
CMAP
École Polytechnique
Palaiseau, Spring 2023

# Proof techniques in first-order optimization

**Darya Todoskova**

**Supervisor: Baptiste Goujaud**

Bachelor thesis, Applied Mathematics

Double major: Mathematics and Computer Science

# Abstract

First-order convex optimization methods have recently gained in popularity both in theoretical optimization and in many scientific applications, such as signal and image processing, communications, machine learning, and many more. First-order methods, which in general involve very cheap and simple computational iterations, are often the best option to tackle such problems in a reasonable amount of time, when moderate accuracy solutions are sufficient. A standard way of analyzing optimization algorithms consists in worst-case analyses, which provides guarantees on the behavior of an algorithm (e.g. its convergence speed), that are independent of the function on which the algorithm is applied and true for every function in a particular class. In this work, we present the Performance Estimation Problem (PEP) framework, exploiting the definition of a worst case guarantee as the solution of an optimization problem itself. The aim of this thesis is to comprehend the fundamental concepts of worst-case analysis in optimization, the challenges in evaluating algorithm performance, and how the PEP framework addresses these challenges. We then use python to verify and establish the convergence guarantees of first-order iterative methods.

***Keywords*** – Gradient descent, rate of convergence, optimization, worst-case analyses

# Introduction

Many problems, including machine learning ones, ends up under the form of an optimization problem.

Essentially, we consider problems of the form

$$f_* \triangleq \min_{x \in \mathbf{R}^d} f(x) \tag{OPT}$$

where we denote an optimal solution by $x_* \in \arg\min_{x \in \mathbf{R}^d} f(x)$.
This motivates the study of optimization methods, including but not limited to Gradient Descent, Accelerated Gradient method, Heavy-ball method, Newton method or Bayesian optimization methods.

**First-order optimization.** Optimization methods are based on oracles such as derivatives. A method is called "$n^{th}$ order method" when it uses derivatives of the minimized function up to the $n^{th}$ order. For example, Bayesian optimization methods are $0^{th}$ order method because they do not require gradients and are then well-suited for hyper-parameters search. The Newton method requires Hessian evaluations and is then a $2^{nd}$ order method. In this work, we focus on first order methods, requiring only function values and gradients, such as Gradient descent, Accelerated Gradient method or Heavy-ball method.

**Worst-case analysis.** Knowing which algorithm to use depending on the situation is therefore crucial and requires a theoretical analysis. The most popular setting is called "worst-case analysis" and consists in obtaining a uniform guarantee over a set (or class) of function. That is, for a given algorithm $\mathcal{A}$ and a given class of functions $\mathcal{F}$, we are looking for guarantees achieved by $\mathcal{A}$ and that hold for any function $f$ of $\mathcal{F}$. Examples of such classes are the sets of convex functions, smooth functions, strongly convex and smooth functions or convex and Lipschitz continuous functions.

Obtaining a proof of worst-case convergence guarantee of an algorithm on a given class is sometimes very challenging. It consists in combining inequalities based on assumptions verified by the functions of the considered class, resulting in lengthy, complex, and sometimes technical proofs. Moreover, it is always hard to determine whether a proof is

tight or can be improved.

**Performance estimation problem and related work.** The performance estimation framework is a systematic method of creating worst-case convergence bounds. It defines the worst-case convergence bound as the solution of an optimization problem itself and solve it to obtain tight bounds.

The concept of a performance estimation problem was first introduced by Drori and Teboulle (3), who focused on the case of smooth convex functions with the performance criterion $f(x_N) - f_*$, where $x_N$ is the final iterate and $f_*$ is the optimal value. This formalism led to the discovery of "optimal" techniques such as (7). Drori and Teboulle suggested reducing PEP to a finite-dimensional problem using only the iterates, gradients, and function values, along with the optimal point and value. They analyzed several standard first-order algorithms, including the fixed-step gradient algorithm, the heavy-ball method, and the accelerated gradient method, and expressed PEP as a non-convex quadratic matrix program. To solve this program, they relaxed and dualized it, resulting in a convex problem that provided worst-case performance bounds.

This methodology was extended with the work of Adrien B Taylor and Glineur. Functional interpolation results are introduced to guarantee tightness of the semi-definite reformulation, that is, ensuring that the numerical solutions are indeed the smallest worst-case bounds. This framework allows performing worst-case analysis of first-order methods on smooth and strongly convex function in a principled way.

The discovery of new optimization methods led to the need for a unified framework that could produce tight analysis of a given algorithm. Drori and Teboulle made a first step in this direction, while Adrien B Taylor and Glineur formalized the idea of "PEP". A python tool has also recently been released to make using PEPs easier (see Goujaud et al.[1] and alternative in Matlab Adrien Taylor[2]).

**Contributions.** In this work, we detail the functioning of the Performance Estimation framework in Sections 1 to 4. To establish the fundamental concept and tools behind it, we choose to demonstrate the method on a simple example: bound guarantee after several steps of the gradient descent method on smooth and convex functions. In this context,

[1] https://pepit.readthedocs.io/en/latest/index.html
[2] https://github.com/PerformanceEstimation/Performance-Estimation-Toolbox

gradient descent is a practical example for two reasons. Firstly, it helps familiarizing the reader with the techniques and methodology required to analyze PEPs in a clearer manner, thereby making it easier to eventually tackle more complex approaches. Secondly, we can prove a precise performance bound for the gradient method, which is given analytically. Then we show that this strategy can be extended to more complex applications and provide some numerical results obtained on other algorithms in Section 5. Finally, we discuss the required ingredients needed for the PEP to work in Section 6.

**Definitions and notations.** We denote by $\langle \cdot, \cdot \rangle : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ the standard Euclidean inner product, and by $\| \cdot \|^2 : \mathbb{R}^d \to \mathbb{R}$ the standard Euclidean norm, i.e. the induced norm: for any $x \in \mathbb{R}^d : \|x\|^2 = \langle x, x \rangle$. The Euclidean norm of a vector $x \in \mathbb{R}^d$ is denoted as $\|x\|$. For two symmetric matrices $A$ and $B$, $A \succcurlyeq B$, $(A \succcurlyeq B)$ means $A - B \succcurlyeq 0 (A - B \succcurlyeq 0)$ is positive semi-definite (positive definite). Other notations are defined throughout the paper.

**Definition 0.1** (Convexity)**.** *A function $f$ is convex on $\mathbb{R}^d$ if and only if* $\forall x, y \in \mathbb{R}^d, \quad f(y) \geq f(x) + \langle \nabla f(x), y - x \rangle$.

**Definition 0.2** (Strong convexity)**.** *A function $f$ is $\mu$-strongly convex on $\mathbb{R}^d$ if and only if $f - \frac{\mu}{2}\|x\|^2$ is convex, i.e. if and only if* $\forall x, y \in \mathbb{R}^d, \quad f(y) \geq f(x) + \langle \nabla f(x), y - x \rangle + \frac{\mu}{2}\|y - x\|^2$.

**Definition 0.3** (Smoothness)**.** *A function $f$ is $L$-smooth if $\nabla f$ is $L$-Lipschitz continuous, that is* $\forall x, y \in \mathbb{R}^d, \quad \|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\|$.

**Lemma 0.4.** *When a function $f$ is $L$-smooth, it verifies $\forall x, y \in \mathbb{R}^d, f(x) \leq f(y) + \langle \nabla f(y), x - y \rangle + \frac{L}{2}\|x - y\|^2$.*

We denote by $\mathcal{F}_{\mu,L}$ the set of $\mu$ strongly convex and $L$ smooth functions. In particular, $\mathcal{F}_{0,L}$ denotes the set of convex and $L$-smooth functions, while $\mathcal{F}_{\mu,+\infty}$ denotes the set of $\mu$ strongly convex functions.

# Contents

**1 Chapter 1 : From Worst-Case Analysis to Performance Estimation Problems (PEP) 1**
1.1 First-order methods and worst-case analysis . . . 1
1.1.1 First-order Methods . . . 1
1.1.2 Worst-case Analysis . . . 1
1.2 An Example of Worst-case Proof . . . 2
1.3 Formal Definition of the Tight Convergence Guarantee . . . 5

**2 Chapter 2 : From Infinite to Finite Dimension : Interpolation Conditions 6**
2.1 Convex interpolation . . . 7
2.2 Smooth convex interpolation . . . 9
2.2.1 Applying classical interpolation conditions . . . 9
2.2.2 Smooth convex interpolation conditions . . . 11

**3 Chapter 3 : Quadratic Reformulation 14**

**4 Chapter 4 : Semi-definite Reformulation 15**

**5 Chapter 5 : Numerical Illustrations 17**
5.1 Numerical Worst-case Guarantee for Gradient Descent . . . 17
5.2 Other Examples . . . 18
5.2.1 Accelerated Gradient Method on Smooth Strongly Convex Function 18
5.2.2 Heavy Ball Momentum on Smooth Strongly Convex Function . . . 19

**6 Conclusion 22**

**References 24**

**Appendices 25**
A1 Gradient Descent in PEPit . . . 25
A2 Accelerated Gradient Method in PEPit . . . 27
A3 Heavy-Ball Method in PEPit . . . 29
A4 Heavy-Ball Method : representation of a region of convergence guaranteed by the PEP framework. . . . 31

# 1 Chapter 1 : From Worst-Case Analysis to Performance Estimation Problems (PEP)

In this chapter, we will present the fundamental components that make up the performance estimation method. To demonstrate, we will use gradient descent as a case study for analysis.

## 1.1 First-order methods and worst-case analysis

### 1.1.1 First-order Methods

We consider unconstrained minimization problems involving a given class of objective functions, and only treat first-order methods. This means that the method can only gather information about the objective function using a "black box" (that is able to produce a solution for any instance of a given computational problem). In our case, it returns first-order information about specific points, i.e. $\{x, \nabla f(x), f(x)\}$. In other words, for a given algorithm $\mathcal{A}$, we use the gradient and the function value at each point, i.e. $x_{k+1} = \mathcal{A}(x_0, g_0, f_0, x_1, g_1, f_1, ..., x_k, g_k, f_k)$.

**Example 1.1** (Gradient descent)**.** *The Gradient descent method is defined by the update*

$$x_{k+1} = x_k - \gamma \nabla f(x_k). \tag{GD}$$

### 1.1.2 Worst-case Analysis

In worst-case analysis, we want a proof of convergence of an algorithm $\mathcal{A}$ with a rate $\rho$ that holds for all functions of a given class $\mathcal{F}$. In order to measure the performance of a given method on a specific function $f$, we will use a performance criterion $\mathcal{P}$ to be minimized. Moreover, this performance is necessarily impacted by the quality of the initialisation, hence we have to use an initialisation term to be compared with the performance metric.

**Example 1.2.** *1. Gradient Descent on the class $\mathcal{F}_L$ of smooth convex functions verifies* $f(x_k) - f_* \leq \frac{L}{k}\|x_0 - x_*\|^2$.

2. *Gradient Descent on the class $\mathcal{F}_{\mu,L}$ of smooth strongly-convex functions verifies* $\|x_k - x_*\|^2 \leq \left(\frac{L-\mu}{L+\mu}\right)^{2k} \|x_0 - x_*\|^2$.

For the sake of our Gradient Descent example, we use distance to an optimal solution $\|x_k - x_*\|$, with $x_*$ defined as the minimizer of $f$; and objective function accuracy $f_k - f_*$. We also consider the class $\mathcal{F}_{0,L}$ of smooth convex functions, or $\mathcal{F}_{\mu,L}$ of smooth strongly convex functions, over which we want to estimate the worst-case performance of the method (over several iterations in our gradient descent analysis). The initialisation term is the distance of the first iterate to the optimum $\|x_0 - x_*\|^2$.

## 1.2 An Example of Worst-case Proof

In this section, we look at the construction of a worst-case proof (c.f. Waldspurger). Since $f$ is a smooth convex function, we can prove that $(f(x_k))_{k\in\mathbb{N}}$ converges to $f(x_*)$. We will thus find the rate of convergence $\rho_k$ . We proceed as follows.

**Definition 1.3** (Descent Lemma)**.** *Let $f$ be $L$-smooth, for some $L > 0$. We consider gradient descent with stepsize $\gamma_k = \frac{1}{L}$ for all $k$. Then, for any $k$,*

$$f(x_{k+1}) \leq f(x_k) - \frac{1}{2L}\|\nabla f(x_k)\|^2. \tag{1.1}$$

*Proof.* The Gradient descent update $x_{k+1} = x_k - \frac{1}{L}\nabla f(x_k)$ together with the smoothness inequality (0.4) leads to

$$\begin{aligned} f(x_{k+1}) \leq & f(x_k) + \langle \nabla f(x_k), x_{k+1} - x_k\rangle + \tfrac{L}{2}\|x_{k+1} - x_k\|^2 \\ \leq & f(x_k) + \left\langle \nabla f(x_k), -\tfrac{1}{L}\nabla f(x_k)\right\rangle + \tfrac{L}{2}\| - \tfrac{1}{L}\nabla f(x_k)\|^2 \\ \leq & f(x_k) - \tfrac{1}{L}\|\nabla f(x_k)\|^2 + \tfrac{1}{2L}\|\nabla f(x_k)\|^2 \\ \leq & f(x_k) - \tfrac{1}{2L}\|\nabla f(x_k)\|^2. \end{aligned}$$

□

Let $f$ be convex and $L$-smooth, for some $L < 0$. We consider gradient descent with constant step-size $\frac{1}{L}$ for all $k$.

We will first show that the sequence of iterates gets closer to $x_*$ at each iteration:

$$\forall k \in \mathbb{N}, \quad \|x_* - x_{k+1}\| \leq \|x_* - x_k\|$$

- Let $k$ be fixed. By the convexity of $f$, we have that

$$\begin{aligned} f(x_*) &\geq f(x_k) + \langle \nabla f(x_k), x_* - x_k \rangle \\ &= f(x_k) + L\langle x_k - x_{k+1}, x_* - x_k \rangle \end{aligned}$$

- By definition, $f(x_*) \leq f(x_{k+1})$. Moreover, by the $L$-smoothness of $f$ (see Descent Lemma 1.1, which is a key inequality in this proof), we have :

$$\begin{aligned} f(x_*) &\leq f(x_k) - \frac{1}{2L}\|\nabla f(x_k)\|^2 \\ &= f(x_k) - \frac{L}{2}\|x_k - x_{k+1}\|^2 \end{aligned}$$

We have thus found upper and lower bounds for $f(x_*)$. By combining both, we obtain :

$$f(x_k) + L\langle x_k - x_{k+1}, x_* - x_k \rangle \leq f(x_*) \leq f(x_k) - \frac{L}{2}\|x_{k+1} - x_k\|^2$$

Implying that

$$\langle x_k - x_{k+1}, x_* - x_k \rangle + \frac{1}{2}\|x_{k+1} - x_k\|^2 \leq 0$$

Which is equivalent to

$$\|x_* - x_{k+1}\|^2 \leq \|x_* - x_k\|^2$$

Now, we find an inequality that links $f(x_{k+1}) - f_*$ and $f(x_t) - f*$ which, after unrolling through iterations, will give us the desired result.

From the Descent Lemma 1.1, we have

$$f(x_{k+1}) - f_* \leq f(x_k) - f_* - \frac{1}{2L}\|\nabla f(x_k)\|^2 \tag{1.2}$$

Moreover, by convexity, as mentioned earlier, we get :

$$f(x_k) - f_* \leq \langle \nabla f(x_k), x_k - x_* \rangle$$

By the Cauchy-Schwarz inequality,

$$f(x_k) - f_* \leq \|\nabla f(x_k)\|\|x_k - x_*\| \leq \|\nabla f(x_k)\|\|x_0 - x_*\|$$

In other words, $\|\nabla f(x_k)\| \geq \frac{f(x_k)-f_*}{\|x_0-x_*\|}$, which we inject into 1.2 :

$$f(x_{k+1}) - f_* \leq f(x_t) - f_* - \frac{1}{2L}\frac{(f(x_k) - f_*)^2}{\|x_0 - x_*\|^2}$$

Since $\frac{1}{0} = +\infty$ and $\frac{1}{1-x} \geq 1 + x$ for any $x \in [0; 1]$ we get

$$\begin{aligned}
\frac{1}{f(x_{k+1}) - f_*} &\geq \frac{1}{f(x_k) - f_*} \times \frac{1}{1 - \frac{1}{2L}\frac{f(x_t)-f_*}{\|x_0-x_*\|^2}} \\
&\geq \frac{1}{f(x_k) - f_*}\left(1 + \frac{1}{2L}\frac{f(x_k) - f_*}{\|x_0 - x_*\|^2}\right) \\
&= \frac{1}{f(x_k) - f_*} + \frac{1}{2L\|x_0 - x_*\|^2}.
\end{aligned}$$

Hence, for any $k \in \mathbb{N}$,

$$\frac{1}{f(x_k) - f_*} \geq \frac{1}{f(x_0) - f_*} + \frac{k}{2L\|x_0 - x_*\|^2}.$$

According to the Descent Lemma 1.1, and the fact that $\nabla f_* = 0$, we have:

$$f(x_0) - f_* \leq \frac{L}{2}\|x_0 - x_*\|^2$$

Which implies that:

$$\begin{aligned}
\frac{1}{f(x_k) - f_*} &\geq \frac{2}{L\|x_0 - x_*\|^2} + \frac{k}{2L\|x_0 - x_*\|^2} \\
&= \frac{k+4}{2L\|x_0 - x_*\|^2}
\end{aligned}$$

Finally, we get:

$$f(x_k) - f_* \leq \frac{2L}{k+4}\|x_0 - x_*\|^2 \tag{1.3}$$

Taking $||x_0 - x_*||$ as a constant, $f(x_k) - f_* = O(\frac{1}{k})$.

As mentioned in the introduction, after such a computation, questions may arise regarding the optimality of the convergence rate. Is this solution the most optimal one? Did we use

the right inequalities? Are there generic rules to follow when choosing the inequalities and combining them? Or is there a unique way to reach the optimal result? These are the main questions we wish to overcome in the following sections.

## 1.3 Formal Definition of the Tight Convergence Guarantee

Let's formally define the problem of looking for a tight rate of convergence $\rho$. We fix a function class $\mathcal{F}$ and an algorithm $\mathcal{A}$ and we denote iterates by $x_0, ..., x_k$. We look for a certain $\rho$ such that

$$\forall f \in \mathcal{F}, \quad \mathcal{P}(x_k) \leq \rho I(x_0) \tag{1.4}$$

where $P(x_k)$ is a performance metric evaluated in the last iterate $x_k$ and $I(x_0)$ is an initial value of comparison applied on the initial iterate $x_0$. Most often, we see $\|x_{k+1} - x_*\|^2 \leq \rho\|x_k - x_*\|$ (distance to optimality) or even $f(x_{k+1}) - f_* \leq \rho(f(x_k) - f_*)$ (function values).

This brings us to claim that any $\rho$ satisfies (1.4) if and only if $\rho$ is an upper bound of

$$\left\{ \frac{P(x_k)}{I(x_0)} \mid f \in \mathcal{F} \text{ and } x \text{ is generated by } \mathcal{A} \text{ on } f \right\}.$$

Therefore, the tightest such $\rho$ is defined as the smallest upper bound of the above mentioned set, i.e.

$$\rho = \sup_{d,f,x_*,x} \frac{P(x_k)}{I(x_0)}$$

$$\text{s.t.} \quad \begin{cases} f \in \mathcal{F} \\ x \text{ is generated by } \mathcal{A} \text{ on } f \\ \nabla f(x_*) = 0 \end{cases} \tag{1.5}$$

Hence, we need to solve this PEP(1.5). For the sake of simplicity, we derive a solution to this problem on a simple example. Note that this problem is hard to solve for many reasons. The most obvious one is the fact that this problem is infinite-dimensional. In the following section, we show how to reformulate this problem in finite dimension.

# 2 Chapter 2 : From Infinite to Finite Dimension : Interpolation Conditions

Let's now apply the PEP (1.5) on the simple example of Gradient Descent on a class $\mathcal{F}_{0,L}$ of smooth convex functions, with the performance metric $f(x_k) - f(x_*)$ and an initial value of comparison $\|x_0 - x_k\|$ on the initial iterate $x_0$.

Recall that the goal is to compute the smallest $\rho$ (rate of convergence) possible such that the inequality 1.4 is valid. Hence, we have that

$$\rho = \sup_{d,f,x_*,x_0} \frac{f(x_k) - f(x_*)}{\|x_0 - x_*\|^2}$$

$$\text{s.t.} \quad \begin{cases} f \in \mathcal{F}_L \\ x_{k+1} = x_k - \dfrac{1}{L}\nabla f(x_k) \\ \nabla f(x_*) = 0 \end{cases} \tag{2.1}$$

where $d, f, x_*, x_0$ are the variables of the optimization problem.

Notice that because it involves an unknown function $f$ as a variable, the optimization problem is indeed infinite-dimensional. We are thus looking for the worst possible problem instance, that is a function $f$ and an initial point $x_k$ within a given class of problems.

Nevertheless, using the black-box property of the method (and of the performance criterion), we observe that a completely equivalent finite-dimensional problem can be obtained by restricting the function $f$ to the information of the solution of its black box (that is, only the iterates $\{x_i\}_{i\in I}$, the function values $\{f_i\}_{i\in I}$ and the gradients $\{g_i\}_{i\in I}$). In other words, we are currently in infinite dimension, but if we take two different functions that have the same values $\{x_i, g_i, f_i\}_{i\in I}$, we get the same ratio $\frac{f(x_k)-f(x_*)}{\|x_0-x_*\|^2}$. We thus do not need information about the function as a whole, but solely at each iterate.

This being mentioned, we can reformulate the problem as follows:

$$\rho = \sup_{d,x_*,x_0,g_0,\dots,g_k,f_0,\dots,f_k} \frac{f(x_k) - f(x_*)}{\|x_0 - x_*\|^2}$$

$$\text{s.t.} \quad \exists f \in \mathcal{F}_L \quad \textit{such that} \quad \begin{cases} f_i = f(x_i) & i = k, * \\ g_i = g(x_i) & j = k, * \end{cases} \tag{2.2}$$
$$x_{k+1} = x_k - \frac{1}{L} g_k$$
$$g_* = 0$$

As explained above, this problem is strictly equivalent to the original 2.1 in terms of optimal value, since every solution to 2.2 can be interpolated by a solution of 2.1 and, reciprocally, every solution of 2.1 can be discretized to provide a solution to 2.2.

The essential *and problematic* part of this reformulation is the first constraint requiring that $\{x_i, g_i, f_i\}_{i \in I}$ should be interpolated by a function belonging to the class $\mathcal{F}_L$. However, the problem does not seem to have improved due to the existence constraint. We refer to this as a discrete interpolation problem : given a set of triplets {(coordinate, gradient, function value)} can we recover a function within a class that explains those triplets?

In order to get rid of the existence constraint, we look for the conditions that the set of points must verify in order for there to exist a function that interpolates them. This problem is often referred to as an "interpolation problem". In the following, we provide some examples and highlight why this is not a trivial problem.

## 2.1 Convex interpolation

In order to tackle this interpolation problem by building interpolation conditions for the class of smooth convex functions, we start by doing so for the simpler class of convex functions without smoothness assumption $\mathcal{F}_{0,\infty}$, for which we can obtain a simple solution.

Given a convex function, an index set $I$, and any set $(x_i, g_i, f_i)_{i \in I}$ of the form {(coordinate,(sub-)gradient,function value)}, we verify the convex interpolation result:

**Theorem 2.1.**

$$\exists f \in \mathcal{F}_{0,\infty} : \quad g_k \in \partial f(x_k) \quad \textit{and} \quad f_k = f(x_k) \quad \forall k \in I$$

$$\Leftrightarrow$$

$$f_i \geq f_j + \langle g_j, x_i - x_j \rangle \quad \forall i, j \in I$$

*Proof.* (Necessity) Assume there exists a convex function $f : \mathbb{R}^d \to \mathbb{R}$ such that $f_i = f(x_i)$ and $g_i \in \partial f(x_i), \forall i \in I$. By definition of a gradient, we simply have

$$f_i \geq f_j + \langle g_j, x_i - x_j \rangle$$

(Sufficiency) Given a set of triplets $(x_i, g_i, f_i)_{i \in I}$ satisfying the previous inequality for all $i, j \in I$, one can obtain a convex function by the geometric construction:

$$f(x) = \max_{i \in I} \{f_i + \langle g_i, x - x_i \rangle\}$$

We can see that $f$ is a convex function as the maximum of convex functions.

Now, let's show that $f$ indeed interpolates the set of points.
Assume that $\forall i, j \in I$, we have $f_i \geq f_j + \langle g_j, x_i - x_j \rangle$.
By definition of the maximum, we get $f(x_i) = \max_{j \in I} \{f_j + \langle g_j, x_i - x_j \rangle\} \geq f_i$. Hence, $f(x_i) \geq f_i$.
Moreover, according to our assumption: $f(x_i) = \max_{j \in I} \{f_j + \langle g_j, x_i - x_j \rangle\} \leq f_i$ by construction. Hence, $f(x_i) \leq f_i$.
We can thus conclude that $f(x_i) = f_i$ and that ensures that by taking the maximum of each tangent line $f_j + \langle g_j, x - x_j \rangle$ for all $j \in I$, the line at $x_i$ needs to be above all of the others.

Furthermore, this construction also implies that $g_i \in \partial f(x_i)$ for all $i \in I$. Indeed,

$$\begin{aligned} f(x) &= \max_{i \in I} \{f_i + \langle g_i, x - x_i \rangle\} \\ &\geq f_i + \langle g_i, x - x_i \rangle \\ &\geq f(x_i) + \langle g_i, x - x_i \rangle \end{aligned}$$

concluding that $g_i \in \partial f(x_i)$. □

## 2.2 Smooth convex interpolation

We saw in the previous section that interpolation conditions for the class of convex functions is obtained by discretizing 0.1. Naively, we could think that a good candidate for interpolation conditions on the class of $L$-smooth convex functions could be obtained by discretizing both 0.1 and 0.3.

### 2.2.1 Applying classical interpolation conditions

Indeed, let us assume that $\{x_i, g_i, f_i\}_{i\in I}$ verifies the following discrete inequalities of convexity and smoothness, respectively:

$$\forall i, j \in I, \quad f_i \geq f_j + \langle g_j, x_i - x_j \rangle \tag{2.3}$$

$$\forall i, j \in I, \quad \|g_i - g_j\| \leq L\|x_i - x_j\| \tag{2.4}$$

Then there is not necessarily a smooth convex function interpolating those points as shown in the following example. Let $f$ be the green maximum function:

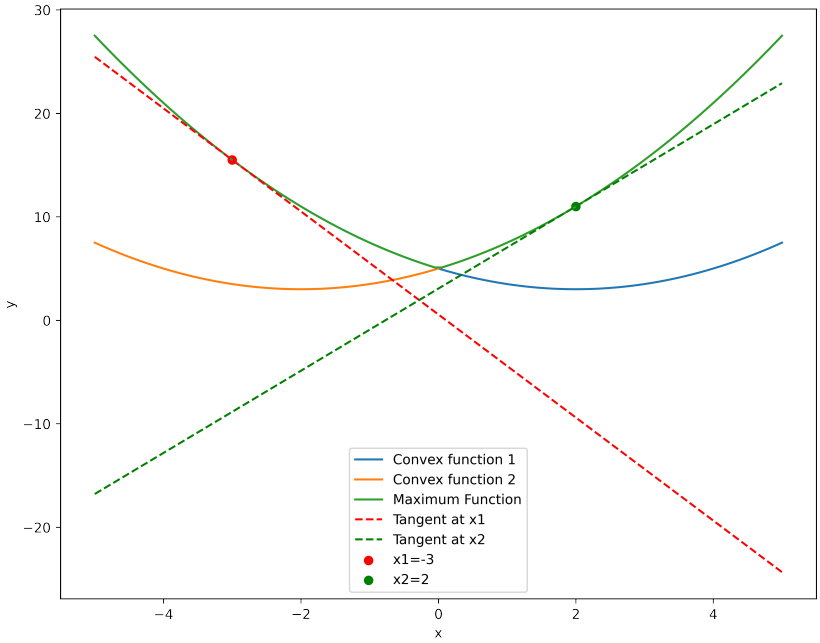


**Figure 2.1:** Maximum Function of Two Random Convex Functions

As seen on the figure above, our function $f$ has to verify two constraints:

- It has to interpolate the set of points $\{x_i, g_i, f_i\}_{i\in I}$.

- It has to be smooth and convex. This implies that $f$ always has to be on top of its tangent lines : $f(x) \geq \max_{i \in I}\{f_i + \langle g_i, x - x_i \rangle\} \quad , \forall x \in \mathbb{R}^d$.

We now see why our function $f$ illustrates a counter-example as to why discrete interpolation conditions (as seen in section 4.1) aren't strong enough to work for all smooth convex functions. Let's interpret it graphically:

We know that according to the two constraints above, the function $f$ must remain above the red tangent line while interpolating the points $x_1$ and $x_2$ (respecting their associated gradients). In order for $f$ to do so, it would have to be curved faster than the orange line. Indeed, at some point, we would need to straighten out the gradient quickly enough to interpolate $x_2$ with its given gradient. Due to the maximum curvature constraint given by the orange line, the gradient of $f$ should not go through a strong variation change (its Hessian must be bounded), which is not possible. Indeed, the gradient of a function tells us how the function changes when we move in a particular direction from a point. If the gradient were to change arbitrarily quickly, the old gradient would not be informative enough even if we take a small step. Smoothness thus ensures that the gradient cannot change too quickly — therefore the gradient information will be informative within a region around where it is taken. This implies that we can decrease the function's value by moving the direction opposite of the gradient. Since this is the case, $f$ cannot stay above the red tangent line all the while interpolating $x_1, x_2$ - even though they verify the discrete interpolation conditions 2.3 and 2.4.

Let's now explain this counter-example in a more formal and mathematical way:

Let $L > 0$. Consider $(x_0, g_0, f_0) = (0, -1, 0)$ and $(x_1, g_1, f_1) = (\frac{2}{L}, 1, \frac{2}{L})$. We verify easily the two inequalities 2.3 and 2.4. However, $-1$ and $1$ are both sub-gradients of $f$ in 0, hence $f$ cannot be differentiable, therefore cannot be smooth.

We can deduce that stronger interpolation conditions are required for convex smooth functions. Indeed, if points of the function verify both of the discretized convexity and smoothness conditions independently, it isn't enough to guarantee the existence of a smooth convex function that interpolates the set of points. Which brings us to the co-coercivity inequality for smooth convex functions.

### 2.2.2 Smooth convex interpolation conditions

Consequently, this section deals with developing a necessary and sufficient condition for the existence of a smooth convex function interpolating through a given set of data triplets $\{x_i, g_i, f_i\}_{i\in I}$, i.e. deciding whether there exists a smooth convex function $f$ such that $f(x_i) = f_i$ and $g_i \in \nabla f(x_i)$ for all $i \in I$.

We saw in the previous section that using discretized versions of 0.1 and 0.3 is not enough to guarantee the existence of such a function. We then establish the following inequality known as co-coercivity.

**Theorem 2.2** (Co-coercivity)**.** *A function $f$ is smooth convex if and only if*

$$\forall x, y, \quad f(y) \geq f(x) + \langle \nabla f(x), y - x\rangle + \frac{1}{2L}\|\nabla f(y) - \nabla f(x)\|^2. \tag{2.5}$$

*Proof.* Let's first show that $f$ is convex and $L$-smooth if and only if

$$\forall x, y, z, \quad f(y) + \langle \nabla f(y), x - y\rangle \leq f(z) + \langle \nabla f(z), x - z\rangle + \frac{L}{2}\|x - z\|^2. \tag{2.6}$$

- Assume that $f$ is convex and $L$-smooth. By 0.1 and 0.3, we get :

  $$\forall x, y, z, \quad f(y) + \langle \nabla f(y), x - y\rangle \leq f(x) \leq f(z) + \langle \nabla f(z), x - z\rangle + \frac{L}{2}\|x - z\|^2.$$

  Thus resulting in the desired inequality 2.6.

- Reciprocally, assume that 2.6 holds for all $x, y, z$. Then, for $x = y$, we get smoothness:

  $$\forall x, z, \quad f(x) \leq f(z) + \langle \nabla f(z), x - z\rangle + \frac{L}{2}\|x - z\|^2$$

  and for $x = z$, we get convexity :

  $$\forall x, y, \quad f(y) + \langle \nabla f(y), x - y\rangle \leq f(x).$$

Now, let's finally show that $f$ is convex and $L$-smooth if and only if

$$\forall y, z, \quad f(z) \geq f(y) + \langle \nabla f(y), z - y\rangle + \frac{1}{2L}\|\nabla f(z) - \nabla f(y)\|^2$$

By 2.6, $f$ is convex and $L$-smooth if and only if

$$\forall x, y, z, \quad 0 \leq f(z) - f(y) + \langle \nabla f(y), y \rangle - \langle \nabla f(z), z \rangle + \langle \nabla f(z) - \nabla f(y), x \rangle + \frac{L}{2}\|x - z\|^2$$

thus, if and only if

$$\forall y, z, \qquad 0 \leq f(z) - f(y) + \langle \nabla f(y), y - z \rangle + \min_{x \in \mathbb{R}^d} (\langle \nabla f(z) - \nabla f(y), x - z \rangle + \frac{L}{2}\|x - z\|^2).$$

equivalent to

$$\forall y, z, \qquad 0 \leq f(z) - f(y) + \langle \nabla f(y), y - z \rangle - \frac{1}{2L}\|\nabla f(z) - \nabla f(y)\|^2$$

resulting in 2.5. □

This stronger inequality is particularly useful in the context of interpolation problem due to the following theorem provided by (1).

**Theorem 2.3.** *For any triplets $(x_i, g_i, f_i)$, there exists an $L$-smooth convex function $f$ verifying*

$$f(x_i) = f_i$$
$$g(x_i) = g_i$$

*if and only if*

$$f_i \geq f_j + \langle g_j, x_i - x_j \rangle + \frac{1}{2L}\|g_i - g_j\|^2. \tag{2.7}$$

Thus, the PEP (1.5) is equivalently written as:

$$\rho = \sup_{d, x_*, x_0, g_0, \ldots, g_k, f_0, \ldots, f_k} \frac{f_k - f_*}{\|x_0 - x_*\|^2}$$

$$\text{s.t.} \quad f_i \geq f_j + \langle g_j, x_i - x_j \rangle + \frac{1}{2L}\|g_i - g_j\|^2 \quad , \forall (i, j) \in \{*, 0, 1, \ldots, k\}^2 \tag{2.8}$$
$$x_{i+1} = x_i - \frac{1}{L} g_i \quad , i = 0, 1, \ldots, k$$
$$g_* = 0$$

Notice that the constraints are linear in $f_i$ but quadratic in $x_i$ and $g_i$. This equivalent reformulation gives us the set of conditions guaranteeing the existence of a smooth convex interpolating function. It is the main key that has just ensured an exact convex rephrasing of the PEP in order to transition from infinite to finite dimension.

# 3 Chapter 3 : Quadratic Reformulation

We notice is the current formulation, that the minimized objective is a ratio of terms that depend on the optimization variables. Thus, it is non convex and then not necessarily easy to optimize. In this section, we convexify the objective.

To do so, we remove the denominator (the distance between the initial point $x_0$ and the optimal solution $x_*$) in the objective function and bound it by a constant. Indeed, this reformulation is done using homogeneity arguments provided by the norm operator. For instance, for a certain matrix $A$ and a vector $x \neq 0$, $\|A\| = \sup_x \frac{\|Ax\|}{\|x\|}$ is maximized for $\|x\| = 1$ due to homogeneity of the formula. We thus choose to set $\|x\|$ to any given constant, since dividing $x$ by its own norm also results in maximizing the ratio (a classic reasoning involving appropriate scaling operations). This explains why removing the denominator and bounding it by 1 (or equal to 1) ensures an equivalence between 3.1 and 2.8.

The problem can thus be reformulated as quadratic in $x_i, g_i$ and linear in $f_i$:

$$\begin{aligned} \rho = \sup_{d, x_*, x_0, g_0, \dots, g_k, f_0, \dots, f_k} \quad & f_k - f_* \\ \text{s.t.} \quad & f_i \geq f_j + \langle g_j, x_i - x_j \rangle + \frac{1}{2L}\|g_i - g_j\|^2 \quad , i \neq j = *, 0, 1, \dots, k \\ & \|x_0 - x_*\|^2 \leq 1 \end{aligned} \tag{3.1}$$

We can see that the objective function is convex, whereas the constraints are not. Indeed, the non-convexity of this problem is given by the maximization of the objective function, as well as the product $\langle g_j, x_i - x_j \rangle$ among the $(k+1)^2$ constraints.

We can now reformulate the problem once again by looking for a completely equivalent one through semi-definite programming.

# 4 Chapter 4 : Semi-definite Reformulation

In this step, our goal is to transform the non-convex optimization problem into an equivalent semi-definite programming (SDP) problem. We want to linearize our problem 3.1, which is linear in terms of $f_i$, but quadratic in terms of $x_i$ and $g_i$.

The basic idea of semi-definite reformulation is to linearize the problem by expressing the objective function and constraints of the original optimization problem in terms of a semi-definite matrix, which is a square matrix that contains all the quadratic terms; and a matrix containing the linear function values. By doing so, the original problem can be reformulated as a SDP problem that is now linear (both the objective function and constraints) and convex, and provides worst-case performance bounds.

To do so, we naturally reformulate our problem using

$$F = \begin{pmatrix} f_0 - f_* & f_1 - f_* & \dots & f_k - f_* \end{pmatrix}$$

and a $(k+2) \times (k+2)$ SDP matrix $G \succcurlyeq 0$ defined as:

$$G = \begin{bmatrix} \|x_0 - x_*\|^2 & \langle x_0 - x_*, g_0 \rangle & \dots & \langle x_0 - x_*, g_k \rangle \\ \langle x_0 - x_*, g_0 \rangle & \|g_0\|^2 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ \langle x_0 - x_*, g_{k-1} \rangle & \ddots & \ddots & \langle g_{k-1}, g_k \rangle \\ \langle x_0 - x_*, g_k \rangle & \dots & \langle g_{k-1}, g_k \rangle & \|g_k\|^2 \end{bmatrix}$$

This works since our previous problem is linear in terms of the entries of the Gram matrix and $f_i, f_*$.

Using these change of variables, we arrive to

$$\begin{aligned} \rho = \sup_{F,G \succeq 0} \; & F_{k+1} \\ \text{s.t.} \quad & F_{i+1} \geq F_{j+1} - \gamma \sum_{l<i} G_{j+2,l+2} + \gamma \sum_{l<j} G_{j+2,l+2} \\ & \qquad + \frac{1}{2L}(G_{i+2,i+2} + G_{j+2,j+2} - G_{i+2,j+2} - G_{j+2,i+2}) \quad , i \neq j = *, 0, 1, ..., k \\ & G_{1,1} \leq 1 \end{aligned}$$

which can be solved numerically. Analytically, we obtain the tight theoretical guarantee

$$\rho = \frac{L}{4k+2} \tag{4.1}$$

as seen in (3). This conjecture is supported by the Figure 5.1 that has been built using the PEP framework.

In the next section, we show how to perform this analysis using PEPit, which automates these procedures.

# 5 Chapter 5 : Numerical Illustrations

Through this thesis, particularly our case study of Gradient Descent, we managed to cast the problem of performing a worst-case analysis (PEP) as a semi-definite program. In addition, a way to implement the PEP while avoiding all the heavy semi-definite programming steps is through PEPit (6). As previously mentioned in the related work part of the introduction, all the tedious work we have just done on PEP can be automatized in Python, using PEPit - a package enabling computer-assisted worst-case analyses of first-order optimization methods.

In this section, we compare the worst-case guarantees obtained with PEPit with reference worst-case guarantees for different examples of algorithms.

## 5.1 Numerical Worst-case Guarantee for Gradient Descent

For our case study on gradient descent, we run code that generates a worst-case scenario for iterations of the method, applied to the minimization of a smooth convex function $f$ (see A1). We plot numerical worst-case guarantees on $f(x_k) - f(x_*)$ with initial condition $\|x_0 - x_*\| \leq 1$ as a function of $k$, for $L = 1$.

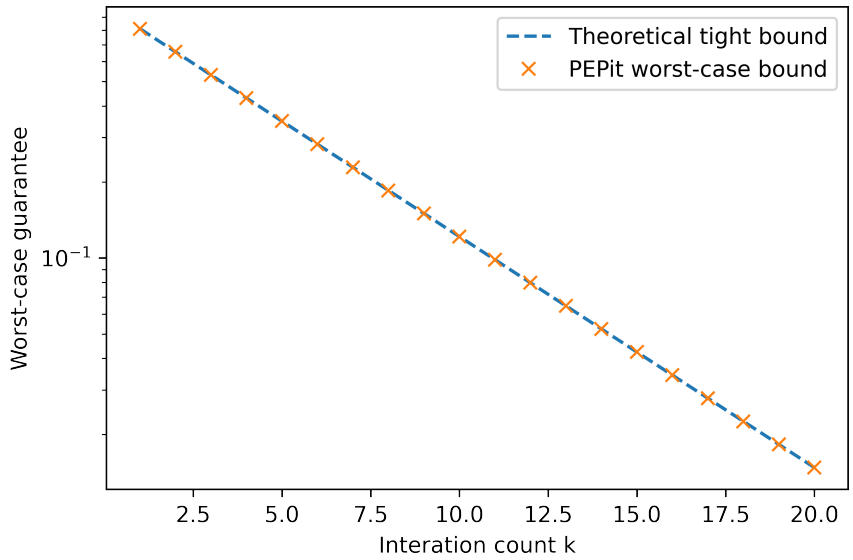


**Figure 5.1:** Gradient Descent : theoretical and PEPit (numerical) worst-case performance bounds as functions of the iteration count

This simple example allows to observe that numerical values obtained from PEPit match the worst-case guarantee given in 4.1.

## 5.2 Other Examples

Since we obtained a worst-case guarantee for gradient descent using the PEPit package, let's see other (more advanced) examples. We compare numerical worst-case bounds from PEPit to reference theoretical worst-case guarantees for three different optimization methods. For simplicity, we fix the smoothness parameter to $L = 1$.

### 5.2.1 Accelerated Gradient Method on Smooth Strongly Convex Function

For this example, we consider the convex minimization problem $f_* = min_x f(x)$ where $f$ is $L$-smooth and $\mu$-strongly convex.

**Algorithm 1** Accelerated Gradient for Strongly Convex Objective

For $i$ in $\{0, ..., k-1\}$,

$$y_i = x_i + \frac{\sqrt{L} - \sqrt{\mu}}{\sqrt{L} + \sqrt{\mu}}(x_i - x_{i-1})$$

$$x_{i+1} = y_i - \frac{1}{L}\nabla f(y_i)$$

with $x_{-1} := x_0$.

We run the PEPit code A2 which computes a worst-case guarantee for this method. It thus computes the smallest possible $\rho_k$ such that

$$f(x_k) - f_* \leq \rho_k(f(x_0) - f(x_*) + \frac{\mu}{2}\|x_0 - x_*\|_2^2)$$

is valid for all the functions of $\mathcal{F}_{\mu,L}$. The theoretical upper guarantee is given by (4) :

$$f(x_k) - f_* \leq (1 - \sqrt{\frac{\mu}{L}})^k(f(x_0) - f(x_*) + \frac{\mu}{2}\|x_0 - x_*\|_2^2).$$

We thus plot numerical worst-case guarantees of $f(x_k) - f_*$ with initial condition $f(x_0) - f(x_*) + \frac{\mu}{2}\|x_0 - x_*\|^2 \leq 1$, as a function of $k$, with $\mu = 0.1$.

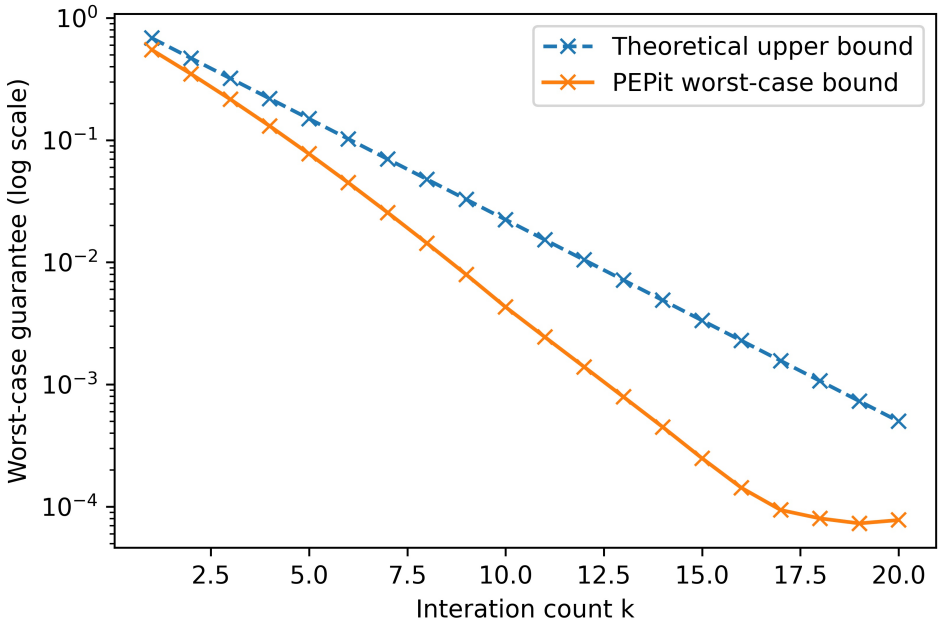


**Figure 5.2:** Accelerated Gradient Method : theoretical and PEPit (numerical) worst-case performance bounds as functions of the iteration count

We observe that this guarantee still has room for improvement in order to better resemble the worst-case behavior of the PEPit algorithm.

### 5.2.2 Heavy Ball Momentum on Smooth Strongly Convex Function

For this example, we consider the convex minimization problem $f_* = min_x f(x)$ where $f$ is $L$-smooth and $\mu$-strongly convex.

**Algorithm 2** Heavy-Ball Momentum

$$x_{k+1} = x_k - \alpha \nabla f(x_k) + \beta(x_k - x_{k-1})$$

with

$$\alpha \in [0, \frac{1}{L}]$$

and

$$\beta = \sqrt{(1 - \alpha\mu)(1 - L\alpha)}$$

We run the PEPit code A3 which computes a worst-case guarantee for the Heavy-Ball

method. It thus computes the smallest possible $\rho_k$ such that

$$f(x_k) - f_* \leq \rho_k(f(x_0) - f(x_*))$$

is valid for all the functions of $\mathcal{F}_{\mu,L}$. The theoretical upper guarantee is given by (5) :

$$f(x_k) - f_* \leq (1 - \alpha\mu)^k(f(x_0) - f(x_*)).$$

We thus plot numerical worst-case guarantees of $f(x_k) - f_*$ with initial condition $f(x_0) - f(x_*) \leq 1$, as a function of $k$, with $\mu = 0.1$, $\alpha = \frac{1}{2L}$, and $\beta = \sqrt{(1-\alpha\mu)(1-L\alpha)}$.

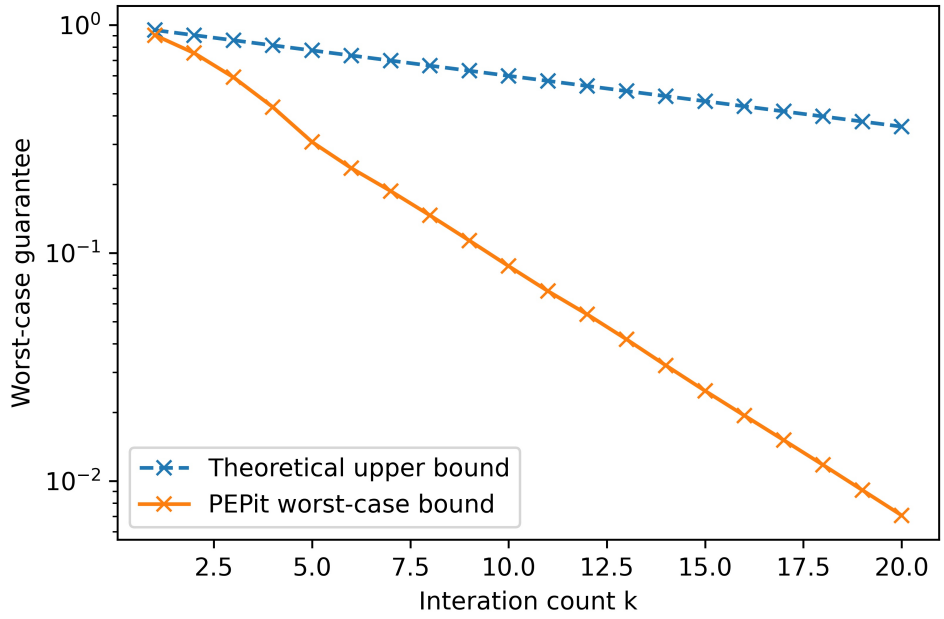


**Figure 5.3:** Heavy-Ball Method : theoretical and PEPit (numerical) worst-case performance bounds as functions of the iteration count

As with the previous example, the worst-case guarantee can still be improved, particularly as the iteration count gets larger.

Another interesting question about the heavy-ball method is to know which parameters $\alpha, \beta$ allow it to converge. The next figure shows the region of $(\alpha, \beta)$ in which Heavy-ball reduces its excess loss after 10 steps (c.f. code A4).

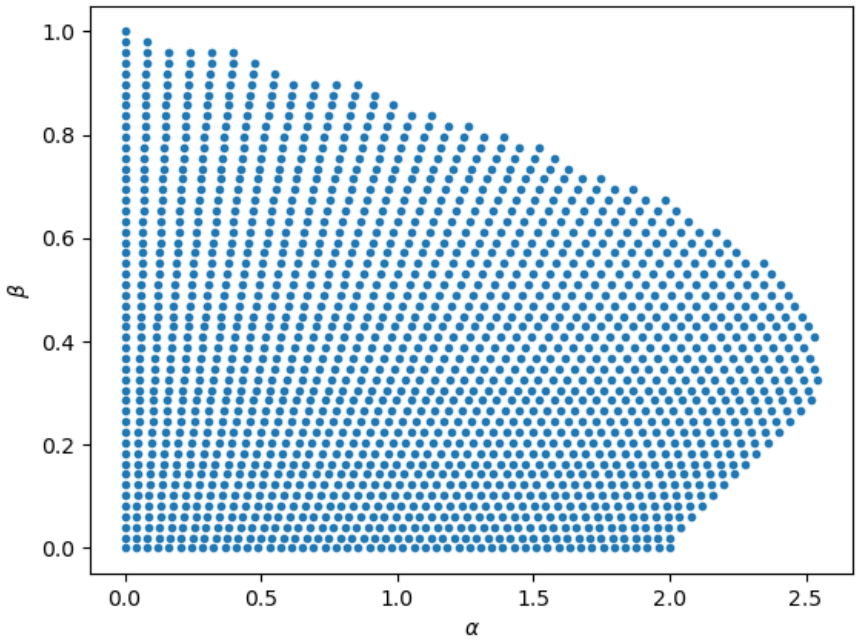


**Figure 5.4:** Heavy-Ball Method : representation of a region of convergence guaranteed by the PEP framework.

# 6 Conclusion

In this thesis, we presented the Performance Estimation framework that automates the search of worst-case guarantees in first order optimization. This framework formulates a worst-case guarantee under the form of an optimization problem. The idea behind it can be decomposed in 4 parts as follows:

- Writing the raw problem as a non convex infinite dimensional optimization problem;
- Using interpolation conditions of the studied class of functions to discretize the worst-case function and end up into a finite dimension problem;
- Using homogeneity arguments to convexify the objective of the PEP;
- Using a semi-lifting technique to convexify the constraints.

In this work, we developed these steps in detail on a simple example to keep the computations light enough for a better comprehension of the different phenomenons that can occur. It is important to understand that this technique can be applied to the study of many algorithms such as Gradient descent or Accelerated method, and on many different classes including - but not limited to - classes of convex functions, smooth functions, smooth and convex or convex and Lipschitz continuous functions.

**Ingredients, limitations and future works.** While this technique can be used in many settings (PEPit package contains about 80 examples illustrating various scenarios [3].), the SDP reformulation needs some ingredients from the studied algorithm and function class to work:

- the class of functions has interpolations conditions that are linear in $G$ and $F$,
- the algorithmic steps can be expressed linearly in terms of the iterates and gradient values (i.e., step-sizes do not depend on the function at hand),
- the performance measure is linear in $G$ and $F$,
- the initial condition is linear in $G$ and $F$.

As soon as those conditions are verified, the PEP framework works and helps finding

[3] https://pepit.readthedocs.io/en/latest/examples.html

optimal guarantee. On the other hand, methods like Polyak step-size Gradient descent do not verify those assumptions and therefore need more effort to be studied into this framework. This is part of the open problems that need to be solved and are left for future work.

# References

[1] Adrien B Taylor, J. M. H. and Glineur, F. (2017). Smooth strongly convex interpolation and exact worst-case performance of first-order methods. *Mathematical Programming, 161(1):307–345.*

[2] Adrien Taylor, Julien Hendrickx, F. G. (2017). Performance estimation toolbox (pesto): automated worst-case analysis of first-order optimization methods. *Proceedings of the 56th Annual Conference on Decision and Control (CDC), pp. 1278-1283.*

[3] Drori, Y. and Teboulle, M. (2014). Performance of first-order methods for smooth convex minimization: a novel approach. *Mathematical Programming, 145(1):451–482.*

[4] d'Aspremont, A., Scieur, D., and Taylor, A. (2021). Acceleration methods. *Foundations and Trends® in Optimization*, 5(1-2):1–245.

[5] Ghadimi, E., Feyzmahdavian, H. R., and Johansson, M. (2014). Global convergence of the heavy-ball method for convex optimization.

[6] Goujaud, B., Moucer, C., Glineur, F., Hendrickx, J., Taylor, A., and Dieuleveut, A. (2022). PEPit: computer-assisted worst-case analyses of first-order optimization methods in Python. *arXiv preprint arXiv:2201.04040.*

[7] Park, C. and Ryu, E. K. (2021). Optimal first-order algorithms as a function of inequalities. *arXiv preprint arXiv:2110.11035.*

[8] Waldspurger, I. (October 15, 2020). Gradient descent. *Course Notes.*

# Appendices

## A1 Gradient Descent in PEPit

Source : https://pepit.readthedocs.io/en/latest/_modules/PEPit/examples/unconstrained_convex_minimization/gradient_descent.html#wc_gradient_descent

```python
from PEPit import PEP
from PEPit.functions import SmoothConvexFunction

def wc_gradient_descent(L, gamma, n, verbose=1):
    # Instantiate PEP
    problem = PEP()

    # Declare a strongly convex smooth function
    func = problem.declare_function(SmoothConvexFunction, mu=0, L=L)

    # Start by defining its unique optimal point xs = x_* and corresponding function
    ↪ value fs = f_*
    xs = func.stationary_point()
    fs = func(xs)

    # Then define the starting point x0 of the algorithm
    x0 = problem.set_initial_point()

    # Set the initial constraint that is the distance between x0 and x^*
    problem.set_initial_condition((x0 - xs) ** 2 <= 1)

    # Run n steps of the GD method
    x = x0
    for _ in range(n):
        x = x - gamma * func.gradient(x)

    # Set the performance metric to the function values accuracy
    problem.set_performance_metric(func(x) - fs)

    # Solve the PEP
    pepit_verbose = max(verbose, 0)
    pepit_tau = problem.solve(verbose=pepit_verbose)

    # Compute theoretical guarantee (for comparison)
    theoretical_tau = L / (2 * (2 * n * L * gamma + 1))

    # Print conclusion if required
    if verbose != -1:
```

```
        print('*** Example file: worst-case performance of gradient descent with fixed
        ↪  step-sizes ***')
        print('\tPEPit guarantee:\t f(x_n)-f_* <= {:.6} ||x_0 -
        ↪  x_*||^2'.format(pepit_tau))
        print('\tTheoretical guarantee:\t f(x_n)-f_* <= {:.6} ||x_0 -
        ↪  x_*||^2'.format(theoretical_tau))

    # Return the worst-case guarantee of the evaluated method (and the reference
    ↪  theoretical value)
    return pepit_tau, theoretical_tau

if __name__ == "__main__":
    L = 3
    pepit_tau, theoretical_tau = wc_gradient_descent(L=L, gamma=1 / L, n=100,
    ↪  verbose=1)
```

Corresponding code for the plotted graph :

```
    L = 1           # smoothness parameter
    mu = 0.1        # strong convexity parameter
    gamma = 1 / L   # step-size

    # Set a list of iteration counter to test
    n_list = list(range(1,21))


    # Compute numerical and theoretical (analytical) worst-case guarantees for each
    ↪  iteration count
    pepit_taus = list()
    theoretical_taus = list()
    for n in n_list:
        pepit_tau, theoretical_tau = wc_gradient_descent(L, gamma, n, verbose=1)
        pepit_taus.append(pepit_tau)
        theoretical_taus.append(theoretical_tau)

    # Plot theoretical and PEPit (numerical) worst-case performance bounds as
    ↪  functions of the iteration count

    plt.plot(n_list, theoretical_taus, '--x', label='Theoretical upper bound')
    plt.plot(n_list, pepit_taus, '-x', label='PEPit worst-case bound')

    plt.semilogy()
    plt.legend()
    plt.xlabel('Interation count k')
    plt.ylabel('Worst-case guarantee')
    plt.show()
```

# A2 Accelerated Gradient Method in PEPit

Source : https://pepit.readthedocs.io/en/latest/_modules/PEPit/examples/unconstrained_convex_minimization/accelerated_gradient_strongly_convex.html#wc_accelerated_gradient_strongly_convex

```python
from math import sqrt
from PEPit import PEP
from PEPit.functions import SmoothStronglyConvexFunction

 wc_accelerated_gradient_strongly_convex(mu, L, n, verbose=1):

    # Instantiate PEP
    problem = PEP()

    # Declare a strongly convex smooth function
    func = problem.declare_function(SmoothStronglyConvexFunction, mu=mu, L=L)

    # Start by defining its unique optimal point xs = x_* and corresponding function
    ↪  value fs = f_*
    xs = func.stationary_point()
    fs = func(xs)

    # Then define the starting point x0 of the algorithm
    x0 = problem.set_initial_point()

    # Set the initial constraint that is a well-chosen distance between x0 and x^*
    problem.set_initial_condition(func(x0) - fs + mu / 2 * (x0 - xs) ** 2 <= 1)

    # Run n steps of the fast gradient method
    kappa = mu / L
    x_new = x0
    y = x0
    for i in range(n):
        x_old = x_new
        x_new = y - 1 / L * func.gradient(y)
        y = x_new + (1 - sqrt(kappa)) / (1 + sqrt(kappa)) * (x_new - x_old)

    # Set the performance metric to the function value accuracy
    problem.set_performance_metric(func(x_new) - fs)

    # Solve the PEP
    pepit_verbose = max(verbose, 0)
    pepit_tau = problem.solve(verbose=pepit_verbose)

    # Compute theoretical guarantee (for comparison)
```

```
    theoretical_tau = (1 - sqrt(kappa)) ** n
    if mu == 0:
        print("Warning: momentum is tuned for strongly convex functions!")

    # Print conclusion if required
    if verbose != -1:
        print('*** Example file: worst-case performance of the accelerated gradient method ***')
        print('\tPEPit guarantee:\t f(x_n)-f_* <= {:.6} (f(x_0) - f(x_*) + mu/2*||x_0 - x_*||**2)'.format(
            pepit_tau))
        print('\tTheoretical guarantee:\t f(x_n)-f_* <= {:.6} (f(x_0) - f(x_*) + mu/2*||x_0 - x_*||**2)'.format(
            theoretical_tau))

    # Return the worst-case guarantee of the evaluated method (and the reference theoretical value)
    return pepit_tau, theoretical_tau

if __name__ == "__main__":
    pepit_tau, theoretical_tau = wc_accelerated_gradient_strongly_convex(mu=0.1, L=1, n=2, verbose=1)
```

Corresponding code for the plotted graph :

```
# Set the parameters
L = 1           # smoothness parameter
mu = 0.1        # strong convexity parameter
gamma = 1 / L   # step-size

# Set a list of iteration counter to test
n_list = list(range(1,21))

# Compute numerical and theoretical (analytical) worst-case guarantees for each iteration count
pepit_taus = list()
theoretical_taus = list()
for n in n_list:
    pepit_tau, theoretical_tau = wc_accelerated_gradient_strongly_convex(mu=mu,L=L,n=n,verbose=verbose)
    pepit_taus.append(pepit_tau)
    theoretical_taus.append(theoretical_tau)

    # Plot theoretical and PEPit (numerical) worst-case performance bounds as functions of the iteration count

    plt.plot(n_list, theoretical_taus, '--x', label='Theoretical upper bound')
```

```
    plt.plot(n_list, pepit_taus, '-x', label='PEPit worst-case bound')

    plt.semilogy()
    plt.legend()
    plt.xlabel('Interation count k')
    plt.ylabel('Worst-case guarantee')
    plt.show()
```

## A3 Heavy-Ball Method in PEPit

Source : https://pepit.readthedocs.io/en/latest/_modules/PEPit/examples/unconstrained_convex_minimization/heavy_ball_momentum.html#wc_heavy_ball_momentum

```
from math import sqrt
from PEPit import PEP
from PEPit.functions import SmoothStronglyConvexFunction

def wc_heavy_ball_momentum(mu, L, alpha, beta, n, verbose=1):
    # Instantiate PEP
    problem = PEP()

    # Declare a smooth strongly convex function
    func = problem.declare_function(SmoothStronglyConvexFunction, mu=mu, L=L)

    # Start by defining its unique optimal point xs = x_* and corresponding function
    ↪  value fs = f_*
    xs = func.stationary_point()
    fs = func(xs)

    # Then define the starting point x0 of the algorithm as well as corresponding
    ↪  function value f0
    x0 = problem.set_initial_point()
    f0 = func(x0)

    # Set the initial constraint that is the distance between f(x0) and f(x^*)
    problem.set_initial_condition((f0 - fs) <= 1)

    # Run one step of the heavy ball method
    x_new = x0
    x_old = x0

    for _ in range(n):
        x_next = x_new - alpha * func.gradient(x_new) + beta * (x_new - x_old)
        x_old = x_new
```

```
        x_new = x_next

    # Set the performance metric to the final distance to optimum
    problem.set_performance_metric(func(x_new) - fs)

    # Solve the PEP
    pepit_verbose = max(verbose, 0)
    pepit_tau = problem.solve(verbose=pepit_verbose)

    # Compute theoretical guarantee (for comparison)
    theoretical_tau = (1 - alpha * mu) ** n

    # Print conclusion if required
    if verbose != -1:
        print('*** Example file: worst-case performance of the Heavy-Ball method ***')
        print('\tPEPit guarantee:\t f(x_n)-f_* <= {:.6} (f(x_0) -
        ↪  f(x_*))'.format(pepit_tau))
        print('\tTheoretical guarantee:\t f(x_n)-f_* <= {:.6} (f(x_0) -
        ↪  f(x_*))'.format(theoretical_tau))

    # Return the worst-case guarantee of the evaluated method (and the reference
    ↪  theoretical value)
    return pepit_tau, theoretical_tau

if __name__ == "__main__":
    mu = 0.1
    L = 1.
    alpha = 1 / (2 * L)  # alpha \in [0, 1 / L]
    beta = sqrt((1 - alpha * mu) * (1 - L * alpha))
    pepit_tau, theoretical_tau = wc_heavy_ball_momentum(mu=mu, L=L, alpha=alpha,
    ↪  beta=beta, n=2, verbose=1)
```

Corresponding code for the plotted graph :

```
    verbose = -1
    L = 1           # smoothness parameter
    mu = 0.1        # strong convexity parameter
    gamma = 1 / L   # step-size

    mu = 0.1
    L = 1
    alpha = 1 / (2 * L)  # alpha \in [0, 1 / L]
    beta = np.sqrt((1 - alpha * mu) * (1 - L * alpha))

    # Set a list of iteration counter to test
    n_list = list(range(1,21))

```

```
14
15      # Compute numerical and theoretical (analytical) worst-case guarantees for each
        ↪ iteration count
16      pepit_taus = list()
17      theoretical_taus = list()
18      for n in n_list:
19          pepit_tau, theoretical_tau = wc_heavy_ball_momentum(mu, L, alpha, beta, n,
            ↪ verbose=1)
20          pepit_taus.append(pepit_tau)
21          theoretical_taus.append(theoretical_tau)
22
23      # Plot theoretical and PEPit (numerical) worst-case performance bounds as
        ↪ functions of the iteration count
24
25      plt.plot(n_list, theoretical_taus, '--x', label='Theoretical upper bound')
26      plt.plot(n_list, pepit_taus, '-x', label='PEPit worst-case bound')
27
28      plt.semilogy()
29      plt.legend()
30      plt.xlabel('Interation count k')
31      plt.ylabel('Worst-case guarantee')
32      plt.show()
```

# A4 Heavy-Ball Method : representation of a region of convergence guaranteed by the PEP framework.

```
1  from tqdm import tqdm
2
3  from math import sqrt
4  import numpy as np
5
6  import matplotlib.pyplot as plt
7
8  from PEPit import PEP
9  from PEPit.functions import SmoothStronglyConvexFunction
10
11 def wc_heavy_ball_momentum(mu, L, alpha, beta, n, verbose=1):
12
13     # Instantiate PEP
14     problem = PEP()
15
16     # Declare a smooth strongly convex function
17     func = problem.declare_function(SmoothStronglyConvexFunction, mu=mu, L=L)
18
```

```
    # Start by defining its unique optimal point xs = x_* and corresponding function value fs = f_*
    xs = func.stationary_point()
    fs = func(xs)

    # Then define the starting point x0 of the algorithm as well as corresponding function value f0
    x0 = problem.set_initial_point()
    f0 = func(x0)

    # Set the initial constraint that is the distance between f(x0) and f(x^*)
    problem.set_initial_condition((f0 - fs) <= 1)

    # Run one step of the heavy ball method
    x_new = x0
    x_old = x0

    for _ in range(n):
        x_next = x_new - alpha * func.gradient(x_new) + beta * (x_new - x_old)
        x_old = x_new
        x_new = x_next

    # Set the performance metric to the final distance to optimum
    problem.set_performance_metric(func(x_new) - fs)

    # Solve the PEP
    pepit_verbose = max(verbose, 0)
    pepit_tau = problem.solve(verbose=pepit_verbose)

    # Compute theoretical guarantee (for comparison)
    theoretical_tau = (1 - alpha * mu) ** n

    # Print conclusion if required
    if verbose != -1:
        print('*** Example file: worst-case performance of the Heavy-Ball method ***')
        print('\tPEPit guarantee:\t f(x_n)-f_* <= {:.6} (f(x_0) - f(x_*))'.format(pepit_tau))
        print('\tTheoretical guarantee:\t f(x_n)-f_* <= {:.6} (f(x_0) - f(x_*))'.format(theoretical_tau))

    # Return the worst-case guarantee of the evaluated method (and the reference theoretical value)
    return pepit_tau, theoretical_tau

if __name__ == "__main__":
    mu = 0.1
    L = 1.

```

```
    beta_list = []
    alpha_list = []
    rate_list = []
    for beta in tqdm(np.linspace(0, 1, 50)):
        for alpha in np.linspace(0, 2 * (1+beta) / L, 50):
            rate, _ = wc_heavy_ball_momentum(mu=mu, L=L, alpha=alpha, beta=beta, n=10,
            ↪  verbose=-1)

            beta_list.append(beta)
            alpha_list.append(alpha)
            rate_list.append(rate)

    betas = np.array(beta_list)
    alphas = np.array(alpha_list)
    rates = np.array(rate_list)

    plt.plot(alphas[rates < 1], betas[rates < 1], '.')
    plt.xlabel(r"$\alpha$")
    plt.ylabel(r"$\beta$")
    plt.savefig("hb.png")
```

# Statement of Academic Integrity Regarding Plagiarism

I, the undersigned TODOSKOVA Darya [family name, given name(s)], hereby certify on my honor that:

1. The results presented in this report are the product of my own work.

2. I am the original creator of this report.

3. I have not used sources or results from third parties without clearly stating thus and referencing them according to the recommended rules for providing bibliographic information.

**Declaration to be copied below:**

*I hereby declare that this work contains no plagiarized material.*

Date 19/03/2023

Signature