MS&E 213 / CS 269O : Chapter 1
Introduction to "Introduction to Optimization Theory"*

By Aaron Sidford (sidford@stanford.edu)

October 7, 2019

1 What is this course about?

The central problem we consider throughout this course is as follows. We have a function $f : \mathbb{R}^n \to \mathbb{R}$, which we refer to as our *objective function*, and we wish to minimize it, i.e. we wish to solve the following.

$$\min_{x \in \mathbb{R}^n} f(x) \,.$$

This is known as an *unconstrained minimization problem* and it is one of the most well studied problems in optimization.

It is also incredibly prevalent. It is hard to think of cases where we have access to a lot of data and then do not want to optimize over that data set, finding the best thing about it in some way. Maybe we have purchase history for some website and we want to know what is the most popular or we want to fit a linear model to it. Maybe we have the map of the world and we want to know the minimum path from one point to another. Maybe we have a bunch of constraints on how we can spend our time, utilities for what we do, and want to find the best schedule. Optimization broadly concerns the problem of finding such optimal solutions and each of these can be easily modeled as such an unconstrained minimization problem.

Note, that most optimization problems we might think of can be phrased as an unconstrained minimization problems even if that is not the typical way we think of it. For example if we have a set $S \subseteq \mathbb{R}^n$, which we refer to as our *constraint set*, and wish to solve the following, *constrained minimization problem*

$$\min_{x \in S} f(x)$$

we can always define $g : \mathbb{R}^n \to \mathbb{R}$ by

$$g(x) = \begin{cases} f(x) & x \in S \\ \infty & x \notin S \end{cases}$$

and then solve the equivalent unconstrained minimization problem

$$\min_{x \in \mathbb{R}^n} g(x) \,.$$

Consequently, many problems can be written as unconstrained minimization problems, though this may obfuscate some salient properties of the original problem.

*These notes are a work in progress. They are not necessarily a subset or superset of the in-class material and there may also be occasional *TODO* comments which demarcate material I am thinking of adding in the future. These notes will converge to a superset of the class material that is TODO-free. Your feedback is welcome and highly encouraged. If anything is unclear, you find a bug or typo, or if you would find it particularly helpful for anything to be expanded upon, please do not hesitate to post a question on the Piazza or contact me directly at sidford@stanford.edu.

The focus of this class is how to solve these optimization problems *efficiently* while making *minimal assumptions* about f (and possibly S). The key question we will ask in the class is under a certain set of mild assumptions on S what algorithm should we design to solve the problem as quickly as possible. Note that there is a lot freedom in these definitions. There are many assumptions we could make on how we get access to f, what we mean by efficiency, and what we assume about f. In the remainder of this section, we will clarify the way in which we will think of optimization in this class and what we will try to show throughout the course.

1.1 How do we access f?

In order to reduce the assumptions we make about f, throughout this course we will typically assume we only have fairly restrictive access to f. For much of this class, rather than assuming we have our objective function f explicitly, we will assume we can only access f through invocation of some procedure that gives us only limited information of f at a certain point. Formally, we will call this procedure an *oracle* and we will refer to an invocation of the procedure as a *query* to the oracle. When we only access f through an oracle we will say we only have *oracle access* to f.

Three standard oracle assumptions we encounter throughout the class are the following.

Definition 1 (Value Oracle)**.** A *value oracle*, also known as an *evaluation oracle* or a *0-th order oracle* for a function $f : \mathbb{R}^n \to \mathbb{R}$ is a procedure that when queried with a point $x \in \mathbb{R}^n$ outputs the value of f at x, i.e. $f(x)$.

Definition 2 (Gradient Oracle)**.** A *gradient oracle* for a function $f : \mathbb{R}^n \to \mathbb{R}$ is a procedure that when queried with a point $x \in \mathbb{R}^n$ outputs the gradient of f at x, i.e. $\bigtriangledown f(x) \in \mathbb{R}^n$ where $[\bigtriangledown f(x)]_i = \frac{\partial}{\partial x_i} f(x)$.

Definition 3 (First Order Oracle)**.** A *first order oracle* for a function $f : \mathbb{R}^n \to \mathbb{R}$ is a procedure that when queried with a point $x \in \mathbb{R}^n$ outputs both the value of f at x, i.e. $f(x)$, as well as the gradient of f at x, i.e. $\bigtriangledown f(x) \in \mathbb{R}^n$ where $[\bigtriangledown f(x)]_i = \frac{\partial}{\partial x_i} f(x)$.

Note that there are many oracles one can encounter in different situations. For example, a natural oracle in many settings is the following:

Definition 4 (Stochastic Gradient Oracle)**.** A *stochastic gradient oracle* for a function $f \in \mathbb{R}^n$ is a procedure that when given a query point $x \in \mathbb{R}^n$ outputs a random vector $e \in \mathbb{R}^n$ such that $\mathbb{E}e = \bigtriangledown f(x)$ where here the expectation is over the randomness of the oracle.

Note that this oracle is a reasonable abstraction for problems where we have an overwhelming amount of data, want to optimize something about the whole, and can only sub-sample pieces. This occurs in many large scale learning problems. For example, we could imagine that we have n data points, associate each $x \in \mathbb{R}^n$ with a model (or its parameters), and let $f_i(x)$ denote how well the model given by x explains or predicts data point i. The objective, $f(x) \stackrel{\text{def}}{=} \frac{1}{n} \sum_{i \in [n]} f_i(x)$, is then average prediction error or loss when using x to predict a data point. Whereas implementing a gradient oracle would involve computing $\frac{1}{n} \sum_{i \in [n]} \bigtriangledown f_i(x)$, i.e. looking at each data point, if instead we just wanted a stochastic gradient oracle, we could just pick a random $i \in [n]$ and use $\bigtriangledown f_i(x)$ as such a random gradient, which would involve just looking at a single data point. Given its prevalence and utility we will work with this oracle model more later in the course.

1.2 Why the oracle model?

The oracle model is a nice abstraction for working with many problems. In many cases exactly specifying the objective function can be quite complicated and perhaps we really only have access to f through a restricted oracle.

For example, we might be trying to set prices and f is the total utility we get when we try selling items at those prices. Here, we can possibly only evaluate f by actually setting those prices and seeing what people buy, i.e. how the world responds.

As another example, the data may be so large or noisy that we cannot interact with the data directly. Maybe we only have stochastic or adversarial noisy access, in this case the oracle model is again natural.

Even in cases where we have all the information about f, the oracle model may be useful. For example, consider a simple, but more-structured optimization problem like *regression*, where we are given a matrix $\mathbf{A} \in \mathbb{R}^{n \times d}$ and a vector $b \in \mathbb{R}^n$ and wish to solve, $\min_{x \in \mathbb{R}^n} \frac{1}{2}\|\mathbf{A}x - b\|_2^2$. When designing algorithms for this problem it is important to understand when having a simple procedure just based on the gradient (e.g. the ability to apply $\mathbf{A}$ and $\mathbf{A}^\top$ to vectors, as $\frac{1}{2}\|\mathbf{A}x - b\|_2^2 = \mathbf{A}^\top(\mathbf{A}x - b)$) and when are exploiting deeper algebraic structure of $\mathbf{A}$.

In short, oracles provide a nice way to extract the salient information about a problem we wish to use for algorithm design. They also provide a good lens to think about the computational complexity of optimization. Whereas for an explicitly given function, if optimizing it is difficult, i.e. takes a lot of time, proving this is currently typically outside the realm of what we know how to do computational complexity theory. However, if we assume oracle access to the input, in many cases it is possible to obtain provable lower bounds on how many oracle calls are required to obtain a certain accuracy on optimization problems. Thus we may hope for tight bounds on optimization problems under oracle models; ultimately this helps inform us as to what an algorithm can or should be exploiting or not.

1.3 What do we want to do?

Once we have specified our oracle model we wish to design algorithms to minimize our objective function f as efficiently as possible or prove that more efficient minimization is impossible. There are two key questions we need to address here (1) what exactly do we mean by *minimize* and (2) what do we mean by *efficiency*.

There is a serious choice to be made in answering each of these questions. The issue of (1) is important as for almost all the problems we see in this course we will rarely compute an exact minimum value or minimizing point of our objective function (as we will rarely know it exactly). Instead, we will be providing an algorithm that is approximately minimum or optimal in some sense. This is somewhat essential as in the oracle model it is rare to know you are at the exact minimum or optimal value, in some sense the input may just be too noisy. One common notion we will use to parameterize our distance from optimality is the following.

Definition 5 (ϵ-optimal point)**.** For $f : \mathbb{R}^n \to \mathbb{R}$ we call $x \in \mathbb{R}^n$ an *ϵ-optimal point* for some $\epsilon > 0$ if

$$f(x) - f_* \leq \epsilon \text{ where } f_* \stackrel{\text{def}}{=} \inf_{x \in \mathbb{R}^n} f(x)\,.$$

Note that this is a strong notion of optimality as it is global. Although we will achieve this in many cases in other cases we may only obtain local optimality perhaps as measured by the first few moments of the function. For example, we may try to compute an ϵ-critical point.

Definition 6 (ϵ-critical point)**.** For $f : \mathbb{R}^n \to \mathbb{R}$ we call $x \in \mathbb{R}^n$ an *ϵ-critical point* for some $\epsilon > 0$ if $\| \bigtriangledown f(x)\|_2 \leq \epsilon$ where $\|x\|_2 \stackrel{\text{def}}{=} \sqrt{\sum_{i \in [n]} x_i^2}$.[1]

Note that a critical point might not be maximal or minimal apriori but if we have an algorithm that decreases the function value and finds a critical point, this may be a decent proxy for a local minima, that is a point where no direction makes progress if we move in that direction infinitesimally.

Next, (2), what do we mean by efficiency? The running time of our algorithms are going to have essentially two components (1) how many times did we invoke our oracle, i.e. what is the *oracle complexity* or *query*

[1] Note, other norms could be considered here, but we will focus on the ℓ_2 norm in the majority of the course.

complexity of the problem[2], and (2) what was the computational complexity of the work we did to manipulate the results of calling the oracle. Note that both of these are important and it is not always clear which is more critical. For example, in the case of setting prices, the oracle calls could be incredibly expensive, however as the size of data gets large, paying running times of $O(n^4)$ can be prohibitively expensive. We will attempt to minimize each and present our running time in terms of both pieces. However, our emphasis will typically be on oracle complexity.

1.4 What do the algorithms look like?

Since we just have oracle access, just have local information, most algorithms we will see are essentially greedy local methods. They run some simple iterative procedure: essentially, query the oracle, make an improvement and repeat. Such algorithms are known as *iterative methods* and despite the simplicity of the procedure efficient iterative methods and their analysis can be quite complicated. This comes from the fact that sometimes to get the best performance somewhat complicated performance measures need to be used. Both finding the best thing to do with all the information available can be complex and finding the best proxies for progress can be difficult. Figuring out how to design algorithms and these potential functions for analysis is an integral part of this course. While we will start with fairly simple potential functions, they will get more complicated as the course progress.

This class can in fact be viewed, in part, as a introduction to the theory of iterative methods. These are powerful tools in designing fast algorithms for all sorts of problems (both continuous and discrete). They have been the workhorse in practice behind many machine learning algorithms and they have recently used to get numerous breakthrough results in the theory of computation. This is also an incredibly active area of research that this course should equip you to go into.

1.5 What do the problems look like?

So far we have summarized what the course will be about. We will introduce some broad class of objective functions f, specify an oracle model for accessing f, design an iterative method for optimizing f, and characterize the performance of the method in terms of the number of oracle calls (typically the number of iterations of the algorithm) and the total computational cost (typically the work per iteration). By doing this we will have a good sense of the computational boundaries for continuous optimization.

What we haven't specified is what exactly these classes of optimization problems will look like. In the remainder of this chapter we will consider some basic classes of f and apply this lens. This will give us a feel for what kind of assumptions are natural. We will conclude these notes with a summary of the main problem classes we will consider in the future.

2 General Unconstrained Optimization

Let us start with the most general class of unconstrained function minimization. Suppose we have an objective function $f : \mathbb{R}^n \to \mathbb{R}$ and we wish to minimize it. Furthermore, suppose we just have a value oracle for f, i.e. for any $x \in \mathbb{R}^n$ we can compute $f(x)$ with a single query. Lastly, suppose we are promised that the minimizer of f lies in $[0, 1]$, i.e. there is $x_* \in [0, 1]$ with $f(x_*) = f_* \stackrel{\text{def}}{=} \inf_{x \in \mathbb{R}^n} f(x)$, and that $f(x) \in [0, 1]$ for all $x \in [0, 1]$. What is the oracle complexity?

Unfortunately this problem seems quite hard. It doesn't seem like information about any point gives you information about any other point in the domain. We can actually make this formal and show that any

[2]Note that this is different than the computational complexity of the oracle, which would be how expensive the oracle is to implement.

deterministic algorithm which makes a finite number of queries to f cannot in the worst case compute an ϵ-optimal point for this class of functions for any $\epsilon < 1$.[3]

Note that this is a somewhat strong claim in that we want to give a lower bound for *all algorithms*. To show this, we will show that for any specific algorithm which makes k queries and outputs a candidate answer, that answer will be incorrect for some input. Our approach to prove this is based on the idea of a *resisting oracle*, that is we will show that we can provide an algorithm for implementing the oracle such that the output is always consistent with some function f under the given assumptions and for which the output of the algorithm is always incorrect. This is also sometimes called an *adversarial oracle* as we will be implementing the oracle, i.e. choosing the function, adversarially in response to what the algorithm does. Ultimately, if we can design such an oracle then it will ultimately prove (and typically find) an input on which the algorithm fails.

We prove this formally below.

Lemma 7. *There is no deterministic algorithm which when given a value oracle for $f : \mathbb{R} \to \mathbb{R}$ that has the property that the minimizer of f lies in $[0, 1]$ and has $f(x) \in [0, 1]$ for all $x \in [0, 1]$ outputs an ϵ-approximate minimizer for $\epsilon < 1$.*

Proof. Proceed by contradiction and suppose that there is an algorithm that runs in k steps and computes and ϵ-approximate minimizer for $\epsilon < 1$. Consider running this algorithm where in response to each of the queries $x_1, ..., x_k$ the oracle return $f(x_i) = 1$. Further, suppose that the algorithm outputs y as the minimizer on this sequence.

For all $z \in [0, 1]$ consider the function f_z defined for all $x \in \mathbb{R}$ by

$$f_z(x) = \begin{cases} 1 & \text{if } x \notin z \\ 0 & \text{if } x = z \end{cases}.$$

Note that minimizer of f_z is 0 and the only ϵ-approximate minimizer of f_z is the point z. Further, note that for any $z \notin \{x_1, ...x_k\}$ the output of the oracle is consistent with the function f being f_z, i.e. $f_z(x_i) = 1$ for all $i \in [k]$. Further, so long as $z \in [0, 1]$ then f_z is the class of functions we are considering.

Consequently, we have just shown that for any $z \notin \{x_1, ..., x_k\}$ the algorithm when run with the function being f_z will output y. However, since there is a $z \notin \{x_1, ...x_k\} \cup \{y\}$ with $z \in [0, 1]$ (in fact an uncountably infinite number of them) and for such f_z the only ϵ-approximate minimizer is $z \neq y$ we have just shown that when run on such f_z the output of the algorithm is incorrect, completing the proof by contradiction. □

This shows that fully general optimization, even in one dimension is fairly hopeless. This examples highlights that in this general setting optimization is essentially as hard as finding the smallest element in an uncountably infinite set of elements using only a finite number of queries to their values. Note we could try to find something weaker, like a critical point, but again the function could easily be non-differentiable.

3 Continuous Functions

The problem in the previous example was that the value of the function at a point told us very little information about the function. It didn't even give us any further local information about the function. To get around this issue we could try assuming just a little more structure on f, e.g. we could assume that f does not change too quickly, or more precisely that f is continuous. To get provable guarantees we could quantify this and make the following common assumption that f is *L-Lipschitz continuous*.

[3]The same is true for randomized algorithms, but proving lower bounds against such algorithms is outside the scope of the course (but I am happy to discuss in office hours or over Piazza).

Definition 8. We say a function $f : \mathbb{R}^n \to \mathbb{R}$ is *L-Lipschitz with respect to a norm* $\|\cdot\|$ if for all $x, y \in \mathbb{R}^n$ we have

$$|f(x) - f(y)| \leq L \cdot \|x - y\| .$$

Where we recall the following definition of norm:

Definition 9. Recall that $\|x\|$ for $x \in \mathbb{R}^n$ is called a norm on $\mathbb{R}^n$ if the following hold, (1) (absolute homogeneity) $\forall \alpha \in \mathbb{R}$ and $x \in \mathbb{R}^n$ it holds $\|\alpha x\| = |\alpha| \cdot \|x\|$ (2) (triangle inequality) $\forall x, y \in \mathbb{R}^n$ it holds $\|x + y\| \leq \|x\| + \|y\|$, (3) $\|x\| = 0$ if and only $x = \vec{0}$.

Typically in this class we will take $\|\cdot\|$ to be the Euclidean norm, that is $\|x\| = \|x\|_2 \stackrel{\text{def}}{=} \sqrt{\sum_{i\in[n]} x_i^2}$. If we don't say the norm, we will assume it to be the Euclidean norm. Occasionally we will work with other Schatten p-norms, i.e. $\|x\|_p \stackrel{\text{def}}{=} (\sum_{i\in[n]} |x_i|^p)^{1/p}$ with $\|x\|_\infty \stackrel{\text{def}}{=} \max_{i\in[n]} |x_i|$. Note that all norms are equivalent up to scaling in finite dimensions and thus if we use a norm ever to define a limit it doesn't matter, but if we use it in the definition of Lipschitz, it does matter and affect the value of L.

So let's go back to our example. Suppose we have $f : \mathbb{R} \to \mathbb{R}$ and suppose that we know that f obtains its minimum on $[0, 1]$ and we know that $f \in [0, 1]$. Now further suppose the f is L-Lipschitz. This tells us that as we move a point by δ that the value of the function can change by at most $L|\delta|$. In other words, if we look at the lines of slope L and $-L$ through a point the value of the function is always between these two lines.

Now, with this assumption, how many oracle queries do we need to find an ϵ-optimal point? One natural algorithm we could consider for solving this problem is to simply query a sequence of spaced out points and return the one with the best value. We analyze this algorithm in the following lemma.

Lemma 10. *If $f : \mathbb{R} \to \mathbb{R}$ is L-Lipschitz, $f(x_*) = f_* \stackrel{\text{def}}{=} \inf_{x\in\mathbb{R}^n} f(x)$ for some $x_* \in [0, 1]$, and $f(x) \in [0, 1]$ for all $x \in [0, 1]$ then for all $\epsilon > 0$ there is an algorithm that finds an ϵ-optimal point with $\lceil L/\epsilon \rceil$ queries to a value oracle.*

Proof. Consider the algorithm which queries the value oracle at the points i/k for all $i \in [k]$ and returns the point with minimum value. Note that $|x_* - \frac{i}{k}| \leq \frac{1}{k}$ for some $i \in [k]$ and therefore $|f(i/k) - f_*| \leq \frac{L}{k}$. Consequently, if q is the point returned by algorithm then by design we have that

$$f(q) \leq f(i/k) = f_* + f(i/k) - f_* \leq f_* + \frac{L}{k} .$$

Picking $k = \lceil L/\epsilon \rceil$ then yields the result. □

Now a natural question to ask is, is this tight? Can we do better? Below we show that in the adversarial oracle setting (i.e. where the oracle just needs to be consistent with some underlying function) we cannot do too much better. To show this we will construct an L-Lipschitz function with these properties, where the only ϵ-optimal points (and all information about it) is restricted to a small bounded region. Leveraging this function we can lower bound the number of queries to an value oracle needed to compute an ϵ-optimal point.

In particular, we consider the following function that is 1 everywhere except for a small region where all the ϵ-optimal points lie.

Lemma 11. *For all $z \in \mathbb{R}$ and $L, \alpha \geq 0$ the function $f_{z,\alpha,L} : \mathbb{R} \to \mathbb{R}$ is defined for all $x \in \mathbb{R}$ by*

$$f_{z,\epsilon,L}(x) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } |x - z| \geq \frac{\alpha}{L} . \\ 1 - \alpha + L\,|x - z| & \text{otherwise} \end{cases}$$

is L-Lipschitz and x is ϵ-optimal for $\epsilon < \alpha$ if and only if $|x - z| \leq \frac{\epsilon}{L}$

Proof. We can write this function alternatively as $f_{z,\alpha,L} = 1 - \alpha + \min\{\alpha, L|x - z|\}$. By design, when we change x by δ the value of this function changes by at most $L\delta$ since the function is either constant or changing locally at rate L by the $|x - z|$ term. Consequently, $f_{z,\alpha,L}$ is L-Lipschitz as desired. Further, note that the function obtains its minimum value of $1 - \alpha$ at z and then grows in value monotonically as we move away from z yielding the desired bound on the set of ϵ-optimal points.

Using this we show that for any interval of length larger than $\frac{4\epsilon}{L}$ we can construct two L-Lipschitz functions which both have value 1 everywhere outside the interval and have non-overlapping intervals containing the ϵ-optimal points. □

Lemma 12. *For any $c \in \mathbb{R}$ and $\epsilon, L \geq 0$ the functions $f_{c+\frac{\epsilon}{L},\epsilon,L}$ and $f_{c-\frac{\epsilon}{L},\epsilon,L}$ have disjoint sets of ϵ'-optimal points for all $\epsilon' < \epsilon$ and have value 1 outside the interval $[c - \frac{2\epsilon}{L}, c + \frac{2\epsilon}{L}]$.*

Proof. Each of the functions has value that is not 1 on an interval of length $\frac{2\epsilon}{L}$ and these intervals overlap only on the boundary at which point they have the same value. □

Using this we see that if we consider the resisting oracle that always returns value 1 (which is always consistent with the 1-Lipschitz function that is the constant function, i.e. $f(x) = 1$ for all $x \in \mathbb{R}$) then if the algorithm does not query a point in an interval of length $\frac{4\epsilon}{L}$ contained in $[0, 1]$ then the algorithm will fail on some input. Since there are two functions with disjoint ϵ-optimal points that are consistent with the oracle. Using this we can prove the following lower bound.

Lemma 13. *Suppose that there is an algorithm always computes an ϵ-optimal point with k queries to a value oracle for $f : \mathbb{R} \to \mathbb{R}$ that is L-Lipschitz and with $f(x_*) = f_* \stackrel{\text{def}}{=} \inf_{x \in \mathbb{R}^n} f(x)$ for $x_* \in [0, 1]$ and $f(x) \in [0, 1]$ for all $x \in [0, 1]$. Then $k \geq \frac{1}{4} \cdot \frac{L}{\epsilon} - 1$.*

Proof. Let $0 \leq x_1 \leq x_2 \leq ... \leq x_{k'} \leq 1$ be the $k' < k$ points queried by the algorithm (not necessarily in order) that lied between 0 and 1 when the algorithm was given value 1 as the result of each of its value queries. Now consider the intervals $[0, x_1], [x_2, x_3], ..., [x_{k'-1}, x_{k'-1}], [x_{k'-1}, 1]$. Since these $k' + 1$ have lengths summing to 1 at least one of these intervals has length $1/(k' + 1) \geq 1/(k + 1)$. If $k > 4\epsilon/L$ then by Lemma 12 whatever the algorithm outputs will be incorrect for some function consistent with the output of the oracle. Consequently $1/(k + 1) \leq \frac{4\epsilon}{L}$ and the result follows. □

We have just shown an upper bound of $\lceil L/\epsilon \rceil$ on the number of queries to get an ϵ-optimal point and a lower bound of $(1/4)(L/\epsilon) - 1$ queries. These are the same up to additive and multiplicative constants. Interestingly, the arguments above can be sharpened slightly to show that $L/(2\epsilon)$ is the optimal number of queries up to an additive constant (with a small change to the upper bound algorithm).

Lemma 14. *If $f : \mathbb{R} \to \mathbb{R}$ is L-Lipschitz, $f(x_*) = f_* \stackrel{\text{def}}{=} \inf_{x \in \mathbb{R}^n} f(x)$ for some $x_* \in [0, 1]$, and $f(x) \in [0, 1]$ for all $x \in [0, 1]$ then for all $\epsilon > 0$ there is an algorithm that finds an ϵ-optimal point with $\lceil L/(2\epsilon) \rceil + 1$ queries to a value oracle.*

Proof. Consider the algorithm which queries the value oracle at the points i/k for all $i \in [k]$ and also queries the point 0 and then returns the point, q_*, for which the oracle output the smallest value. Note that the intervals $[0, 1/k], [1/k, 2/k], ..., [k - 1/k, 1]$ partition $[0, 1]$ and that x_* lies in one of these intervals. Since the length of each interval is $1/k$ we see that x_* is distance $1/(2k)$ from one of the endpoints. Since we queried the oracle at each endpoint we have that $|x_* - q| \leq 1/(2k)$ for one of the queried points and therefore

$$f(q_*) \leq f(q) = f_* + f(q) - f_* \leq f_* + L|q - x_*| \leq f_* + \frac{L}{2k}.$$

Picking $k = \lceil L/(2\epsilon) \rceil$ and noting that this algorithm queried $k + 1$ points yields the result □

Lemma 15. *Suppose that there is an algorithm always computes an ϵ-optimal point with k queries to an value oracle for $f : \mathbb{R} \to \mathbb{R}$ that is L-Lipschitz and with $f(x_*) = f_* \stackrel{\text{def}}{=} \inf_{x \in \mathbb{R}^n} f(x)$ for $x_* \in [0,1]$ and $f(x) \in [0,1]$ for all $x \in [0,1]$. Then $k \geq \frac{1}{2} \cdot \frac{L}{\epsilon} - 1$.*

Proof. Let $0 \leq x_1 \leq x_2 \leq ... \leq x_{k'} \leq 1$ be the $k' < k$ points queried by the algorithm (not necessarily in order) that lied between 0 and 1 when the algorithm was given value 1 as the result of each of its value queries. Now consider the intervals $[0, x_1], [x_2, x_3], ..., [x_{k'-1}, x_{k'-1}], [x_{k'-1}, 1]$. If one interval has length $> 4\epsilon/L$ then by Lemma 12 the algorithm will be incorrect on some input. Further, if two intervals have length $> 2\epsilon/L$ then by Lemma 11 the algorithm will be incorrect on some input (since the function could be the $f_{z,\alpha,L}$ in two locations corresponding two the two intervals and these functions have disjoint ϵ-optimal points). Consequently, we have that 1 interval has length $\leq 4\epsilon/L$ and the others all have length $\leq 2\epsilon/L$. Since the lengths of these intervals sum to 1 and there are $k'+1$ intervals we have $2k'\epsilon/L + 4\epsilon k'/L \geq 1$ and $2\epsilon(k'+1) \geq L$. As $k \geq k'$ the result follows. □

Throughout most of the class we will not be too focused on these exact constants and will be more focused on the rates. We will informally use the notation $O(\cdot)$ to hide additive and multiplicative constants in our upper bounds and $\Omega(\cdot)$ to hide the same in our lower bounds. In this notation the above results can be restated as saying that $O(L/\epsilon)$ queries suffice to compute an ϵ-optimal point and $\Omega(L/\epsilon)$ queries are needed. We say that the oracle or query complexity of a problem is $\Theta(\alpha)$ if $O(\alpha)$ queries suffice and $\Omega(\alpha)$ queries are needed. Consequently, we say that the query complexity of this one-dimensional Lipschitz optimization problem is $\Theta(L/\epsilon)$.

A natural question to ask then is, what happens if we are in higher dimensions?

Lemma 16. *If $f : \mathbb{R}^n \to \mathbb{R}$ is L-Lipschitz with respect to $\|\cdot\|_\infty$ such that $f_* = f(x_*)$ for some $x_* \in [0,1]^n$ and $f(x) \in [0,1]$ for all $x \in [0,1]^n$ then for all $\epsilon \in (0,1)$ there is an algorithm that finds an ϵ-optimal point with $\lceil \frac{L}{\epsilon} \rceil^n$ queries to a value oracle.*

Proof. We query the value oracle at points $x \in \mathbb{R}^n$ for all vectors where $x_i = i/k$ for $i \in [k]$ and return the point with the minimum value. Note that there are k^n such points. Note that for all $i \in [n]$ it is the case that $|x_*(i) - \frac{j}{k}| \leq \frac{1}{k}$ for some $j \in [k]$ and therefore for some point q that was queried we have that $\|q - x_*\|_\infty \leq \frac{1}{k}$ and therefore

$$f(q) - f_* = |f(q) - f(x_*)| \leq L\|q - x_*\|_\infty \leq \frac{L}{k}.$$

Consequently, $f(q) \leq f_* + \frac{L}{k}$ and setting $k = \lceil L/\epsilon \rceil$ then yields the result. □

Using similar tricks as before we can show that an algorithm needs $(\Omega(L/\epsilon))^n$ queries to solve this problem for some ϵ. Consequently, while global minimization for Lipschitz functions is doable, it comes at an exponential cost in dimension and we only get a weak, i.e. polynomial dependence on the error ϵ.

4 What Else?

What will we do in the rest of the course? Now that we understand the game a bit we can better explain the structure of the course. In the next few lectures we will restrict our attention to the minimization of *smooth functions* that is functions where the gradient doesn't change to quickly, i.e. the gradient is Lipschitz. We will show that under this assumption we may not be able to compute global optimum efficiently, but we can compute critical points fairly sufficiently.

With the structure of smooth functions in hand we will add the assumption that our function is convex, i.e. that $f(\alpha x + (1-\alpha)y) \leq \alpha \cdot f(x) + (1-\alpha) \cdot f(y)$ for all x and y in the domain and $\alpha \in [0,1]$, that is the function lies underneath the curve between any two points. These assumptions will be enough that we can provably find global optimum of our function. We will consider different algorithms that better exploit

the exact convexity and smoothness assumptions we make. We will also discuss extensions of this to other norms and other oracle models which may be useful in various settings.

Next, we study the structure of convex functions that are continuous, i.e. Lipschitz, but not necessarily smooth. Here the structure is a little more complicated and we will consider two classes of algorithms. First we will consider algorithms with a great dependence on the desired ϵ-accuracy but a poor (i.e. polynomial) dependence on dimension, these are known as cutting plane method and they underly the theory of optimization and are critical for many of the best guarantees we have for solving both combinatorial and continuous problems. Then we will consider algorithms with a worse dependence on ϵ but a better dependence on dimension, the algorithms will be called mirror descent or subgradient descent; these are used in various online, streaming, stochastic, and practical settings to get good performance.

With a firm understanding of smooth and non-smooth convex optimization we will cover some more advanced topics in iterative methods like acceleration and variance reduction.

Finally, we will conclude the course by studying higher-order methods, that is iterative methods which compute not just the gradient but the Hessian of a function, i.e. Newton's method. Moreover, we will show how to use such methods to outperform cutting plane methods for broad classes of more structured optimization. In particular, we will introduce interior point methods which provide the fastest known algorithms for problems like linear programming and semidefinite programming.