Mathematical Programming Computation (2024) 16:337–367
https://doi.org/10.1007/s12532-024-00259-7

FULL LENGTH PAPER

PEPIT: computer-assisted worst-case analyses of first-order optimization methods in Python

Baptiste Goujaud[1] · Céline Moucer[2,3] · François Glineur[4,5] · Julien M. Hendrickx[5] · Adrien B. Taylor[2] · Aymeric Dieuleveut[1]

Received: 16 December 2022 / Accepted: 26 May 2024 / Published online: 9 August 2024
© Springer-Verlag GmbH Germany, part of Springer Nature and Mathematical Optimization Society 2024

Abstract

PEPIT is a PYTHON package aiming at simplifying the access to worst-case analyses of a large family of first-order optimization methods possibly involving gradient, projection, proximal, or linear optimization oracles, along with their approximate, or Bregman variants. In short, PEPIT is a package enabling computer-assisted worst-case analyses of first-order optimization methods. The key underlying idea is to cast the problem of performing a worst-case analysis, often referred to as a performance estimation problem (PEP), as a semidefinite program (SDP) which can be solved numerically. To do that, the package users are only required to write first-order methods nearly as they would have implemented them. The package then takes care of the SDP modeling parts, and the worst-case analysis is performed numerically via standard solvers.

✉ Baptiste Goujaud
baptiste.goujaud@gmail.com

✉ Adrien B. Taylor
adrien.taylor@inria.fr

Céline Moucer
celine.moucer@inria.fr

François Glineur
francois.glineur@uclouvain.be

Julien M. Hendrickx
julien.hendrickx@uclouvain.be

Aymeric Dieuleveut
aymeric.dieuleveut@polytechnique.edu

1 CMAP, École Polytechnique, Institut Polytechnique de Paris, Paris, France

2 INRIA & D.I. École Normale Supérieure, CNRS & PSL Research University, Paris, France

3 Ecole des Ponts, Marne-la-Vallée, France

4 CORE, UCLouvain, Louvain-la-Neuve, Belgium

5 ICTEAM Institute, UCLouvain, Louvain-la-Neuve, Belgium

Keywords Optimization · Worst-case analyses · Convergence analyses · Performance estimation problems · First-order methods · Splitting methods · Semidefinite programming

Mathematics Subject Classification 65K10

1 Introduction

Due to their low cost per iteration, first-order optimization methods became a major tool in the modern numerical optimization toolkit. Those methods are particularly well suited when targeting only low to medium accuracy solutions, and play a central role in many fields of applications that include machine learning and signal processing. Their simplicity further allows both occasional and expert users to use them. On the contrary, when it comes to their analyses (usually based on worst-case scenarios), they are mostly reserved for expert users. The main goal of this work is to allow simpler and reproducible access to worst-case analyses for first-order methods.

PEPit is a PYTHON package enabling computer-assisted worst-case analysis of a large family of first-order optimization methods. After being provided with a first-order method and a standard problem class, the package reformulates the problem of performing a worst-case analysis as a semidefinite program (SDP). This technique is commonly referred to as *performance estimation problems* (PEPs) and was introduced by [6, 7]. The package uses PEPs as formalized by [34, 36].

In short, performing a worst-case analysis of a first-order algorithm usually relies on four main ingredients: a first-order algorithm (to be analyzed), a class of problems (containing the assumptions on the function to be minimized), a performance measure (measuring the quality of the output of the algorithm under consideration; for convenience here we assume that the algorithm aims at minimizing this performance measure and our analysis aims at finding a worst-case guarantee on it), and an initial condition (measuring the quality of the initial iterate). Performing the worst-case analysis (i.e., computing worst-case scenarios) corresponds to maximizing the performance measure of the algorithm on the class of problems, under a constraint induced by the initial condition. It turns out that such optimization problems can often be solved using SDPs in the context of first-order methods.

PEPs provide a principled approach to worst-case analyses but usually rely on potentially tedious semidefinite programming (SDP) modeling and coding steps. The PEPit package eases access to the methodology by automatically handling the modeling part, thereby limiting the amount of time spent on this tedious task and the risk of introducing coding mistakes in the process. In short, this work allows users to (i) write their first-order algorithms nearly as they would have implemented them, (ii) let PEPit (a) perform the modeling and coding steps, and (b) perform the worst-case analysis numerically using tools for semidefinite programming in PYTHON [5, 21, 26].

As a result, the package enables users to easily obtain worst-case analyses for most of the standard first-order methods, classes of problems, performance measures, and initial conditions. This is useful to numerically verify existing convergence guarantees, as well as to ease the development of new analyses and methods. To this end,

the toolbox contains tools for analyzing classical scenarios of the first-order literature: standard problem classes (such as convex functions, smooth convex functions, Lipschitz convex functions, etc.) and algorithmic operations (such as gradient, proximal, or linear optimization oracles, etc.). Finally, the package contains more than 75 examples and is designed in an open fashion, allowing users to easily add new ingredients (such as their own problem classes, oracles, or algorithms as examples).

1.1 Organization of the paper

This paper is organized as follows. First, Sect. 2 exemplifies the PEP approach on a very simple example, namely computing a worst-case contraction factor for gradient descent, and shows how to code this example in PEPit. Section 3 provides details on the semidefinite programs that can be formulated through the package along with the relationship between those formulations and the coding steps. Then, Sect. 4 provides a roadmap through the package. Finally, Sect. 5 provides three additional numerical examples (including a composite minimization problem and a stochastic one), and some concluding remarks and perspectives are drawn in Sect. 6.

1.2 Related works

The PEPit package relies on performance estimation problems as formalized in [34]. It also contains some improvements and generalizations to other problem and algorithmic classes such as monotone and nonexpansive operators [20, 29], quadratic optimization [2], stochastic methods and verification of potential (or Lyapunov/energy) functions [10, 15, 31] as inspired by the related control-theoretic IQC framework [19]. The package also contains numerous examples; e.g., recent analyses and developments from [1, 11, 12, 14, 16–18, 20, 32, 39]. The package can be seen as an extended open source PYTHON version of the MATLAB package PESTO [35] on various aspects such as its documentation, its coding style and its access through standard open interfaces (such as pip), PEPit is more professional than PESTO.

1.3 Dependencies

The package heavily builds on existing software for solving semidefinite programs, including CVXPY [5], SCS [26], and MOSEK [21].

2 PEPit on a simple example

In this section, we illustrate the use of the package for studying the worst-case properties of a standard scenario: gradient descent for minimizing a smooth strongly convex function. The goal of this elementary example is twofold. First, we want to provide the base mathematical steps enabling the use of semidefinite programming for performing worst-case analyses, together with a corresponding PEPit code. Second, we want to highlight the main ingredients that can be generalized to other problem setups

(e.g., Theorem 1 below providing "interpolation conditions" for the class of smooth strongly convex functions), allowing us to analyze more algorithms under different assumptions (which are listed in Sect. 4).

For this example, we consider the convex optimization problem

$$\min_{x\in\mathbb{R}^d} f(x), \tag{1}$$

where f is L-smooth and μ-strongly convex (notation $f \in \mathcal{F}_{\mu,L}(\mathbb{R}^d)$, or $f \in \mathcal{F}_{\mu,L}$ when d is unspecified). So we assume f to satisfy

- (L-smoothness) $\forall x, y \in \mathbb{R}^d$ we have that

$$f(x) \leqslant f(y) + \langle \nabla f(y); x - y\rangle + \frac{L}{2}\|x - y\|_2^2,$$

- (μ-strong convexity) $\forall x, y \in \mathbb{R}^d$ we have that

$$f(x) \geqslant f(y) + \langle \nabla f(y); x - y\rangle + \frac{\mu}{2}\|x - y\|_2^2.$$

Our goal for the rest of this section is to show how to compute the smallest possible $\tau(\mu, L, \gamma)$ (often referred to as the "contraction factor") such that

$$\|x_1 - y_1\|_2^2 \leqslant \tau(\mu, L, \gamma)\|x_0 - y_0\|_2^2, \tag{2}$$

is valid for all $f \in \mathcal{F}_{\mu,L}$ and all $x_0, y_0 \in \mathbb{R}^d$ when x_1 and y_1 are obtained from gradient steps from respectively x_0 and y_0. That is, $x_1 = x_0 - \gamma\nabla f(x_0)$ and $y_1 = y_0 - \gamma\nabla f(y_0)$. First, we show that the problem of computing $\tau(\mu, L, \gamma)$ can be framed as a semidefinite program (SDP), and then illustrate how to use PEPit for computing it without going into the SDP modeling details.

2.1 A performance estimation problem for the gradient method

It is relatively straightforward to establish that the smallest possible $\tau(\mu, L, \gamma)$ for which (2) is valid can be computed as the worst-case value of $\|x_1 - y_1\|_2^2$ when $\|x_0 - y_0\|_2^2 \leqslant 1$. That is, we compute $\tau(\mu, L, \gamma)$ as the optimal value to the following optimization problem:

$$\begin{aligned}\tau(\mu, L, \gamma) = \max_{\substack{f,d\\ x_0,x_1\in\mathbb{R}^d\\ y_0,y_1\in\mathbb{R}^d}} \ & \|x_1 - y_1\|_2^2 \\ \text{s.t. } & d \in \mathbb{N},\ f \in \mathcal{F}_{\mu,L}(\mathbb{R}^d), \\ & \|x_0 - y_0\|_2^2 \leqslant 1, \\ & x_1 = x_0 - \gamma\nabla f(x_0), \\ & y_1 = y_0 - \gamma\nabla f(y_0).\end{aligned} \tag{3}$$

As written in (3), this problem involves an infinite-dimensional variable f. Our first step towards formulating (3) as an SDP consists of reformulating it by sampling f (i.e., evaluating its function value and gradient) at the two points where its gradient is evaluated:

$$\begin{aligned}\tau(\mu, L, \gamma) = \max_{\substack{d, f_{x_0}, f_{y_0} \\ x_0, x_1, g_{x_0} \in \mathbb{R}^d \\ y_0, y_1, g_{y_0} \in \mathbb{R}^d}} & \|x_1 - y_1\|_2^2 \\ \text{s.t. } & d \in \mathbb{N}, \\ & \|x_0 - y_0\|_2^2 \leqslant 1, \\ & \exists f \in \mathcal{F}_{\mu,L}(\mathbb{R}^d) : \begin{cases} f(x_0) = f_{x_0} \ \nabla f(x_0) = g_{x_0} \\ f(y_0) = f_{y_0} \ \nabla f(y_0) = g_{y_0} \end{cases} \\ & x_1 = x_0 - \gamma g_{x_0}, \\ & y_1 = y_0 - \gamma g_{y_0},\end{aligned} \tag{4}$$

where we replaced the variable f by its discrete version, which we constrain to be "interpolable" (or "extendable") by a smooth strongly convex function over $\mathbb{R}^d$. To arrive at a tractable problem, we use the following interpolation (or extension) result.

Theorem 1 [36, Theorem 4] *Let I be an index set and $S = \{(x_i, g_i, f_i)\}_{i \in I}$ be such that $x_i, g_i \in \mathbb{R}^d$ and $f_i \in \mathbb{R}$ for all $i \in I$. There exists a function $F \in \mathcal{F}_{\mu,L}(\mathbb{R}^d)$ such that $f_i = F(x_i)$ and $g_i = \nabla F(x_i)$ (for all $i \in I$) if and only if for all $i, j \in I$ we have*

$$f_i \geq f_j + \langle g_j; x_i - x_j \rangle + \tfrac{1}{2L} \left\| g_j - g_i \right\|_2^2 + \tfrac{\mu L}{2(L-\mu)} \left\| x_i - x_j - \tfrac{1}{L}(g_i - g_j) \right\|_2^2. \tag{5}$$

Using Theorem 1, we can formulate the problem of computing $\tau(\mu, L, \gamma)$ as a (nonconvex) quadratic problem:

$$\begin{aligned}\max_{\substack{d, f_{x_0}, f_{y_0} \\ x_0, g_{x_0} \in \mathbb{R}^d \\ y_0, g_{y_0} \in \mathbb{R}^d}} & \left\| (x_0 - \gamma g_{x_0}) - (y_0 - \gamma g_{y_0}) \right\|_2^2 \\ \text{s.t. } & d \in \mathbb{N}, \\ & \|x_0 - y_0\|_2^2 \leqslant 1, \\ & f_{y_0} \geq f_{x_0} + \langle g_{x_0}; y_0 - x_0 \rangle + \tfrac{1}{2L} \|g_{y_0} - g_{x_0}\|_2^2 \\ & \qquad + \tfrac{\mu L}{2(L-\mu)} \left\| x_0 - y_0 - \tfrac{1}{L}(g_{x_0} - g_{y_0}) \right\|_2^2, \\ & f_{x_0} \geq f_{y_0} + \langle g_{y_0}; x_0 - y_0 \rangle + \tfrac{1}{2L} \|g_{y_0} - g_{x_0}\|_2^2 \\ & \qquad + \tfrac{\mu L}{2(L-\mu)} \|x_0 - y_0 - \tfrac{1}{L}(g_{x_0} - g_{y_0})\|_2^2.\end{aligned} \tag{6}$$

Relying on a standard trick from semidefinite programming, one can convexify this problem using a Gram representation of the variable (this is due to maximization over

d). That is, we formulate the problem using a positive semidefinite matrix $G \succcurlyeq 0$ defined as

$$G \triangleq \begin{pmatrix} \|x_0 - y_0\|_2^2 & \langle x_0 - y_0; g_{x_0}\rangle & \langle x_0 - y_0; g_{y_0}\rangle \\ \langle x_0 - y_0; g_{x_0}\rangle & \|g_{x_0}\|_2^2 & \langle g_{x_0}; g_{y_0}\rangle \\ \langle x_0 - y_0; g_{y_0}\rangle & \langle g_{x_0}; g_{y_0}\rangle & \|g_{y_0}\|_2^2 \end{pmatrix} \succcurlyeq 0. \tag{7}$$

Using this change of variable, we arrive at

$$\begin{aligned} \max_{f_{x_0}, f_{y_0}, G} \; & G_{1,1} - 2\gamma(G_{1,2} - G_{1,3}) + \gamma^2(G_{2,2} + G_{3,3} - 2G_{2,3}) \\ \text{s.t. } & G \succcurlyeq 0, \\ & G_{1,1} \leqslant 1, \\ & f_{y_0} \geq f_{x_0} + \tfrac{1}{L-\mu}\left(\tfrac{\mu L}{2}G_{1,1} - LG_{1,2} + \mu G_{1,3} + \tfrac{1}{2}G_{2,2} - G_{2,3} + \tfrac{1}{2}G_{3,3}\right), \\ & f_{x_0} \geq f_{y_0} + \tfrac{1}{L-\mu}\left(\tfrac{\mu L}{2}G_{1,1} - \mu G_{1,2} + LG_{1,3} + \tfrac{1}{2}G_{2,2} - G_{2,3} + \tfrac{1}{2}G_{3,3}\right), \end{aligned} \tag{8}$$

which can be solved numerically using standard tools, see, e.g., [5, 21, 26]. Using numerical and/or symbolical computations, one can then easily arrive at $\tau(\mu, L, \gamma) = \max\{(1 - L\gamma)^2, (1 - \mu\gamma)^2\}$ and hence that

$$\|x_1 - y_1\|_2^2 \leqslant \max\{(1 - L\gamma)^2, (1 - \mu\gamma)^2\}\|x_0 - y_0\|_2^2, \tag{9}$$

for all $d \in \mathbb{N}$, $f \in \mathcal{F}_{\mu,L}(\mathbb{R}^d)$ and $x_0, y_0 \in \mathbb{R}^d$ when $x_1, y_1 \in \mathbb{R}^d$ are generated from gradient steps from respectively x_0 and y_0. In the next section, we show how to perform this analysis using PEPit, which automates the sampling (i.e., the evaluations of the function or its gradients on given points) and SDP-modeling procedures. In more complex settings where more functions need to be sampled and/or more iterates have to be taken into account, avoiding those steps allows to largely limits the probability of making a mistake in the process of performing the worst-case analysis (numerically) while allowing to spare a significant amount of time in the process.

Remark 1 (Important ingredients for the SDP reformulations) To understand what PEPit can do, it is crucial to understand which elements allowed to cast the worst-case analysis as such a semidefinite program (which is what we refer to as the "modeling" of the problem). In short, the SDP reformulation of the worst-case computation problem was made possible due to 4 main ingredients (see, e.g., [34, Sect. 2.2]):

1. 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),
2. The class of functions has "interpolation condition"[1] that are linear in G and the function values,

[1] Interpolation conditions characterize the **existence** of a function (that has particular function values and gradients at given points) in the considered class by a list of constraints on those gradients, points, and function values. Such interpolation theorems (see, e.g., Theorem 1) have been obtained in the literature for various problem classes, see, e.g., [29, 34, 36].

3. The performance measure is linear (or convex piecewise linear) in G and the function values,
4. The initial condition is linear in G and the function values.

Those ingredients allow the use of PEPs much beyond the simple setup of this section. That is, PEPs apply for performing worst-case analyses involving a variety of first-order oracles, initial conditions, performance measures, and problem classes (see Sect. 3 for the general modeling of the problem, and Sect. 4 for a non-exhaustive list of cases that are covered).

2.2 Code

In the previous section, we introduced the PEP and SDP modeling steps for computing a tight worst-case contraction factor for gradient descent in the form (2). Although this particular SDP (8) might be solved analytically, many optimization methods lead to larger SDPs with more complicated structures. In general, we can reasonably only hope to solve them numerically. In the following lines, we describe how to use PEPIT for computing a *contraction factor* without explicitly going into the modeling steps. Compared to previous section, we allow ourselves to perform $n \in \mathbb{N}$ iterations and compute the smallest possible value of $\tau(\mu, L, \gamma, n)$ such that

$$\|x_n - y_n\|_2^2 \leqslant \tau(\mu, L, \gamma, n)\|x_0 - y_0\|_2^2, \tag{10}$$

where x_n and y_n are computed from n iterations of gradient descent with step-size γ starting from respectively x_0 and y_0. As illustrated in the previous section for the case $n = 1$, computing the smallest possible such $\tau(\mu, L, \gamma, n)$ is equivalent to computing the worst-case value of $\|x_n - y_n\|_2^2$ under the constraint that $\|x_0 - y_0\|_2^2 \leqslant 1$ (note that we naturally have that $\tau(\mu, L, \gamma, n) \leqslant (\tau(\mu, L, \gamma, 1))^n$). This is what we do in the following lines using PEPIT.

Imports. Before going into the example, we have to include the right PYTHON imports. For this example, it is necessary to perform two imports.

```
from PEPit import PEP
from PEPit.functions import \
    SmoothStronglyConvexFunction
```

Initialization of PEPIT. First, we set the stage by initializing a PEP object. This object allows manipulating the forthcoming ingredients of the PEP, such as functions and iterates.

```
problem = PEP()
```

For the sake of the example, let us pick some simple values for the problem class and algorithmic parameters, for which we perform the worst-case analysis.

```
L = 1.              # Smoothness parameter
mu = .1             # Strong convexity parameter
gamma = 1. / L      # Step size
n = 1               # Number of iterations
```

Specifying the problem class. Second, we specify our working assumptions on the function to be optimized and instantiate a corresponding object. Here, the minimization problem at hand was of the form (1) with a smooth strongly convex function.

```
# Declare an L-smooth mu-strongly convex function
# named "func"
func = problem.declare_function(
    SmoothStronglyConvexFunction,
    mu=mu,   # Strong convexity param.
    L=L)     # Smoothness param.
```

Algorithm initialization. Third, we can instantiate the starting points for the two gradient methods that we will run, and specify an *initial condition* on those points. To this end, two starting points x_0 and y_0 are introduced, one for each trajectory, and a bound on the initial distance between those points is specified as $\|x_0 - y_0\|^2 \leqslant 1$.

```
# Declare two starting points
x_0 = problem.set_initial_point()
y_0 = problem.set_initial_point()

# Initial condition ||x_0 - y_0||^2 <= 1
problem.set_initial_condition((x_0 - y_0) ** 2 <= 1)
```

Algorithm implementation. In this fourth step, we specify the algorithm in a natural format. In this example, we simply use the iterates (which are PEPit objects) as if we had to implement gradient descent in practice using a simple loop.

```
# Initialize the algorithm
x = x_0
y = y_0
# Run n steps of the GD method for the two sequences
for _ in range(n):
    # Replace x and y with their next iterate
    x = x - gamma * func.gradient(x)   # call to f'(x)
    y = y - gamma * func.gradient(y)   # call to f'(y)
```

Setting up a performance measure. It is crucial for the worst-case analysis to specify the *metric* for which we wish to compute a worst-case performance. In this example, we wish to compute the worst-case value of $\|x_n - y_n\|^2$, which we specify as follows.

```
# Set the performance metric to the distance
# ||x_n - y_n||^2
problem.set_performance_metric((x-y)**2)
```

Solving the PEP. The last natural stage in the process is to solve the corresponding PEP. This is done via the following line, which will ask PEPit to perform the modeling steps and to call an appropriate SDP solver (which should be installed beforehand) to perform the worst-case analysis.

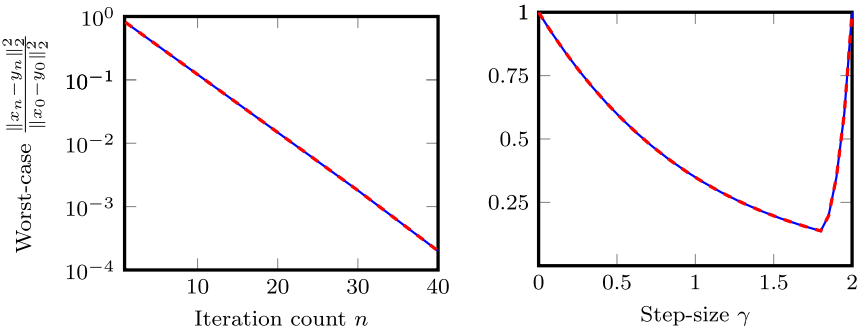
```
# Solve the PEP
pepit_tau = problem.solve()
```

Output. Running these pieces of code (see PEPit/examples/ for the complete example) for some specific values of the parameters $n = 1$, $L = 1$, $\mu = .1$ and $\gamma = 1$, one obtains the following output.

```
(PEPit) Setting up the problem: size of the Gram matrix: 4x4
(PEPit) Setting up the problem: performance measure is the minimum of
                                1 element
(PEPit) Setting up the problem: Adding initial conditions
                                and general constraints ...
(PEPit) Setting up the problem: initial conditions
                                and general constraints
                                (1 constraint(s) added)
(PEPit) Setting up the problem: interpolation conditions for
                                1 function(s)
            Function 1 : Adding 2 scalar constraint(s) ...
            Function 1 : 2 scalar constraint(s) added
(PEPit) Setting up the problem: additional constraints for
                                0 function(s)
(PEPit) Compiling SDP
(PEPit) Calling SDP solver
(PEPit) Solver status: optimal (wrapper:cvxpy, solver: MOSEK);
                       optimal value: 0.8100000029203449
(PEPit) Primal feasibility check:
        The solver found a Gram matrix that is positive semi-definite
        up to an error of 1.896548260018477e-09
        All the primal scalar constraints are verified
        up to an error of 3.042855638898251e-09
(PEPit) Dual feasibility check:
        The solver found a residual matrix that is positive
        semi-definite
        All the dual scalar values associated with inequality
        constraints are nonnegative
(PEPit) The worst-case guarantee proof is perfectly reconstituted
        up to an error of 4.0078754315331366e-08
(PEPit) Final upper bound (dual): 0.8100000036427537
        and lower bound (primal example): 0.8100000029203449
(PEPit) Duality gap: absolute: 7.224087994472939e-10
                     and relative: 8.918627121515396e-10
```

Note that the size of the SDP is larger than that of Sect. 2 (4×4 instead of 3×3 in (7)) because the modeling step is done in a slightly more generic way, which might not be exploiting all specificities of the problem at hand (see formulation in (7) where we use the variable $x_0 - y_0$ instead of both x_0 and y_0 as they only appear together in the original problem formulation). For more complete examples of worst-case analyses using PEPit, see Sect. 5.

It is also possible to run the code for different values of the parameters, as exemplified in Fig. 1. This simple example allows us to observe that numerical values obtained from PEPit match the worst-case guarantee (1a), and to optimize the step-size numerically in Fig. 1b.



(a) Worst-case guarantee and theoretical tight bound as a function of the iteration count n for $\gamma = \frac{1}{L}$.

(b) Worst-case guarantee and theoretical tight bound as a function of the step-size γ, at iteration $n = 5$.

Fig. 1 Comparison: worst-case guarantee from PEPIT (*plain blue*) and theoretical tight worst-case bound (9) for gradient descent in terms of $\frac{\|x_n-y_n\|_2^2}{\|x_0-y_0\|_2^2}$ (*dashed red*). Problem parameters fixed to $\mu = 0.1$ and $L = 1$

3 PEPit code structure and semidefinite formulation

This section provides the general semidefinite program (SDP) that is formulated by the package, as well as its relationship with the code. As already underlined, PEPIT aims at providing simple ways to model performance estimation problems (PEPs) by abstracting the coding of those SDPs. Then, PEPIT passes the SDP either (i) to CVXPY [5], thereby benefiting from all SDP solvers that are interfaced with it (such as [21, 26]), or (ii) directly to MOSEK [21].

Before going into the implementation details of PEPit, note that executing the codes from Sect. 3.2 to 3.5 requires the following imports.

```
# For Section 3.2 to Section 3.5
import PEPit
import PEPit.functions

# For Section 3.3 and Section 3.5 we also need:
from PEPit.functions import\
    ConvexFunction,\
    ConvexIndicatorFunction,\
    SmoothConvexFunction

# For the examples of Section 3.3:
from PEPit.primitive_steps import\
    proximal_step,\
    linear_optimization_step,\
    inexact_gradient_step
```

3.1 Semidefinite formulation

The package formulates SDPs of the form

$$\begin{aligned} \max_{\tau\in\mathbb{R}, G\in\mathbb{S}^{n_p}, H\in\mathbb{R}^{n_h}} \quad & \tau \\ \text{s.t. } & G \succcurlyeq 0 \\ & \tau - \text{Tr}\left(A_i G\right) - a_i^T H - \alpha_i \leqslant 0 \quad \forall i \in I_1 \\ & \left[\text{Tr}\left(B_{j,k,i} G\right) + b_{j,k,i}^T H + \beta_{j,k,i}\right]_{1\leq j,k\leq n_i} \succcurlyeq 0 \quad \forall i \in I_2, \end{aligned} \tag{11}$$

with $n_h,\ n_p \in \mathbb{N}$, some index sets I_1 and I_2 in $\mathbb{N}$, and sets of problem parameters $\{(A_i, a_i, \alpha_i)\}_{i\in I_1}$, and $\{(B_{j,k,i}, b_{j,k,i}, \beta_{j,k,i})\}_{1\leq j,k\leq n_i, i\in I_2}$ ($n_i \in \mathbb{N}_{>0}$ for all $i \in I_2$) of appropriate dimensions, which are constructed from the algorithm and the class of problems at hand (and hence all depend on the parameters of the algorithm and of those of the class of problems) for computing appropriate worst-case scenarios.

Formulating such an SDP is usually cumbersome and relatively error-prone, and the role of PEPIT is to generate all those parameters in a user-friendly way. That is, PEPIT parses the natural description of the problem the user is familiar with. The problem is not described in terms of the SDP variables (G, H) but rather in terms of another couple (P, H) with the following structure:

$$P \triangleq [p_1,\ p_2,\ \ldots,\ p_{n_p}], \qquad H \triangleq [h_1,\ h_2\ \ldots,\ h_{n_h}], \tag{12}$$

for some $\{p_i\}_{1\leq i\leq n_p} \subset \mathbb{R}^d$ for some $d \in \mathbb{N}$, and $\{h_i\}_{1\leq i\leq n_h} \subset \mathbb{R}$. Hence $P \in \mathbb{R}^{d\times n_p}$ and $H \in \mathbb{R}^{n_h}$. The relationship with (11) is that G can be constructed without loss of generality as $G \triangleq P^T P \succeq 0$.

3.2 Base PEPIT objects

Using previous notations, PEPIT allows the user to formulate the problem in terms of (P, H) instead of (G, H), which turns out much more natural for describing many algorithms and problem classes. The base working procedure is as follows:

- Each p_i corresponds to a base PEPIT.POINT object (referred to as a *leaf* element). Such objects can be added and subtracted together, and scaled by real values for forming new objects PEPIT.POINT (which are then combinations of *leaf* elements). PEPIT.POINT objects can be understood as elements of $\mathbb{R}^d$, the space of iterates/gradients.
- Each h_i corresponds to a base PEPIT.EXPRESSION object (referred to as a *leaf* element). Such objects can be added and subtracted together, but also scaled by real values for forming new objects PEPIT.EXPRESSION (which are combinations of *leaf* elements). PEPIT.EXPRESSION objects can be understood as elements of $\mathbb{R}$, such as, for instance, (scalar) function values.
- It is possible to compute dot products of PEPIT.POINT objects (e.g., $p_i^T p_j$ for some $1 \leq i,\ j \leq n_p$), resulting in PEPIT.EXPRESSION objects.

- Comparing two PEPit.Expression objects with an operator in $\{=, \leq, \geq\}$ leads to a PEPit.Constraint object. Similarly, PEPit.Expression can be gathered in arrays to form PEPit.PSD_matrix objects for formulating semidefinite constraints.

Example 1 Following up on the notations from Sect. 2 for describing one iteration of gradient descent, a possibility is to think of p_1 as corresponding to some x_0, of p_2 as a corresponding gradient g_{x_0}, and of h_1 as the corresponding function value f_{x_0}. For a unit step-size, one can form $x_1 = p_1 - p_2$ as follows.

```
x_0 = PEPit.Point()  # a leaf PEPit.Point (p_1)
g_x_0 = PEPit.Point()  # a leaf PEPit.Point  (p_2)
f_x_0 = PEPit.Expression() # a leaf PEPit.Expression (h_1)

x_1 = x_0 - g_x_0  # x_1 is a PEPit.Point
g_x_1 = PEPit.Point()  # a leaf PEPit.Point  (p_3)
f_x_1 = PEPit.Expression() # a leaf PEPit.Expression (h_2)
```

It is important to note that each call to PEPit.Point() increments n_p (dimension G in 11). Similarly, PEPit.Expression() increments n_h. It is cheaper to solve a problem with as few leaf points and leaf expressions as possible.

To formulate the objective, initial conditions, and interpolation constraints, we use PEPit.Expression objects, which we compare together to form PEPit.Constraint objects. For instance, interpolation conditions for convex functions can be formulated as objects of the PEPit.Constraint class:

```
# this is a PEPit.Expression object:
expr = f_x_0 + g_x_0 * (x_1 - x_0)
# those are two PEPit.Constraint objects:
cons_1 = (f_x_0 + g_x_0 * (x_1 - x_0) <= f_x_1)
cons_2 = (f_x_1 + g_x_1 * (x_0 - x_1) <= f_x_0)
```

Specifying constraints as done above, in a natural mathematical way, is convenient for the user. Once all required points, expressions, and constraints are associated with a PEP (as in the example of Sect. 2.2 or in the following lines), PEPit takes care of formulating the appropriate index sets I_1, and I_2 as well as all problem parameters for (11): $\{(A_i, a_i, \alpha_i)\}_{i\in I_1}$, and $\{(B_{j,k,i}, b_{j,k,i}, \beta_{j,k,i})\}_{1\leq j,k\leq n_i, i\in I_2}$ for feeding (11) to SDP solvers [5, 21, 26]. However, on the user side, specifying a large number of such constraints remains relatively cumbersome. Therefore, PEPit relies on a few additional structures and aliases that allow abstractly generating constraints blocks.

3.3 Main PEPit simplifying abstractions and aliases

Two key abstractions appearing in PEPit are the *functions* (which can also be manipulated algebraically) and *oracles* (or *primitive steps*, which are simple routines). In both cases, those structures were designed to allow users to easily add new types of functions and oracles.

PEPit *functions.* Among the most important building blocks simplifying the formulation of the constraints in (11), PEPit.Function objects are particularly important.

They allow for generating large numbers of constraints by remaining close to clean mathematical statements. Their most important characteristics are as follows:

- PEPit.Function() creates a new *leaf* function,
- Each leaf function contains a list of triplet $\{(x_i, g_i, f_i)\}_i$ which corresponds to the sampled version of the function in the form (points, gradients, function values), as presented in Sect. 2.
- Each PEPit.Function object F is featured with a F.add_point(triplet) method, which adds a triplet to the list of samples associated to F. By relying on appropriate abstractions (exemplified later), the users are expected to almost never use this method explicitly.
- Each PEPit.Function object F contains a F.add_class_constraints() method that generate the list of interpolation PEPit.Constraint associated to the sampled version of F. This method is never explicitly used by the user but allows PEPit to translate the mathematical statements (description of the function) to actual constraints.
- Each PEPit.Function can also be scaled by scalar values and added or subtracted to other PEPit.Function for creating new PEPit.Function objects.

Functions also feature a number of aliases that allow to easily create specific PEPit.Point and PEPit.Expression objects associated with the function (such as accessing the gradient/value at a specific point, or accessing an optimal point of a function).

Example 2 (Base operations with functions) This example shows that we can manipulate functions to create new functions.

```
f = PEPit.Function()  # a leaf PEPit.Function
h = PEPit.Function()  # a leaf PEPit.Function
F = 2 * f + h  # a PEPit.Function
```

We can also call base function operations on functions that are constructed by combining *leaf* functions.

```
x = PEPit.Point()  # this is p_1
g_x = PEPit.Point()  # this is p_2
f_x = PEPit.Expression()  # this is h_1

F.add_point((x, g_x, f_x))
# internally creates p_3 (a new point)
# and h_2 (a new expression),
# and adds one point to the list of f,
# and one to the list of h so that
# the weighted sums of gradients and
# function values at x are correct.
```

However, to simplify the usage of PEPit, one should essentially avoid directly using the method ADD_POINT, and rather rely on more readable statements. In that regard, the following piece of code is equivalent to the previous one but relies on more readable operations.

```
f = PEPit.Function()  # a leaf PEPit.Function
h = PEPit.Function()  # a leaf PEPit.Function
F = 2 * f + h  # a PEPit.Function

x = PEPit.Point()  #  p_1 (arbitrary point)
g_x = F.gradient(x)  # Define g_x  the gradient of F at x
f_x = F(x)  # f_x is F(x)
```

The list of points the function (or its gradient) is evaluated on is stored in the underlying object and finally used to generate the corresponding list of interpolation constraints. For creating a class that features a new type of interpolation conditions, it suffices to create a new class that inherits PEPit.Function and implements the operation ADD_CLASS_CONSTRAINTS with the appropriate interpolation constraints (see https://github.com/PerformanceEstimation/PEPit/tree/master/PEPit/functions for examples).

Let us now mention an important class of convenient abstraction that belongs to the heart of PEPit's philosophy.

PEPit *primitive steps (or oracles).* PEPit *primitive steps* (or *oracles*) are essentially simple aliases defining notational shortcuts for some algorithmic operations, rendering them closer to their mathematical abstractions. They generally consist of the appropriate creations (and constraining) of points and expressions, for instance by adding appropriate triplets to sampled versions of the functions under consideration. Let us provide a few examples.

Example 3 (Proximal operators) In this example, we provide two ways to use proximal operators within PEPit. The first one uses the base PEPit methods for doing this, and the second one relies on the more readable PROXIMAL_STEP that is provided as a primitive step of PEPit. Those two ways are computationally equivalent, though different in terms of readability. Let us recall that the proximal operation (with unit step-size) associated to f is given by

$$x_1 = \text{prox}_f(x_0) \triangleq \text{argmin}_x \left\{ f(x) + \tfrac{1}{2}\|x - x_0\|^2 \right\}.$$

For any (closed, proper) convex function f, this operation is well-defined and amounts to an *implicit subgradient* operation:

$$x_1 = x_0 - g_1,$$

with $g_1 \in \partial f(x_1)$.

```
# f is a (closed, proper) convex function
f = PEPit.functions.ConvexFunction()
x0 = PEPit.Point()  # is a point

# how to construct x_1 = prox_f (x_0)?
# (i) initiate some g1 and f1
g1 = PEPit.Point()
f1 = PEPit.Expression()

# (ii) form x1 using x0 and g1
x1 = x0 - g1

# (iii) constrain g1 to be a subgradient of
# f at x1
f.add_point((x1, g1, f1))
```

Equivalently, using PROXIMAL_STEP, we have:

```
# f is a (closed, proper) convex function
f = PEPit.functions.ConvexFunction()
x0 = PEPit.Point()  # is a point

x1, g1, f1 = proximal_step(x0, f, gamma=1)
# note: gamma is a step-size, set to 1
# in the example.
```

Example 4 (Linear optimization oracles) This operation is at the core of the Frank-Wolfe (a.k.a. conditional gradient) method, and consists in, given a (closed, convex) domain $\mathcal{K}$ (whose indicator function is denoted by $i_{\mathcal{K}}$) and a search direction d_0, in computing a solution to

$$x_1 \in \operatorname{argmin}_x d_0^T x + i_{\mathcal{K}}(x),$$

which is mathematically equivalent to writing $-d_0 \in \partial i_{\mathcal{K}}(x_1)$. In PEPit, this can also easily be coded as follows.

```
# ind is a (closed) convex indicator function
ind = PEPit.functions.ConvexIndicatorFunction()
d0 = PEPit.Point()

# how to construct a solution to
# min_x d0*x + ind(x) ?
# (i) Initiate a new point and an expression
x1 = PEPit.Point()
f1 = PEPit.Expression()
```

```
# (ii) Constrain -d0 to be a subgradient
# of the indicator at x1
ind.add_point((x1, -d0, f1))
```

Using abstraction again, this code is equivalent to

```
# ind is a (closed) convex indicator function
ind = PEPit.functions.ConvexIndicatorFunction()
d0 = PEPit.Point()

# how to construct a solution to
# min_x d0*x + ind(x) ?
x1, _, f1 = linear_optimization_step(d0, ind)
```

Example 5 (Using approximate gradients) Another standard operation in first-order optimization consists in using *approximate* gradient values. For instance, for computing some x_1 using a gradient iteration (with unit step-size) with an approximate gradient $\tilde{d}_0 \approx \nabla f(x_0)$ for some appropriate function f:

$$x_1 = x_0 - \tilde{d}_0,$$

with $\tilde{d}_0$ being an ϵ approximation to $\nabla f(x_0)$ in the following sense: $\|\tilde{d}_0 - \nabla f(x_0)\| \leq \epsilon \|\nabla f(x_0)\|$, say with $\epsilon = 0.1$.

```
# f is a 1-smooth convex function
f = PEPit.functions.SmoothConvexFunction(L=1)
x0 = PEPit.Point()
epsilon = .1

# how to construct a x1?
# (i) initiate a d0 and the gradient of x0
g0 = f.gradient(x0)
d0 = PEPit.Point()
f.add_constraint((d0 - g0) ** 2 <=
 epsilon * g0 ** 2)

x1 = x0 - d0
```

This can equivalently be done using INEXACT_GRADIENT_STEP, as follows.

```
# f is a 1-smooth convex function
f = PEPit.functions.SmoothConvexFunction(L=1)
x0 = PEPit.Point()

# how to construct a x1?
from PEPit.primitive_steps\
    import inexact_gradient_step
```

```
x1, d0, _ = inexact_gradient_step(x0, f,
                                  gamma=1,
                                  epsilon=.1,
                                  notion=
                                  'relative')
```

To conclude this section, PEPit's philosophy is to contain many *abstract* routines that can be associated with simple mathematical statements. Taken separately, those routines are relatively simple and can easily be created or modified by the users.

3.4 The objective function of the PEP: performance metrics

A key point that we did not mention so far concerns the objective function of (11). It is handled by what is referred to as *performance metrics* in PEPit. In the SDP formulation (11), they correspond to the index set I_1 and the set of parameters $\{(A_i, a_i, \alpha_i)\}_{i\in I_1}$, and they are handled by calling the SET_PERFORMANCE_METRIC method (which takes a PEPit.Expression as sole argument) associate to a PEP object. By introducing the variable τ in (11), the objective of the PEP corresponds to the minimum value of all specified performance metrics.

Example 6 This example shows how to specify an objective function (a.k.a. *performance metric*) within PEPit.

```
problem = PEPit.PEP()

p1 = PEPit.Point()
p2 = PEPit.Point()
h1 = PEPit.Expression()

# set the PEP objective as the minimum value
# among ||p1||^2, ||p2||^2, and h1.
problem.set_performance_metric(p1**2)
problem.set_performance_metric(p2**2)
problem.set_performance_metric(h1)
```

3.5 Formulating and solving the PEP

As a final stage for formulating (11), we need to gather all functions and constraints together and reformulate in terms of (11). PEPit performs it through the use of the PEP object (as provided in Sect. 2.2) as follows:

```
problem = PEPit.PEP()
```

which is used for centralizing all the information about the PEP at hand. In particular, it is a good practice to avoid using PEPIT.POINT() directly, and to rather use PROBLEM.SET_INITIAL_POINT(). Similarly, new functions should be declared through the PEP object using the DECLARE_FUNCTION method:

```
# declares a convex (closed, proper) function h
h = problem.declare_function(ConvexFunction)

# declares a convex (closed) indicator ind
ind = problem.declare_function(
      ConvexIndicatorFunction)

# declares a 1-smooth convex function f
f = problem.declare_function(
    SmoothConvexFunction, L=1)
```

To model (11), we gather all PEPIT.CONSTRAINT objects that are associated with PROBLEM through the different abstractions used in the code. For instance, the PEPIT.PEP object PROBLEM will call the ADD_CLASS_CONSTRAINTS method of all functions involved in the PEP. Once this is done, PEPIT generates all appropriate matrices and index sets to model the problem as an SDP in the form (11), and passes it to either CVXPY [5] or directly to MOSEK [21], before performing a few post-processing steps.

3.6 Post-processing

Once the PEP is solved numerically, we need to manipulate its output, either for constructing proofs (on the dual PEP side), or counter-examples (on the primal PEP side).

Dual reconstructions. Dual values associated with the different constraints play an important role, as they allow us to construct mathematical proofs.

In the following, we assume that *func* has been defined as a Function used in the PEP, that *constraints_list* has been defined as a list of Constraint objects involved in the PEP, and that the PEP has been solved through the PEP.SOLVE() method.

One of the important features of PEPit is to automatically deal with interpolation constraints for the different functions involved in the PEP. This way, the attribute *list_of_class_constraints* enables accessing the list of Constraint objects encoding the interpolation constraints of *func*.

Each dual value can be accessed through the EVAL_DUAL() method of the associated Constraint object:

```
for constraint in\
      func.list_of_class_constraints:
    dual_value = constraint.eval_dual()
    print(dual_value)
```

Moreover, to ease the reconstruction of the proof, and avoid mistakes by associating a constraint to the wrong dual value, a user can name a constraint through the SET_NAME() method and access it later on through the GET_NAME() method. Note it is the responsibility of the user to set a name to the constraints to be able to recover it later.

```
for constraint in constraints_list:
    constraint_name = constraint.get_name()
    dual_value = constraint.eval_dual()
    print(constraint_name, dual_value)
```

Since PEPit deals with interpolation constraints without any intervention from the user side, a short description of each of those constraints is set by default, based on Points' names and *func*'s name which can also be set through a SET_NAME method. Finally, a user can also obtain all the dual values associated with the interpolation constraints of a function at once, using the GET_CLASS_CONSTRAINTS_DUALS method as

```
# assuming func has been defined
# as a Function object involved in the PEP
# and that the PEP has been solved.

tables = func.get_class_constraints_duals()
```

This method returns a dictionary whose values are pandas.DataFrames, offering great readability. Completing the example of Sect. 2.2 (this time for 2 steps) with this feature, we obtain the following code:

```
import numpy as np

from PEPit import PEP
from PEPit.functions import\
    SmoothStronglyConvexFunction

# We set the parameter of the problem
# Here we study the contraction of 1 step of
# the GD method with step 1/L on the class
# of L=1 smooth and mu=.1 strongly
# convex functions.

L = 1.              # Smoothness parameter
mu = .1             # Strong convexity parameter
gamma = 1. / L      # Step size
n = 2               # Number of iterations

# Instantiate the PEP object
problem = PEP()

```

```
# Declare an L-smooth mu-strongly convex
# function named "func"
func = problem.declare_function(
    SmoothStronglyConvexFunction,
    mu=mu,      # Strong convexity param.
    L=L,        # Smoothness param.
    name="f")   # Name

# Declare two starting points
x_0 = problem.set_initial_point(name="x_0")
y_0 = problem.set_initial_point(name="y_0")

# Initial condition ||x_0 - y_0||^2 <= 1
problem.set_initial_condition((x_0 - y_0) ** 2
 <= 1)

# Initialize the algorithm
x = x_0
y = y_0
# Run n steps of the GD method for the
# two sequences
for i in range(n):

    # Replace x and y with their next iterates
    x = x - gamma * func.gradient(x) # f'(x)
    x.set_name("x_{}".format(i+1))

    y = y - gamma * func.gradient(y) # f'(y)
    y.set_name("y_{}".format(i+1))

# Set the performance metric to the distance
# ||x_n - y_n||^2
problem.set_performance_metric((x-y)**2)

# Solve the PEP
pepit_tau = problem.solve()

# By linearly combining the interpolation
# constraints with the right coefficients,
# we can prove
# ||x_n - y_n||^2 <= pepit_tau ||x_0 - y_0||^2
# The coefficient we need are the dual values
# of the interpolation constraints of func.
tables = func.get_class_constraints_duals()

# A user can display the dictionary as is,
```

```
# or access one specific table by their name.
# Those names are intuitive, yet to be known,
# for example by displaying the keys of tables.
# Here we use the only key of this dictionary.
table = tables["smoothness_strong_convexity"]

print("\nDual values associated with"
      " interpolation constraints:")
print(table.astype(dtype=np.float16))
```

Running this code outputs the following message:

```
(PEPit) Setting up the problem: size of the Gram matrix: 6x6
(PEPit) Setting up the problem: performance measure is the
                                minimum of 1 element
(PEPit) Setting up the problem: Adding initial conditions
                                and general constraints ...
(PEPit) Setting up the problem: initial conditions
                                and general constraints
                                (1 constraint(s) added)
(PEPit) Setting up the problem: interpolation conditions
                                for 1 function(s)
            Function 1 : Adding 12 scalar constraint(s) ...
            Function 1 : 12 scalar constraint(s) added
(PEPit) Setting up the problem: additional constraints
                                for 0 function(s)
(PEPit) Compiling SDP
(PEPit) Calling SDP solver
(PEPit) Solver status: optimal (wrapper:cvxpy, solver: MOSEK);
                       optimal value: 0.6561000087150534
(PEPit) Primal feasibility check:
    The solver found a Gram matrix that is positive semi-definite
        up to an error of 2.584803430062655e-09
    All the primal scalar constraints are verified
        up to an error of 4.590572921792102e-09
(PEPit) Dual feasibility check:
    The solver found a residual matrix that is positive semi-definite
    All the dual scalar values associated with inequality constraints
        are nonnegative up to an error of 3.8932973849815053e-10
(PEPit) The worst-case guarantee proof is perfectly reconstituted
        up to an error of 1.349705540942825e-07
(PEPit) Final upper bound (dual): 0.6561000067100656
        and lower bound (primal example): 0.6561000087150534
(PEPit) Duality gap: absolute: -2.004987731396568e-09
                     and relative: -3.055917855150253e-09

Dual values associated with interpolation constraints:
IC_f       x_0       y_0       x_1       y_1
x_0   0.000000  1.458008  0.000000  0.000000
y_0   1.458008  0.000000  0.000000  0.000000
x_1  -0.000000  0.000000  0.000000  1.799805
y_1   0.000000 -0.000000  1.799805  0.000000
```

Primal reconstructions. In order to construct an example of a problem on which the algorithm behaves as poorly as possible with respect to the given performance metrics,

all PEPit.Point, all PEPit.Expression and all PEPit.Constraint objects can be conveniently evaluated through the eval() method. Moreover, their name attribute can help to sort them. PEPit also offers the possibility to search for simpler, potentially low-dimensional problem instances via the trace norm [28] or the logdet [9] heuristics (aiming at finding low-rank feasible matrices G for the problem (11) while keeping the same objective value). Those post-processing steps can be accessed via the option of the solve method (see https://pepit.readthedocs.io/en/latest/api/main_modules.html#pep). Examples of such usages can be found in the documentation at https://pepit.readthedocs.io/en/latest/examples/j.html.

4 PEPit: general overview and content

In this section, we go back to the mathematical content of the toolbox and describe the various choices of (i) elementary oracles used in algorithms, (ii) problem or function classes, (iii) performance measures, and (iv) initial conditions, that are naturally handled by PEPit. PEPit also allows studying methods for monotone inclusions and fixed point problems, but we do not cover them in this summary. In the optimization setting, the minimization problem under consideration has the form

$$F_\star \triangleq \min_{x\in\mathbb{R}^d} \left\{ F(x) \equiv \sum_{i=1}^{K} f_i(x) \right\}, \tag{13}$$

for some $K \in \mathbb{N}$ and where each f_i is assumed to belong to some class of functions denoted by $\mathcal{F}_i$, encoding our working assumptions on f_i. We further assume the algorithms to gather information about the functions $\{f_i\}_i$ only via black-box oracles such as gradient or proximal evaluations.

Black-box oracles. The base black-box optimization oracles/operations available in PEPit are the following:

- (sub)gradient steps,
- proximal and projection steps,
- linear optimization (or conditional) steps.

PEPit also allows for their slightly more general approximate/inexact and Bregman (or mirror) versions. Those oracles might be combined with a few other operations, such as exact line-search procedures, as detailed in Table 1.

Problem classes. A few base classes of functions are readily available within the package (see Table 2 for further details) such as:

- Convex functions within different classes of assumptions possibly involving bounded gradients (Lipschitz functions), quadratically upper bounded functions, bounded domains, smoothness, and/or strong convexity. Those assumptions might be combined when compatible.
- Convex indicator functions, possibly with a bounded domain.
- Smooth nonconvex functions.
- Quadratic functions.

Table 1 Main base primitive steps (oracles) included in PEPIT

Oracle name	Description	Tightness
Gradient step	$x_+ = x - \gamma g$ with $g = \nabla f(x)$	✔
Subgradient step	$x_+ = x - \gamma g$ with $g \in \partial f(x)$	✔
Epsilon-subgradient step	$x_+ = x - \gamma g$ with $g \in \partial_\varepsilon f(x)$	✔
Inexact gradient step	$x_+ = x - \gamma g$ with $g \approx_\epsilon \nabla f(x)$ for some notion "$\approx_\epsilon$" of approximation	✔
Exact line-search step	$x_+ = \underset{z \in x + \text{span}\{d_i, i \in [\![1,T]\!]\}}{\arg\min} f(z)$	✖
Proximal step	$x_+ = \underset{z}{\arg\min} \left\{ \gamma f(z) + \frac{1}{2}\|z - x\|^2 \right\}$	✔
Inexact proximal step	$x_+ \approx_\epsilon \underset{z}{\arg\min} \left\{ \gamma f(z) + \frac{1}{2}\|z - x\|^2 \right\}$ for some notion "$\approx_\epsilon$" of approximation	✔
Bregman gradient step	$x_+ = \underset{z}{\arg\min} \left[\langle \nabla f(x); z - x \rangle + \frac{1}{\gamma} D_h(z; x) \right]$	✔
Bregman proximal step	$x_+ = \underset{z}{\arg\min} \left[f(z) + \frac{1}{\gamma} D_h(z; x) \right]$	✔
Linear optimization step	$x_+ = \underset{z \mid \text{ind}(z)=0}{\arg\min} \langle \text{dir}; z \rangle$	✔

Appropriate references are provided in the corresponding documentation. Some oracles are overlapping and are present for promoting a better readability of the code and for numerical efficiency. Variations around the present oracles can be added at will. For each oracle, x_+ denotes the output of the oracles; the other elements are either input of the oracles or intermediary optimization variables

Table 2 Some base function classes included in PEPIT, detailed in the documentation

Function class name	Tightness
Convex (closed, proper) functions	✔
Convex (closed, proper) Lipschitz-continuous functions	✔
Convex (closed, proper) indicator functions	✔
Convex support functions	✔
Smooth strongly convex functions	✔
Smooth convex functions	✔
Smooth (possibly nonconvex) functions	✔
Smooth convex Lipschitz functions	✔
Strongly convex functions	✔
Convex quadratically upper-bounded functions	✔
Restricted secant inequality and error bound	✔
Smooth strongly convex quadratic functions	✔
Smooth convex function by block	✖

Default functional classes within PEPIT. Some classes are overlapping and are present only to promote a better readability of the code

Table 3 Base operator classes within PEPIT, detailed in the documentation

Operator class name	Tightness
Monotone (maximally)	✔
Strongly monotone (maximally)	✔
Cocoercive	✔
Lipschitz continuous	✔
Negative comonotone	✔
Cocoercive and strongly monotone	✖
Lipschitz continuous and strongly monotone	✖
Non-expansive	✔
Linear	✔
Symmetric Linear	✔
Skew-symmetric Linear	✔

Some classes are overlapping and are present only to promote a better readability of the code. Note that, for some classes, the associated constraints might not be tight, meaning that the methodology might only be able to generate upper-bound on the worst-case behaviors

Beyond the pure optimization setting, PEPIT also allows using operators (see Table 3) within different classes of assumptions (namely: nonexpansive, Lipschitz, cocoercive, maximally monotone, and strongly monotone operators) for studying first-order methods for monotone inclusions and variational inequalities.
Performance measures and initial conditions. An important degree of freedom of the package is to allow a large panel of performance measures and initial conditions. Essentially, everything that can be expressed linearly (or slightly beyond) in function values and quadratically in gradient/iterates (i.e., linear in the Gram representation of Sect. 2) might be considered. Following the notation of Sect. 2.2 (and denoting by $x_\star$ an optimal point of F), typical examples of initial conditions include:

- $\|x_0 - x_\star\|_2^2 \leqslant 1$,
- $\|\nabla F(x_0)\|_2^2 \leqslant 1$ (when F is differentiable, otherwise similar criterion involving some subgradient of F might be used),
- $F(x_0) - F(x_\star) \leqslant 1$,
- any linear combination of the above (see, e.g., examples in the *potential functions* folder of the package).

Similarly, typical examples of performance measures (see Table 4 for examples) include: $\|x_n - x_\star\|_2^2$, $\|\nabla F(x_n)\|_2^2$ (when F is differentiable), $F(x_n) - F(x_\star)$, or linear combinations and minimum values of the above,
e.g. $\min_{0 \leqslant i \leqslant n} \|\nabla F(x_i)\|_2^2$ (when F is differentiable).
Examples. PEPIT contains about 75 examples that can readily be used, instantiating the different sets of black-box oracles, problem classes, and initial condition/performance measures. Those examples can be found in the folder PEPIT/EXAMPLES/.

Contributing. PEPIT is designed to allow users to easily contribute to add features to the package. Classes of functions (or operators) as well as black-box oracles can be implemented by following the contributing guidelines from the documentation. We

Table 4 Examples of common performance measures

Performance measure	Description
Distance	$\|x_n - x_\star\|_2^2$
Gradient norm	$\|\nabla F(x_n)\|_2^2$
Function value	$F(x_n) - F(x_\star)$
Contraction	$\|x_n - y_n\|_2^2$
Lyapunov functions	$a(F(x_n) - F(x_\star)) + \begin{pmatrix} x_n - x_\star \\ \nabla F(x_n) \end{pmatrix}^\top (P \otimes I_d) \begin{pmatrix} x_n - x_\star \\ \nabla F(x_n) \end{pmatrix}$

This topic is discussed extensively in excellent references that include [22, 23]; see also [37, Tables 1–3] for examples in the context of (proximal) gradient descent. Considering appropriate performance measures is key for the analyses, and is particularly exploited when looking for appropriate Lyapunov functions, see, e.g., the related [19, 31, 33]

also welcome any new example for analyzing a method/setting that is not already present in the toolbox.

5 A few additional numerical examples

The following section provides a few additional numerical worst-case analyses obtained through PEPit; namely an accelerated gradient method [25], an accelerated Douglas-Rachford splitting [27], and point-SAGA [4] (a proximal method for finite-sum minimization).

5.1 Analysis of an accelerated gradient method

For this example, we focus again on the problem of minimizing a L-smooth μ-strongly convex function (problem (13) with $F \in \mathcal{F}_{\mu,L}$). We consider a classical accelerated gradient method with constant momentum [24, 25]. It can be described as follows for $t \in \{0, ..., n-1\}$ with $y_0 = x_0$:

$$\begin{aligned} x_{t+1} &= y_t - \alpha \nabla F(y_t), \\ y_{t+1} &= x_{t+1} + \beta(x_{t+1} - x_t), \end{aligned} \tag{14}$$

with $\kappa = \frac{\mu}{L}$, $\alpha = \frac{1}{L}$ and $\beta = \frac{1-\sqrt{\kappa}}{1+\sqrt{\kappa}}$. We decide to compute the smallest possible $\tau(n, L, \mu)$ such that the guarantee

$$F(x_n) - F_\star \leqslant \tau(n, L, \mu) \left(F(x_0) - F(x_\star) + \frac{\mu}{2} \|x_0 - x_\star\|_2^2 \right),$$

holds for all $d \in \mathbb{N}$, $F \in \mathcal{F}_{\mu,L}$, $x_0, x_n, x_\star \in \mathbb{R}^d$ where x_n is an output of the accelerated gradient method (14) and $x_\star$ is the minimizer of F. In this setting, $\tau(n, L, \mu)$ can be computed as the worst-case value of $F(x_n) - F_\star$ (the performance metric) when $F(x_0) - F_\star + \frac{\mu}{2}\|x_0 - x_\star\|_2^2 \leqslant 1$ (initial condition). As a reference, we compare the

output of PEPit to the following worst-case guarantee [8, Corollary 4.15]:

$$F(x_n) - F_\star \leqslant \left(1 - \sqrt{\frac{\mu}{L}}\right)^n \left(F(x_0) - F_\star + \frac{\mu}{2}\|x_0 - x_\star\|_2^2\right). \tag{15}$$

A comparison between the output of PEPit and (15) is presented in Fig. 2a, where we see that (15) could be slightly improved to better match the worst-case behavior of the algorithm. The corresponding code can be found in accelerated_gradient_strongly_convex.py from the PEPit/examples/unconstrained_convex_minimization/ directory.

5.2 Analysis of an accelerated Douglas-Rachford splitting

In this section, we provide a simple PEPit example for studying an accelerated Douglas-Rachford splitting method. This method was introduced in [27] where a worst-case analysis is provided for quadratic minimization. We perform a worst-case analysis numerically for a slightly more general setting:

$$F_\star \triangleq \min_x \{F(x) \equiv f_1(x) + f_2(x)\},$$

where f_1 is closed proper and convex, and f_2 is μ-strongly convex and L-smooth. This section focuses on the following accelerated Douglas-Rachford splitting method, described in [27, Sect. 4]:

$$\begin{aligned} x_t &= \text{prox}_{\alpha f_2}(u_t), \\ y_t &= \text{prox}_{\alpha f_1}(2x_t - u_t), \\ w_{t+1} &= u_t + \theta(y_t - x_t), \\ u_{t+1} &= \begin{cases} w_{t+1} + \frac{t-1}{t+2}(w_{t+1} - w_t) & \text{if } t \geqslant 1, \\ w_{t+1} & \text{otherwise,} \end{cases} \end{aligned}$$

where prox denotes the usual proximal operator, available in PEPit through the operation PROXIMAL_STEP, as exemplified below. Note that we only show the algorithm description here, the full PEPit code for this example can be found in the file accelerated_douglas_rachford_splitting.py from the directory PEPit/examples/composite_convex_minimization/.

```
# Compute n steps of
# An accelerated Douglas-Rachford splitting
for t in range(n):
 x[t], _, _ = proximal_step(u[t], func2, alpha)
 y, _, fy = proximal_step(2*x[t]  - u[t], func1,
                          alpha)
 w[t+1] = u[t] + theta * (y-x[t])
 if t >= 1:
  u[t+1] = w[t+1] + (t-1)/(t+2) * (w[t+1]-w[t])
```

```
10  else:
11    u[t+1] = w[t+1]
```

When f_2 is a L-smooth μ-strongly convex quadratic function, the following worst-case guarantee is provided by [27, Theorem 5]:

$$F(y_n) - F_\star \leqslant \frac{2\|w_0 - w_\star\|_2^2}{\alpha\theta(n+3)^2}, \tag{16}$$

when $\theta = \frac{1-\alpha L}{1+\alpha L}$ and $\alpha < \frac{1}{L}$. A numerical worst-case guarantee for the case $L = 1$, $\mu = 0.01$, $\alpha = 0.9$, $\theta = \frac{1-\alpha L}{1+\alpha L}$ is provided on Fig. 2b for a few different values of N, where we use (16) as a reference for comparison. For each of those values, PEPIT computed a tight (up to numerical precision) worst-case value for which we are not aware of any proven analytical worst-case guarantee beyond the quadratic setting. We see that PEPIT provides an improvement over this guarantee, even when the problem under consideration is not quadratic.

5.3 Analysis of point-SAGA

In this section, we use PEPIT for studying point-SAGA [4], a stochastic algorithm for finite sum minimization:

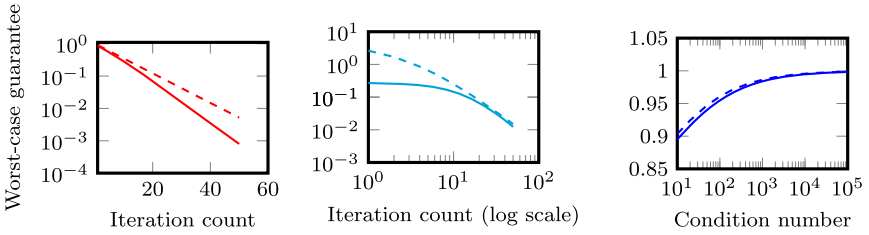
$$F_\star \triangleq \min_x \left\{ F(x) \equiv \frac{1}{n}\sum_{i=1}^{n} f_i(x) \right\},$$

where $f_1, \ldots, f_n$ are L-smooth and μ-strongly convex functions with a proximal operator available for each of them. At each iteration t, point-SAGA picks $j_t \in \{1, \ldots, n\}$ uniformly at random and performs the following updates (a superscript is used for denoting iteration numbers; the subscript is used for referring to the function f_{j_t} chosen uniformly at random):

$$\begin{aligned} z_{j_t}^{(t)} &= x^{(t)} + \gamma\left(g_{j_t}^{(t)} - \frac{1}{n}\sum_i g_i^{(t)}\right), \\ x_{j_t}^{(t+1)} &= \text{prox}_{\gamma f_{j_t}}\left(z_{j_t}^{(t)}\right), \\ g_{j_t}^{(t+1)} &= \frac{1}{\gamma}\left(z_{j_t}^{(t)} - x_{j_t}^{(t+1)}\right), \end{aligned}$$

where $\gamma = \frac{\sqrt{(n-1)^2+4n\frac{L}{\mu}}}{2Ln} - \frac{\left(1-\frac{1}{n}\right)}{2L}$ is the step-size. In this example, we use a Lyapunov (or potential / energy) function $V(x) = \frac{1}{L\mu}\frac{1}{n}\sum_{i\leqslant n}\|\nabla f_i(x) - \nabla f_i(x_\star)\|_2^2 + \|x - x_\star\|_2^2$, and compute the smallest $\tau(n, L, \mu)$ such that the guarantee

$$\mathbb{E}_{j_t}\left[V\left(x_{j_t}^{(t+1)}\right)\right] \leqslant \tau(n, L, \mu)V(x^{(t)}),$$



(a) Accelerated gradient method: strong convexity parameter fixed to $\mu = 0.1$. Worst-case guarantee on $F(x_n) - F_\star$ as a function of n.

(b) Accelerated Douglas-Rachford splitting with parameter $\alpha = 0.9$ and $\mu = 0.1$. Worst-case guarantee on $F(y_n) - F_\star$ as a function of n.

(c) Point-SAGA: $n = 5$ functions. Worst-case guarantee on $E_{j_t}[V(x_{j_t}^{(t+1)})]$ as a function of the condition number $\kappa = \frac{L}{\mu}$.

Fig. 2 Comparisons between (numerical) worst-case bounds from PEPIT (*plain lines*) VS. reference established worst-case guarantees (*dashed lines*) for three different optimization methods. For simplicity, we fixed smoothness constants to $L = 1$

holds for all $d \in \mathbb{N}$, $f_i \in \mathcal{F}_{\mu,L}(\mathbb{R}^d)$ (for all $i = 1, \dots, n$), $x^{(t)} \in \mathbb{R}^d$ where $x_{j_t}^{(t+1)}$ is the (random) output generated by point-SAGA, and the expectation is taken over the randomness of j_t (and computed in practice as an average of all the possible j_t). The following simple worst-case guarantee is provided in [4, Theorem 5] and is used as a reference:

$$\mathbb{E}_{j_t}[V(x_{j_t}^{(t+1)})] \leqslant \frac{1}{1+\mu\gamma} V(x^{(t)}). \tag{17}$$

We compare (17) to PEPIT's tight (up to numerical precision) output in Fig. 2c. We see that the worst-case guarantee (17) can be slightly improved, although pretty accurate, particularly for large values of the condition number. The corresponding PEPIT code of this example can be found in the file point_saga.py from the directory PEPit/examples/stochastic_and_randomized_convex_minimization/.

6 Conclusion

The PEPIT package, briefly described in this paper, aims at providing simplified access to worst-case analyses of first-order optimization methods in PYTHON. To achieve this goal, PEPIT implements the performance estimation approach while allowing to avoid the possibly heavy semidefinite programming modeling steps. The first version of the package already contains about 75 examples of first-order methods that can be analyzed through this framework. Those examples allow either reproducing or tightening, numerically, known worst-case analyses or provide new ones depending on the particular method and problem class at hand.

Overall, we believe that this package allows quick (in)validations of proofs (a step towards reproducible theory) which should help both the development and the review process in optimization. We also argue that this is a nice pedagogical tool for learning algorithms together with their worst-case properties just by playing with them. Possible extensions under consideration for future versions include an option for searching for

Lyapunov (or potential/energy) functions [31, 33, 38], for disproving convergence [13], as well as a numerical proof assistant, and to incorporate recent extensions of PEP/IQCs to distributed and decentralized optimization [3, 30].

Acknowledgements The work of B. Goujaud and A. Dieuleveut is partially supported by ANR-19-CHIA-0002-01/chaire SCAI, and Hi!Paris. A. Taylor acknowledges support from the European Research Council (grant SEQUOIA 724063). This work was partly funded by the French government under the management of Agence Nationale de la Recherche as part of the "Investissements d'avenir" program, reference ANR-19-P3IA-0001 (PRAIRIE 3IA Institute). The authors would like to deeply thank the two anonymous referees as well as a technical editor for extremely constructive feedback that contributed to improving this work as well as the related package.

Funding The work of B. Goujaud and A. Dieuleveut is partially supported by ANR-19-CHIA-0002-01/chaire SCAI, and Hi Paris. A. Taylor acknowledges support from the European Research Council (grant SEQUOIA 724063). This work was partly funded by the French government under the management of Agence Nationale de la Recherche as part of the "Investissements d'avenir" program, reference ANR-19-P3IA-0001 (PRAIRIE 3IA Institute).

Availability of data and materials No data has been used in this publication, every displayed result has been generated using our code PEPit.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

Code availability. The full code is publicly available at https://github.com/PerformanceEstimation/PEPit, downloadable using the shell command line PIP INSTALL PEPIT, and its documentation can be found at https://pepit.readthedocs.io.

References

1. Abbaszadehpeivasti, H., de Klerk, E., Zamani, M.: The exact worst-case convergence rate of the gradient method with fixed step lengths for L-smooth functions. Optim. Lett. **16**(6), 1649–1661 (2022)
2. Bousselmi, N., Hendrickx, J.M., Glineur, F.: Interpolation conditions for linear operators and applications to performance estimation problems. arXiv:2302.08781 (2023)
3. Colla, S., Hendrickx, J.M.: Automated worst-case performance analysis of decentralized gradient descent. In: Proceedings of the 60th Conference on Decision and Control (CDC) (2021)
4. Defazio, A.: A simple practical accelerated method for finite sums. In: Advances in Neural Information Processing Systems (NIPS) (2016)
5. Diamond, S., Boyd, S.: CVXPY: a Python-embedded modeling language for convex optimization. J. Mach. Learn. Res. (JMLR) **17**(1), 2909–2913 (2016)
6. Drori, Y.: Contributions to the complexity analysis of optimization algorithms. Ph.D. thesis, Tel-Aviv University (2014)
7. Drori, Y., Teboulle, M.: Performance of first-order methods for smooth convex minimization: a novel approach. Math. Program. **145**(1), 451–482 (2014)
8. d'Aspremont, A., Scieur, D., Taylor, A.: Acceleration methods. Foundations and Trends® in Optimization **5**(1–2), 1–245 (2021)
9. Fazel, M., Hindi, H., Boyd, S.P.: Log-det heuristic for matrix rank minimization with applications to Hankel and Euclidean distance matrices. In: American Control Conference (ACC), vol. 3, pp. 2156–2162. IEEE (2003)
10. Fazlyab, M., Ribeiro, A., Morari, M., Preciado, V.M.: Analysis of optimization algorithms via integral quadratic constraints: nonstrongly convex problems. SIAM J. Optim. **28**(3), 2654–2689 (2018)

11. Gannot, O.: A frequency-domain analysis of inexact gradient methods. Math. Program. **194**(1), 975–1016 (2022)
12. Gorbunov, E., Loizou, N., Gidel, G.: Extragradient method: $o(1/k)$ last-iterate convergence for monotone variational inequalities and connections with cocoercivity. In: International Conference on Artificial Intelligence and Statistics (AISTATS), pp. 366–402 (2022)
13. Goujaud, B., Dieuleveut, A., Taylor, A.: Counter-examples in first-order optimization: a constructive approach. IEEE Control Syst. Lett. **7**, 2485–2490 (2023)
14. Gu, G., Yang, J.: Tight sublinear convergence rate of the proximal point algorithm for maximal monotone inclusion problems. SIAM J. Optim. **30**(3), 1905–1921 (2020)
15. Hu, B., Wright, S., Lessard, L.: Dissipativity theory for accelerating stochastic variance reduction: a unified analysis of SVRG and Katyusha using semidefinite programs. In: International Conference on Machine Learning (ICML) (2018)
16. Kim, D.: Accelerated proximal point method for maximally monotone operators. Math. Program. **190**(1), 57–87 (2021)
17. Kim, D., Fessler, J.A.: Optimized first-order methods for smooth convex minimization. Math. Program. **159**(1–2), 81–107 (2016)
18. Kim, D., Fessler, J.A.: Optimizing the efficiency of first-order methods for decreasing the gradient of smooth convex functions. J. Optim. Theory Appl. **188**(1), 192–219 (2021)
19. Lessard, L., Recht, B., Packard, A.: Analysis and design of optimization algorithms via integral quadratic constraints. SIAM J. Optim. **26**(1), 57–95 (2016)
20. Lieder, F.: On the convergence rate of the Halpern-iteration. Optim. Lett. **15**(2), 405–418 (2021)
21. Mosek, A.: The MOSEK optimization software. Online at http://www.mosek.com 54 (2010)
22. Nemirovski, A.: Information-based complexity of convex programming. Lecture notes. http://www2.isye.gatech.edu/~nemirovs/Lec_EMCO.pdf (1994)
23. Nemirovsky, A.S.: Information-based complexity of linear operator equations. J. Complex. **8**(2), 153–175 (1992)
24. Nesterov, Y.: A method of solving a convex programming problem with convergence rate $O(1/k^2)$. Soviet Math. Doklady **27**(2), 372–376 (1983)
25. Nesterov, Y.: Introductory Lectures on Convex Optimization. Springer, Cham (2003)
26. O'Donoghue, B., Chu, E., Parikh, N., Boyd, S.: Conic optimization via operator splitting and homogeneous self-dual embedding. J. Optim. Theory Appl. **169**(3), 1042–1068 (2016)
27. Patrinos, P., Stella, L., Bemporad, A.: Douglas–Rachford splitting: complexity estimates and accelerated variants. In: Proceedings of the 53rd Conference on Decision and Control (CDC) (2014)
28. Recht, B., Fazel, M., Parrilo, P.A.: Guaranteed minimum-rank solutions of linear matrix equations via nuclear norm minimization. SIAM Rev. **52**(3), 471–501 (2010)
29. Ryu, E.K., Taylor, A.B., Bergeling, C., Giselsson, P.: Operator splitting performance estimation: tight contraction factors and optimal parameter selection. SIAM J. Optim. **30**(3), 2251–2271 (2020)
30. Sundararajan, A., Van Scoy, B., Lessard, L.: Analysis and design of first-order distributed optimization algorithms over time-varying graphs. IEEE Trans. Control Netw. Syst. **7**(4), 1597–1608 (2020)
31. Taylor, A., Bach, F.: Stochastic first-order methods: non-asymptotic and computer-aided analyses via potential functions. In: Proceedings of the 32nd Conference on Learning Theory (COLT) (2019)
32. Taylor, A., Drori, Y.: An optimal gradient method for smooth strongly convex minimization. Math. Program. **199**(1–2), 557–594 (2023)
33. Taylor, A., Van Scoy, B., Lessard, L.: Lyapunov functions for first-order methods: tight automated convergence guarantees. In: International Conference on Machine Learning (ICML) (2018)
34. Taylor, A.B., Hendrickx, J.M., Glineur, F.: Exact worst-case performance of first-order methods for composite convex optimization. SIAM J. Optim. **27**(3), 1283–1313 (2017)
35. Taylor, A.B., Hendrickx, J.M., Glineur, F.: Performance estimation toolbox (PESTO): automated worst-case analysis of first-order optimization methods. In: Proceedings of the 56th Conference on Decision and Control (CDC) (2017)
36. Taylor, A.B., Hendrickx, J.M., Glineur, F.: Smooth strongly convex interpolation and exact worst-case performance of first-order methods. Math. Program. **161**(1–2), 307–345 (2017)
37. Taylor, A.B., Hendrickx, J.M., Glineur, F.: Exact worst-case convergence rates of the proximal gradient method for composite convex minimization. J. Optim. Theory Appl. **178**, 455–476 (2018)
38. Upadhyaya, M., Banert, S., Taylor, A.B., Giselsson, P.: Automated tight Lyapunov analysis for first-order methods. arXiv:2302.06713 (2023)

39. Van Scoy, B., Freeman, R.A., Lynch, K.M.: The fastest known globally convergent first-order method for minimizing strongly convex functions. IEEE Control Syst. Lett. **2**(1), 49–54 (2017)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.