into two cases, depending on how the value of $\hat{q}$ was calculated.

Case 1: $\square_w \geq \square_h$ **(**$\hat{q} \leftarrow 1$) The while loop only makes a single query, in the middle of the interval $(\square_\ell, \square_r]$.

If we trigger the $f(m_i) + (m_i - \square_\ell) < b$ condition, then the only parallelogram we insert is PARALLELOGRAM($\ell = m_i, r = \square_r, d = \min\{f(m_i), \square_d\}, u = b$). It has half the width of the original parallelogram $\square$, so we know ϕ_{binary} drops by $\frac{1}{2}\square_w$ ($\square_w$ is the original width). We will eagerly pay for our query using this drop in potential. Since $\square$ had the maximum width among all parallelograms in S when we extracted it, we know that the best possible f-value is at most $b + \square_w$.

$$\begin{aligned}
f(\square_\ell) - f(m_i) &\leq m_i - \square_\ell && \text{(left-Lipschitz)} \\
&= \frac{1}{2}\square_w \\
b + \square_w - f(m_i) &\leq b + \square_w - f(\square_\ell) + \frac{1}{2}\square_w \\
&= \frac{3}{2}\square_w + b - f(\square_\ell) \\
&= \frac{3}{2}\square_w + \square_h \\
&\leq \frac{5}{2}\square_w && (\square_w \geq \square_h)
\end{aligned}$$

Hence, we can eagerly charge our incurred regret ($b + \square_w - f(m_i)$) to the drop in ϕ_{binary} times a constant factor (namely, five). Our original rectangle might also have been lazily associated, but we can eagerly pay for it here as well; the best possible f-value is at most $b+\square_w$ so we can charge the difference of $\square_w$ to the same drop in ϕ_{binary}, increasing the constant factor (to seven).

If we do not trigger this condition, then what happens next depends on the comparison $f(m_i) \leq b$. If this condition is true, then we consider how much of the interval we skip over, i.e. $b - f(m_i)$. Since we did not trigger the $f(m_i) + (m_i - \square_\ell) < b$ condition, we know that $b - f(m_i)$ is at most $\square_w/2$. We break into two subcases, depending on how $b - f(m_i)$ compares to $\square_w/4$.

In our first subcase, $b - f(m_i) \leq \square_w/4$, i.e. we did not incur that much regret and the left parallelogram is still sizable. The plan is to lazily assign the query $f(m_i)$ to the left parallelogram and pass on the original parallelogram's lazy query to the right parallelogram. We need to pay upfront for $b - f(m_i)$, which we do by observing that this amount is missing from the width of this parallelogram and hence we can charge it to ϕ_{binary}. We then need to guarantee that each parallelogram has at least one quarter the width of the maximum width in S. This is why we extracted the maximum width parallelogram from S. The left parallelogram is half the width minus the amount skipped (which is at most a quarter of the width), so it is at least a quarter the width of the original parallelogram. The right parallelogram is half the width. Hence it is safe to lazily associate queries with these parallelograms.

In our second subcase, $b - f(m_i) \geq \square_w/4$, i.e. we incurred more regret and the left parallelogram is not that sizable. The plan is now to eagerly pay upfront for everything. Observe that at least $\square_w/4$ is missing from the width of the left parallelogram, i.e. ϕ_{binary} dropped by that much. Remember that $b - f(m_i) \leq \square_w/2$ due to skipping the first condition, and the best possible f-value is at most $b + \square_w$. We can pay for $\frac{3}{2}\square_w$ for $f(m_i)$ and $\square_w$ for the lazily associated query of the original parallelogram by charging to ϕ_{binary} times a constant factor (ten).

This completes the analysis for case 1.

Case 2: $▱_w < ▱_h$ **(**$\hat{q} \leftarrow \lfloor 2▱_h/▱_w + 4 + ▱_w/▱_h \rfloor$**)** The while loop makes multiple queries, but we know that all new parallelograms have width at most $\delta_q \triangleq \frac{▱_w}{\hat{q}+1} \leq \frac{▱_w}{2▱_h/▱_w+4+▱_w/▱_h}$. Additionally, we know that the final parallelogram has at least as much height as the original parallelogram. The drop in potential from the ϕ_{KL} of the original parallelogram to the ϕ_{KL} of the final parallelogram is hence at least:

$$\begin{aligned}
&\phi_{KL}(▱) - \phi_{KL}(▱_{final})\\
\geq& -▱_h \log\log\left[2▱_h/▱_w + 2\right]\\
&\quad + ▱_h \log\log\left[(2▱_h/▱_w)\cdot(2▱_h/▱_w + 4 + ▱_w/▱_h) + 2\right]\\
=& -▱_h \log\log\left[2▱_h/▱_w + 2\right]\\
&\quad + ▱_h \log\log\left[4▱_h^2/▱_w^2 + 8▱_h/▱_w + 4\right]\\
=& -▱_h \log\log\left[2▱_h/▱_w + 2\right]\\
&\quad + ▱_h \log\log\left[(2▱_h/▱_w + 2)^2\right]\\
=& -▱_h \log\log\left[2▱_h/▱_w + 2\right]\\
&\quad + ▱_h\left[1 + \log\log\left[2▱_h/▱_w + 2\right]\right]\\
=& ▱_h
\end{aligned}$$

Note that we were able to safely omit the min with $(2T+2)^2$ because the height of parallelogram $▱$ is at most one and the width is at least $\frac{1}{T}$ if it was extracted from S by the while-loop, implying $2▱_h/▱_w + 2 \leq 2T + 2$. As a result, our log log factor must still have the capacity to grow by one.

Our plan is to use this $▱_h$ drop in potential to eagerly pay for all queries, including the one possibly lazily associated with the original parallelogram.

Observe that $\hat{q}$ is $O(▱_h/▱_w)$, since $▱_h/▱_w \geq 1$ and $▱_w/▱_h \leq 1$. For each query that does not trigger the condition $f(m_i) + (m_i - ▱_\ell) < b$, the regret is at most $b + ▱_w - f(m_i) \leq m_i - ▱_\ell + ▱_w \leq 2▱_w$, even assuming the best possible f-value. Since $▱_h \geq ▱_w$, we can charge this to our potential drop times a constant factor (two).

We may make a single query that triggers the condition $f(m_i) + (m_i - ▱_\ell) < b$, because doing so ends the querying. However, the value of that query is limited by $▱_d$ and Lipschitzness. Formally:

$$\begin{aligned}
f(▱_\ell) - f(m_i) &\leq m_i - ▱_\ell && \text{(left-Lipschitz)}\\
▱_d &\leq f(▱_\ell) && \text{(parallelogram)}\\
▱_d - f(m_i) &\leq m_i - ▱_\ell\\
▱_d - f(m_i) &\leq ▱_w\\
(\max f) - f(m_i) &\leq (\max f) - ▱_d + ▱_w\\
&\leq (▱_h + ▱_w) + ▱_w\\
&\leq 3▱_h
\end{aligned}$$

Hence this single query can also be charged to our potential drop, bringing the constant factor up to five.

Finally, we pay for lazily associated query attached to the original parallelogram, which risks the best f-value increasing by $▱_w$ which is again dominated by $▱_h$. This increases our constant factor to six.

This completes the analysis for case 2.

The only thing left to consider is the parallelograms in S when the while loop finishes. Some of them have nonzero width, in which case we charge their growth in height directly to their loss in width, using the potential drop of ϕ_{binary}. Others ran out of width, and now we use the fact that queries can only be lazily associated with parallelograms that have width at least a quarter of the maximum width in S. For such a parallelogram, we incur regret at most the maximum with in S when that parallelogram was added to S. Since we lost all our width, we know that ϕ_{binary} dropped by at least a quarter that, and hence we can charge to the drop in ϕ_{binary} times a constant factor (four). As a reminder, the actual best f-value might be $\frac{1}{T}$ better than the best seen f-value when the while loop terminates, but since there are at most T outstanding queries this was only $O(1)$ uncharged regret that we can safely file under the "plus a constant" in the theorem statement.

This finishes the charging accounting and concludes our proof. $\square$