**Lecture 1**

## 1.1 Introduction

In this course, we study efficient algorithms for solving optimization problems in the following general form:

$$\min_{x \in Q} f(x), \tag{1.1}$$

where $Q \subseteq \mathbb{R}^n$ is a given *feasible set* or *constraint set*, $x \in Q$ is called the vector of *optimization parameters* or the *decision variables*, and $f : Q \to \mathbb{R}$ is a target *objective function*, which we always assume to be continuous. The goal is to find a best possible point $x^\star \in Q$ that achieves the minimum (1.1).

Modern models that are used in practice typically include a large number of parameters, which means that the dimension $n$ of the variable space is large (several thousands, millions, or even trillions of parameters). Therefore, using computers is the only possible way to find a solution. We will study the design of optimization algorithms, along with analysis of their *performance guarantees*, i.e. how fast the algorithm converges, or how many basic operations, such as gradient computations or matrix-vector products, are needed to obtain a desirable solution.

We also note that since $\max_x f(x) = -\min_x [-f(x)]$, we can always focus only on studying minimization problems, as maximization is trivially covered by putting minus in front of the objective.

### 1.1.1 Basic Classification of Optimization Problems

Depending on the structure of the objective and the constraint set in (1.1), we use the following standard classification of optimization problems:

**Unconstrained Optimization.** This is the case when $Q \equiv \mathbb{R}^n$, i.e. there are no any constraints in the problem, and the decision variables can take any values in a finite-dimensional real vector space $\mathbb{R}^n$:

$$\min_{x \in \mathbb{R}^n} f(x). \tag{1.2}$$

Unconstrained optimization problems are commonly considered to be *simpler* than constrained ones, and we devote a significant part of this course to studying the unconstrained situation. However, as we will see, often the ideas from unconstrained optimization can be successfully applied to more general constrained problems, and these use cases will be thoroughly studied.

We call the problems of the form (1.2) as

- **Smooth Optimization**, when the function $f : \mathbb{R}^n \to \mathbb{R}$ is sufficiently "smooth", at least differentiable (e.g. $f(x) = x^2, x \in \mathbb{R}$);
- **Non-smooth Optimization**, when $f$ is a non-differentiable function (e.g. $f(x) = |x|$).

Non-smooth optimization problems are usually regarded as more challenging, while for smooth problems we can ensure strictly positive progress in each iteration of the method.

**Constrained Optimization.** When $Q \subset \mathbb{R}^n$ (strictly), we refer to (1.1) as a constrained optimization problem. Sometimes, the constraint set can be naturally induced as the *domain* of the objective function, i.e., $Q = \operatorname{dom} f$, and may even be omitted from the problem formulation. More often, constraints are given explicitly in the model based on natural physical observations. There are two important situations that are significantly different from the perspective of algorithmic design:

- **Simple Constraints.** This means that the set $Q$ is "explicit enough" and easily understandable. The classic example of simple constraints is a ball of certain radius:

$$Q \;=\; \Big\{x \in \mathbb{R}^n \;:\; \|x - x_0\| \le R\Big\},$$

  where $\|\cdot\|$ is some standard norm (e.g., Euclidean norm $\|\cdot\|_2$ or $\|\cdot\|_\infty$-norm). Another example of simple constraints is the standard simplex:

$$\Delta_n \;=\; \Big\{x \in \mathbb{R}^n_+ \;:\; \langle e, x\rangle = 1\Big\}.$$

  We denote by $\mathbb{R}_+$ the set of nonnegative reals, and by $e = (1, \ldots, 1)^\top$ the vector of all ones. The notion of "simplicity" is informal, and it basically means that we can perform some standard operations with the set efficiently. One of the most important is the *projection* operation[1]:

$$\operatorname{proj}_Q(x) \;:=\; \operatorname*{arg\,min}_{y \in Q} \|y - x\|_2.$$

  For "simple" sets, projection onto them can be done efficiently, either with an explicit formula (as with projection onto the Euclidean ball, or onto a box), or with an efficient algorithm (as is the case for the simplex; projection onto $\Delta_n$ can be computed in $O(n \log n)$ time).

- **Functional Constraints.** This situation is more difficult. The set $Q$ is given in the following *general form*:

$$Q \;=\; \Big\{x \in \Omega \;:\; g_1(x) \le 0,\; \ldots,\; g_m(x) \le 0\Big\}, \tag{1.3}$$

  where $\Omega \subseteq \mathbb{R}^n$ is possibly some simple set, or the whole vector space, and $g_i : \Omega \to \mathbb{R}$, $1 \le i \le m$ are given functions. Note that it is enough to use only "$\le$" in (1.3), as a constraint of the form $g(x) \ge 0$ can be rewritten as $-g(x) \le 0$, and a constraint of the form $g(x) = 0$ is equivalent to the pair of inequality constraints: $g(x) \le 0$ and $-g(x) \le 0$. Sets in the general form (1.3) can be quite difficult and require special attention when solving optimization problems.

**Convex Optimization.** We call optimization problem (1.1) convex if the set $Q$ is convex (that is, for any two points it contains the whole segment) and the objective $f$ is a convex function (i.e., its epigraph is a convex set). Convex optimization problems play an outstanding role in the optimization theory and, without a doubt, are among the problem classes that admit the most efficient algorithms. Moreover, ideas obtained from the convex world help significantly in solving other non-convex problems. We will devote a significant amount of time to the theory and algorithms of Convex Optimization.

[1] We will always assume projection in the Euclidean norm $\|\cdot\|_2$, unless explicitly specified.

### 1.1.2 Types of Solutions

Any point that belongs to the constraint set, $x \in Q$, we call *feasible point*. And points outside of the set, $x \notin Q$ are called *infeasible*.

A point $x^\star \in Q$ is called *global solution* to problem (1.1), if

$$f(x^\star) \quad \leq \quad f(x), \qquad x \in Q.$$

Finding a global solution is our **ideal goal**, which can be very difficult to achieve. We denote the optimal function value by $f^\star = f(x^\star)$. In some cases, it may be easier to compute $f^\star$ than $x^\star$, while in general, finding the optimal function can be equally difficult.

Note that we can easily have a unique global solution (e.g., $x^\star = 0$ for $f(x) = x^2$), many global solutions ($x^\star \in \mathbb{R}$ for $f(x) \equiv 0$) and no solutions at all ($f(x) = e^x$).

A point $\bar{x} \in Q$ is called a *local solution* to problem (1.1), if there exists its neighborhood $\bar{x} \in U \subseteq \mathbb{R}^n$ (i.e. $U$ is an open set containing $\bar{x}$) such that

$$f(x^\star) \quad \leq \quad f(x), \qquad x \in U \cap Q.$$

For general non-convex problems, finding a local solution is a much more tractable goal than finding a global one. For convex problems, these two concepts appear to coincide.

### 1.1.3 Why Continuous?

In this course, we are interested in solving *continuous* optimization problems. This means that the target objective $f$ is a continuous function (in most cases, it will be even differentiable), and the variables $x \in Q \subseteq \mathbb{R}^n$ take a continuous range of values.

Intuitively, it is clear that continuous decisions are much easier than discrete ones. Imagine that we have to answer the following question: *Will it snow tomorrow?*, where $x$ represents our answer. In the case of a discrete space, we might typically have two possible answers: $x = 1$ *(yes, it will snow)* or $x = 0$ *(no, it won't snow)*. Note that such an exact weather prediction is very hard—not only for human intelligence, but for the most advanced AI systems as well. At the same time, if we allow for a *continuous space of answers*, $0 \leq x \leq 1$, where $x$ represents the probability of snow, it is much easier to predict that tomorrow *there is a high chance of snow*, (e.g., $x \geq 0.8$).

These observations show that in real life, continuous decisions are often easier to make than discrete ones. It appears to be a very fruitful approach to work with a continuous space of parameters. Even if our original problem is discrete (such as finding a maximum cut in a graph), we can turn it into a much easier one by applying a *continuous relaxation*.

### 1.1.4 Examples

Let us show a few basic examples demonstrating that even the simplest optimization problems might be difficult to solve.

**Constrained Discrete Problem.** Consider one-dimensional space of variables $x \in \mathbb{R}$ and the following two quadratic functions:

$$g_1(x) \quad = \quad x^2 - 1, \qquad\qquad g_2(x) \quad = \quad 1 - x^2.$$

Note that both of them are not only continuous but infinitely differentiable, and might be considered analytically ideal. However, if we look at the constraint set given by two inequalities with these

function,

$$\begin{aligned} Q &= \{x \in \mathbb{R} \,:\, g_1(x) \le 0,\ g_2(x) \le 0\} \\ &= \{x \in \mathbb{R} \,:\, x^2 \le 1,\ x^2 \ge 1\} \\ &= \{x \in \mathbb{R} \,:\, x^2 = 1\} \;=\; \{\pm 1\}, \end{aligned}$$

we obtain the descrete choice of $x$. Thus, a constrained optimization problem even with innocent-looking functional inequality constraints might be hard. This additionally explains why unconstrained optimization is typically considered to be simpler.

**Linear Programming.** One of the most important examples of a non-trivial constrained problem that can be solved efficiently and is rich in applications is *linear programming*. This is the problem with the simplest possible functional constraints, $g_i(x) = \langle a_i, x\rangle - b_i$, that are affine functions, for given vectors $a_i \in \mathbb{R}^n$ and numbers $b_i \in \mathbb{R}$, $1 \le i \le m$. And the objective is a linear function: $f(x) = \langle c, x\rangle$, for some $c \in \mathbb{R}^n$. Any linear programming formulation can be cast into this form:

$$\min_{x \in \mathbb{R}^n} \Big\{ \langle c, x\rangle \;:\; \langle a_1, x\rangle \le b_1,\ \ldots, \langle a_m, x\rangle \le b_m \Big\}.$$

Forming the matrix $A \in \mathbb{R}^{n\times m}$ with vectors $\{a_i\}$ as its columns, and the vector $b = (b_1, b_2, \ldots, b_m)^\top \in \mathbb{R}^m$, we can write down the constraint set in the following matrix notation:

$$Q \;=\; \Big\{ x \in \mathbb{R}^n \;:\; A^\top x \le b \Big\}.$$

The set defined by affine inequalities is called a *polyhedron*.

Thus, an *instance* of the linear programming problem is given by the triple of *input data*: $\{A, b, c\}$. Linear programming is already non-trivial to solve. Luckily, there exist efficient *polynomial-time* algorithms for linear programming, which we will study in this course. It is notable that the ideas behind these methods can be successfully employed for more general subclasses of nonlinear optimization.

**Feasibility Problem.** A closely related to optimization problems is the *feasibility problem*, that is, for a given set $Q \subseteq \mathbb{R}^n$ to find a point in this set:

$$x^\star \;\in\; Q. \tag{1.4}$$

For example, for $Q = \{x \in \mathbb{R}^n : Ax = b\} = \{x \in \mathbb{R}^n : Ax - b \le 0,\ b - Ax \le 0\}$, the feasibility problem is to solve the linear system: $Ax^\star = b$.

The feasibility problem (1.4) can be seen as a trivial instance of the optimization problem (1.1), when the objective function is zero: $f(x) \equiv 0$. However, we can also do the opposite, by turning *any* optimization problem into a sequence of feasibility problems.

For optimization problem $\min_{x\in Q} f(x)$ with an arbitrary objective, we can introduce an *extra variable* $t \in \mathbb{R}$ and *extra constraint* $f(x) \le t$. Then, it is easy to see that our problem is equivalent to the following one with the univariate linear objective:

$$\min_{(x,t)\in Q'} t, \tag{1.5}$$

where $Q' = \{(x, t) \in \mathbb{R}^{n+1} \;:\; x \in Q,\ f(x) \le t\}$ is the new constraint set with extended dimension. Then, to solve (1.5) we can run the simple *binary search* over $t$, finding the smallest value $t^\star$ such that $(x, t^\star) \in Q'$. Therefore, feasibility problems are in general as hard as optimization ones.

**The Most Difficult Problem in the World.** Let $x^\star \in \mathbb{R}^n$ be a fixed point which is unknown to us. Set the following objective function:

$$f(x) \;=\; \begin{cases} 0, & x = x^\star; \\ 1, & \text{otherwise.} \end{cases} \tag{1.6}$$

Then, to solve the unconstrained problem, $\min_{x \in \mathbb{R}^n} f(x)$, means *to find* $x^\star$, which is arbitrary! Thus, the good news is that *any problem in the world can be encoded as optimization problem.* The language of optimization is indeed universal and can be applied in a vast variety of applications. However, it is impossible to hope to have an algorithm that solves problems like (1.6). We come to the pessimistic conclusion that, unfortunately, *optimization problems are generally unsolvable.*

Note that the objective function in (1.6) can be easily made continuous, as to show in the following simple exercise.

**Exercise 1.1.1.** Let $x^\star \in \mathbb{R}^n$ be fixed. For any $\varepsilon > 0$, construct a function $f_\varepsilon : \mathbb{R}^n \to \mathbb{R}$ such that

1. $f_\varepsilon$ is continuous;
2. $f_\varepsilon(x^\star) = 0$;
3. $f_\varepsilon(x) = 1$ for all $x \in C_\varepsilon$, where $C_\varepsilon$ is the closed complement of the Euclidean ball:

$$C_\varepsilon \;=\; \{x \in \mathbb{R}^n \;:\; \|x - x^\star\|_2 \geq \varepsilon\}.$$

*Hint.* Consider the case $n = 1$ first.

Therefore, the idea of trying to develop the most universal algorithm capable of solving any problem in the world is hopeless. Instead, we will investigate specific *problem classes* that we can solve with *efficient algorithms*. For each problem class, we will associate its corresponding *complexity*, that will describe the methods' efficiency.

## 1.2 Complexity of Optimization Problems

To understand the importance of working with problem classes, let us consider another extreme situation: assume that we have *one fixed* function $f_0 : \mathbb{R}^n \to \mathbb{R}$ which has global minimum $x^\star$. What is *the best algorithm* for solving this problem? Undoubtedly, the following method will be very efficient:

**return** $x^\star$

as it will solve the problem immediately. Moreover, it will work perfectly on *all functions* that have a minimum at $x^\star$. However, on any other function, this hard-coded method will fail.

### 1.2.1 Notion of Problem Class

What we want instead is not a single problem, but a *family of problems*, which we call a problem class $\mathcal{P}$. Examples include:

- $\mathcal{P} = \{f_0\}$ consisting of one function — *too small* to be interesting.
- $\mathcal{P} = \{f \;\text{ s.t. }\; f \in \mathcal{C}(\mathbb{R}^n)\}$ consisting of all continuous functions — *too large*, as we have seen in the previous section.

- $\mathcal{P} = \{f \text{ s.t. } f \text{ is convex and } \nabla f \text{ is Lipschitz}\}$ — already *much better*; we will study this problem class in more detail in the following lectures.
- $\mathcal{P} = \{(f, Q) \text{ s.t. } f \text{ is linear and } Q \text{ is a polyhedron}\}$ — *linear programming*, that is polynomially solvable.
- ...

Performance of an algorithm is measured over *all problems* from $\mathcal{P}$. Note the same algorithm can behave differently for different problem classes.

### 1.2.2 Oracles

Another important ingredient is the notion of *oracle*, which describes how exactly an algorithm can learn any information about the function. Indeed, if the algorithm somehow "knew" the entire function, it could simply output its minimum immediately. However, this is unrealistic, except for some rare and simple cases where we can calculate the minimum explicitly by hands, such as minimizing a univariate quadratic function.

The most common case is when an algorithm has access to so-called *black-box local* oracles, which is for any given $x \in \operatorname{dom} f$ returns a local information $\mathcal{O}(x)$ about the function in a small neighborhood of $x$. For example,

- **Zeroth-order oracle:** $\mathcal{O}(x) = \{f(x)\}$ returns only the function value at the given point.
- **First-order oracle:** $\mathcal{O}(x) = \{f(x), \nabla f(x)\}$ returns both the function value and the gradient vector (assuming that the function is differentiable).
- **Second-order oracle:** $\mathcal{O} = \{f(x), \nabla f(x), \nabla^2 f(x)\}$ returns the function value, the gradient, and the Hessian matrix.

From a theoretical perspective, the following oracles are also interesting:

- **$p$th-order oracle:** $\mathcal{O}(x) = \{f(x), \nabla f(x), \ldots, \nabla^p f(x)\}$ returns all derivatives of the objective up to the order $p \geq 1$.
- **Full local oracle:** $\mathcal{O}(x)$ returns all function values in a small neighborhood $U \subset \mathbb{R}^n$ of the point $x \in \mathbb{R}^n$: $\mathcal{O} = \{f(y) : y \in U\}$. Clearly, having all function values in an open neighborhood of $x$ is sufficient to recover all derivatives of $f$.

Sometimes, even first-order information is unavailable or expensive to obtain exactly, as is often the case in stochastic problems or when the objective function is non-differentiable. At the same time, we, as algorithm designers, almost always have additional structural knowledge about the problem or its class, such as in linear programming, that helps in developing the best possible algorithms.

### 1.2.3 What is Optimization Algorithm?

To formally define an optimization algorithm, we associate it with three sequences of mappings:

- **Main iterates:** $(A_0, A_1, \ldots)$ — each rule $A_k$ describes how to generate the next point $x_{k+1}$ at every iteration $k \geq 0$ of the method, given all prior learned information about the objective. Hence, the main iterates of our method are as follows, for every $k \geq 0$:

$$x_{k+1} \;=\; A_k(I_k), \qquad \text{where} \qquad I_k \;=\; (\mathcal{O}(x_0), \ldots, \mathcal{O}(x_k)).$$

  The method starts with an initialization $x_0 \in Q$, which is part of the algorithm.

- **Stopping conditions:** $(S_0, S_1, \ldots)$ — each rule $S_k$ decides at each iteration $k \geq 0$ whether to stop the method or continue its running, based on all prior information $I_k$.

- **How to form the result:** $(R_0, R_1, \ldots)$ — if we decide to stop at a certain iteration, $R_k$ forms the output given all information we have seen so far.

Therefore, we define the following *general algorithmic scheme:*

**Initialization:** $x_0 \in Q$
**For** $k \geq 0$ **iterate:**
$I_k \;=\; \{\, \mathcal{O}(x_0), \ldots, \mathcal{O}(x_k) \,\}$ // collect information
**If** $S_k(I_k)$ **then** // stopping condition
  **return** $R_k(I_k)$ // return the result
$x_{k+1} = A_k(I_k)$ // compute next point

The return rule $R_k(I_k)$ is typically the simplest part of the algorithm. While in practice we might return the point with the smallest function value among $\{x_0, \ldots, x_k\}$ or a weighted average of the iterates, we can, without loss of generality, assume that the algorithm returns the latest point:

$$R_k(I_k) \;\equiv\; x_k.$$

Indeed, if this is not the case, we can modify the algorithm so that after $S_k(I_k)$ is triggered, it performs an extra "dumb" iteration. This iteration replaces $x_{k+1} = A_k(I_k)$ with $x_{k+1} = R_k(I_k)$ and then terminates, returning the last point. Such a modification costs just one additional oracle call.

### 1.2.4 Stopping Conditions

Note that we formally allow the algorithm to perform an infinite number of iterations if the stopping condition is never triggered. However, in practice, we always run a method for a *finite number of iterations*. As a consequence, we cannot hope to obtain an exact solution $x^\star$ as an output. What we get instead is an approximation, $x_k \approx x^\star$, where $x_k$ denotes the method's result.

The following measures of inexactness are the most popular for unconstrained minimization of a differentiable function, given a fixed *desired accuracy* $\varepsilon > 0$:

- **Small functional residual:** $f(x_k) - f^\star \leq \varepsilon$.
- **Small pointwise distance:** $\|x_k - x^\star\| \leq \varepsilon$.
- **Small gradient norm:** $\|\nabla f(x_k)\| \leq \varepsilon$.

It is important to be aware that both the choice of the inexactness measure and the required accuracy level $\varepsilon > 0$ are *included in the problem formulation*. Correspondingly, in our formal algorithmic scheme, we assume that the method's stopping condition ensures that the returned point satisfies the desired guarantee.

### 1.2.5 Definition of Complexity

We are ready to formally define the notion of *complexity* for an optimization algorithm, as applied to a specific problem class.

The formal definition of the problem class includes all three key elements, which we have already discussed:

- A family of problems $\mathcal{P}$ , which describes both the type of objective and the constraints;
- A measure of inexactness and the required accuracy level $\varepsilon > 0$;
- An oracle $\mathcal{O}$, which is a type of information available to an algorithm.

The oracle (which is part of the problem formulation) typically determines the class of algorithms that we consider: for example, the class of all first-order methods, second-order methods, etc.

**Definition 1.2.1.** The *complexity* of an optimization algorithm on a problem is the *total number of oracle calls* required to solve the instance with a fixed accuracy $\varepsilon > 0$.

Note that the complexity can be $+\infty$ if the method is unable to solve the problem with the given accuracy in a finite number of iterations. Often, this notion of complexity is also called *oracle complexity*, *analytical complexity*, or *iteration complexity* of the method.

**Definition 1.2.2.** The *complexity* of an optimization algorithm on a problem class is the *maximum* over complexity on a problem $p$, over all $p \in \mathcal{P}$, with a fixed accuracy $\varepsilon > 0$.

In other words, for a fixed algorithm, we pick the "worst" problem within our problem class. Thus, this is often called the *worst-case* complexity.

## 1.3 Grid Search Algorithm

### 1.3.1 Global Optimization

To illustrate the new concept, we consider the following example of a problem class.

The problem:

$$\min_{x \in B} f(x), \tag{1.7}$$

where $B = \{x \in \mathbb{R}^n \ : \ \|x\|_\infty \leq R\}$ is a ball of radius $R > 0$ in $\ell_\infty$-norm: $\|x\|_\infty := \max_{1 \leq i \leq n} |x^{(i)}|$, and $f : B \to \mathbb{R}$ is a Lipschitz continuous function. That is, for some constant $L > 0$, it holds:

$$|f(y) - f(x)| \ \leq \ L\|y - x\|_\infty, \qquad x, y \in B. \tag{1.8}$$

Note that we can employ different norms in (1.8), but it is convenient to use $\ell_\infty$-norm as in the constraint set. From (1.8) it follows that the function is continuous and hence it achieves its global minimum $x^\star$ over compact set $B$.

Parameters of our problem class are:

- Dimension $n \geq 1$;
- Radius of the ball $R > 0$;
- Lipschitz constant $L > 0$.

For the accuracy measure we fix the *functional residual* condition. Thus we want to find a point $\bar{x} \in B$ that satisfies

$$f(\bar{x}) - f^\star \quad \leq \quad \varepsilon. \tag{1.9}$$

And we use *zeroth-order black-box* oracle: $x \mapsto \mathcal{O}(x) = \{f(x)\}$.

### 1.3.2 Grid Search

We consider the following simple algorithm, widely used, for example, for tuning hyperparameters in machine learning models.

The method generates a grid of points with a given density $p \geq 1$, and returns the best point among generated.

1. **Choose** $p \geq 1$ (an integer parameter of the method).
2. **Generate** $p^n$ points,

$$x_{(t_1,\ldots,t_n)} \quad = \quad \left[-\tfrac{p-1}{p} \cdot R + \tfrac{2R}{p} t_1 \ , \ \ldots \ , \ -\tfrac{p-1}{p} \cdot R + \tfrac{2R}{p} t_n\right],$$

   where $0 \leq t_i \leq p-1$ for each coordinate $1 \leq i \leq n$.
3. **Find** the point $\bar{x}$ with the smallest function value among all generated points.
4. **Return** $\bar{x}$.

To implement step 3 the algorithm needs only access to the values of $f$, which is provided by the zeroth-order oracle. We can prove the following simple result about this method.

**Theorem 1.3.1.** *Let $\bar{x}$ be the output of the grid search algorithm. Then,*

$$f(\bar{x}) - f^\star \quad \leq \quad \tfrac{LR}{p}. \tag{1.10}$$

*Consequently, to solve an instance of the problem from our problem class with an $\varepsilon$-accuracy in terms of the functional residual, it is enough to perform*

$$K \quad = \quad \left(\lfloor \tfrac{LR}{\varepsilon} \rfloor + 1\right)^n. \tag{1.11}$$

*zeroth-order oracle calls.*

*Proof.* By the definition of our grid, we cover the entire box $B$ with $p^n$ small boxes, and we generated all centers of these small boxes in step 2 of the algorithm. The side length of the initial box is $2R$, while the side length of each small box is $\frac{2R}{p}$. Hence, the radius in $\ell_\infty$-norm of each small box is $\frac{R}{p}$.

Since we cover the entire $B$, there exists a small box that contains a global solution $x^\star$. We denote the center of this small box by $\hat{x}$. It remains to note that

$$f(\bar{x}) - f^\star \quad = \quad f(\bar{x}) - f(x^\star) \quad \overset{\text{step 3}}{\leq} \quad f(\hat{x}) - f(x^\star) \quad \overset{(1.8)}{\leq} \quad L\|\hat{x} - x^\star\|_\infty \quad \leq \quad \tfrac{LR}{p},$$

where in the last inequality we used that $x^\star$ belongs to the small box. This proves (1.10).

To establish (1.11), it is enough to choose $p := \lfloor \frac{LR}{\varepsilon} \rfloor + 1 \geq \frac{LR}{\varepsilon}$. Substituting this bound into (1.10) gives $f(\bar{x}) - f^\star \leq \varepsilon$, which justifies the sufficient number of oracle calls. □

## Literature

For additional reading, we refer to Section 1.1 of [Nes18] and Sections 1-2 of [Nem95], the material on which these notes are largely based. Complexity Theory in optimization was initially developed by Nemirovski and Yudin in their seminal book [NY83].


[Nem95] Arkadi Nemirovski. *Information-based complexity of convex programming.* Lecture notes, 1995.

[Nes18] Yurii Nesterov. *Lectures on convex optimization.* Springer, 2018.

[NY83] Arkadi Nemirovski and David Yudin. *Problem complexity and method efficiency in optimization.* Wiley-Interscience, 1983.