Performance Estimation Toolbox (PESTO): User Manual*

Adrien B. Taylor†, Sebastien Colla‡, Julien M. Hendrickx§, François Glineur¶

Current version: September 10, 2021

*This research is supported by the Belgian Network DYSCO (Dynamical Systems, Control, and Optimization), funded by the Interuniversity Attraction Poles Programme, initiated by the Belgian State, Science Policy Office, and of the Concerted Research Action (ARC) programme supported by the Federation Wallonia-Brussels (contract ARC 14/19-060). The scientific responsibility rests with its authors.

†INRIA, SIERRA project-team, and D.I. Ecole normale supérieure, Paris, France. Email: adrien.taylor@inria.fr
‡UCLouvain, ICTEAM Institute, Louvain-la-Neuve, Belgium. Email: sebastien.colla@uclouvain.be
§UCLouvain, ICTEAM Institute, Louvain-la-Neuve, Belgium. Email: julien.hendrickx@uclouvain.be
¶UCLouvain, ICTEAM Institute/CORE, Louvain-la-Neuve, Belgium. Email: francois.glineur@uclouvain.be

Foreword

This toolbox was written with as only objective to ease the access to the performance estimation framework for performing automated worst-case analyses. The main underlying idea is to allow the user writing the algorithms nearly as he would have implemented them, instead of performing the potentially demanding SDP modelling steps required for using the methodology.

Contributions and feedbacks

In case the toolbox and/or the methodology raises some interest to you, and that you would like to provide/suggest new functionalities or improvements, we would be very happy to hear from you (also if you find any kind of typo or error). In particular, if you want to provide a PESTO example involving new methods, we would be very happy to include them within the example section along with appropriate acknowledgements.

Acknowledgements

The authors would like to thank François Gonze (UCLouvain), Yoel Drori (Google Inc.) and Théo Golvet (ENSTA ParisTech) for their very constructive feedbacks on preliminary versions of the toolbox. Additional material was incorporated thanks to:

- Ernest Ryu (UCLA), Carolina Bergeling (Lund), and Pontus Giselsson (Lund) [monotone operators and splitting methods],
- Francis Bach (Inria & ENS Paris) [stochastic methods, potential functions, and inexact proximal operations],
- Radu-Alexandru Dragomir (ENS Paris & TSE), Alexandre d'Aspremont (CNRS & ENS Paris), and Jérôme Bolte (TSE) [Bregman divergences and new notions of smoothness].
- Mathieu Barré (Inria & ENS Paris), and Alexandre d'Aspremont (CNRS & ENS Paris) [adaptive methods, and inexact proximal operations].

Last but not least, we thank Loic Estève (Inria) for technical support.

Recent updates

09/2021 **[New]** New auxiliary functions and methods tailored for simplifying the writing of decentralized gradient-type methods within `PESTO`. Those new functions allow considering all agents at once in the operations, instead of having to implement multiple loops for modeling the network. See Examples/11/A.

09/2021 **[New]** Creation of the Matrix class for representing consensus matrices within a PEP. The goal is to simplify the writing of decentralized gradient-type methods.

05/2021 **[Update]** Default verbose setting is medium ("2"): `PESTO` in verbose mode, but solver not.

05/2021 **[Update]** Update in the `PerformanceMetric` command, which allows replacing previous performance measure (instead of taking the min, which it still does by default, see example in Section 3).

05/2021 **[Update]** We can add names to additional (custom) constraints, which now appear in the corresponding list of dual variables (along with their names).

05/2021 **[New]** Possibility of using additional linear matrix inequalities (LMIs) using the new command `AddLMIConstraint`.

05/2021 **[New]** Decentralized Gradient Descent (DGD) added to the examples.

05/2020 **[New]** New primitives and examples: inexact proximal operations.

02/2020 **[New]** Example section extended with two adaptive methods (variants of Polyak steps).

11/2019 **[New]** The example section was largely extended and now contain about **50 examples**. Newcomers include: stochastic methods, non-Euclidean methods (e.g., NoLips), nonconvex applications and monotone inclusions. We also show how to use the toolbox to verify potential functions.

11/2019 **[New]** New primitives: mirror and proximal mirror steps.

11/2019 **[Update]** Updated operations on functions: additions, subtractions, and scalings are implemented in a more efficient fashion.

11/2019 **[Update]** Behaviors of tags was slightly modified (see late Section 4.5), allowing more straightforward implementations of algorithms such as Bregman gradient (a.k.a., NoLips) methods.

11/2019 **[Correction]**: Proximal operations for sums of functions were not effectively implemented.

12/2018 **[New]** Maximally monotone, Lipschitz, cocoercive and strongly monotone operators were added (see Section 3.9). Note that `PESTO` was initially thought for handling performance of optimization algorithms. Internally, the toolbox handles operators exactly as it does with function, except that no function values are available for operators.

12/2018 **[New]** Example section completed with much more material. In particular, we added applications to operator splitting methods and fixed-point iterations.

Contributors

- Adrien Taylor (personal page)
- Julien Hendrickx (personal page)
- François Glineur (personal page)
- Sébastien Colla (personal page)

1 Introduction

This note details the working procedure of the performance estimation toolbox, whose aim is to ease and improve the performance analyses of first-order optimization methods. The methodology originates from the seminal work on performance estimation of Drori and Teboulle [1] (see also [2] for a full picture of the original developments), and on the subsequent convex interpolation framework developed by the authors [3, 4] for obtaining non-improvable guarantees for families of first-order methods and problem classes. The interested readers can find a complete survey on the performance estimation literature in the recent [5].

The performance estimation toolbox relies on the use of the YALMIP [6] modelling language within MATLAB, and on the use of an appropriate semidefinite programming (SDP) solver (see for example [7, 8, 9]). Note that the toolbox is not intended to provide the most efficient implementation of the performance estimation methodology, but rather to provide a simple, generic and direct way to use it. In addition, it is important to have in mind that our capability to (accurately) solve PEPs is inherently limited to our capability to solve semidefinite programs. Typically, the methodology is well-suited for studying a few iterations of simple optimization schemes, but its computational cost may become prohibitive in the case of a large number of iterations (see examples below).

1. Please reference PESTO when used in a published work:

   - A. Taylor, J. Hendrickx, and F. Glineur. Performance Estimation Toolbox (PESTO): automated worst-case analysis of first-order optimization methods. In *Proceedings of the 56th IEEE Conference on Decision and Control (CDC 2017)*, 2017

   Note that the general methodology used in PESTO is presented in the following works:

   - A. B. Taylor, J. M. Hendrickx, and F. Glineur. Smooth strongly convex interpolation and exact worst-case performance of first-order methods. *Mathematical Programming*, 161(1-2):307–345, 2017
   - A. B. Taylor, J. M. Hendrickx, and F. Glineur. Exact worst-case performance of first-order methods for composite convex optimization. *SIAM Journal on Optimization*, 27(3):1283–1313, 2017

2. We distribute PESTO for helping researchers of the field, but we do not provide any warranty on the provided results. In particular, note that:

   - our capability to solve performance estimation problems is limited by our capability to solve semidefinite programs. Therefore, the solver choice is of utmost importance, as well as an appropriate treatment of the errors/numerical problems within the solver. Good practices regarding the use of the toolbox are presented in Section 3.
   - The toolbox is not aimed to provide computationally efficient implementations of the PEPs. It is foremost designed for (1) obtaining preliminary results on the worst-case performance of simple optimization schemes, (2) helping researchers obtaining worst-case guarantees on their algorithms, and (3) providing a simple numerical validation tool for assessing the quality of other analytical or numerical worst-case guarantees.

Depending on the final goal, the advanced users may prefer develop their own (optimized) codes for studying specific algorithms.

Related methodology Semidefinite programming was also used in a related approach [11] for obtaining bounds on the worst-case guarantees. This alternative approach is specialized for obtaining asymptotic linear rates of convergence and relies on control theory via the so-called *integral quadratic constraints* (IQC) framework. The relation between performance estimation and integral quadratic constraints for studying performances of first-order optimization schemes was recently showed in [12]. In a few words, [12] formulates performance estimation problems for looking towards *Lyapunov functions* (i.e., compact proofs of linear convergence).

2 Setting up the toolbox

Pre-requisites In order to install the package, please make sure that both YALMIP (Version 19-Sep-2015 or later) and some SDP solver (e.g., SeDuMi [7], MOSEK [8], or SDPT3 [9]) are installed and properly working on your computer. For testing the proper installation of YALMIP and a SDP solver, you may run the following command

```
>> yalmiptest
```

Downloading the code The toolbox is fully available from the following GITHUB repository:

ADRIENTAYLOR/PERFORMANCE-ESTIMATION-TOOLBOX.

Install PESTO

```
>> Install_PESTO
```

First aid within PESTO

```
>> help pesto
```

Further support can be obtained by contacting the authors.

The best way to quickly get used to the framework is by probably by using the different demonstration files that are available within PESTO. Available demos are summarized by typing:

```
>> demo
```

3 Basic use of the toolbox

For the complete pictures and details on the approach, we refer to [3, Section 1&3] for the simplified case for smooth strongly convex unconstrained minimization, and to [4, Section 1&2] for the full approach for taking into account non-smooth, constrained, composite or finite sums terms in the objective function, with first-order methods possibly involving projection, linear-optimization, proximal or inexact operations. For the sake of simplicity, we approach the toolbox via an example, allowing to go through the different important elements to consider.

Let us consider the following non-smooth convex minimization problem

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

with f being a closed, convex and proper function with bounded subgradients, i.e., for all $g \in \partial f(x)$ for some $x \in \mathbb{R}^d$, we have $\|g\| \leq R$ for some constant $R \geq 0$ (for convenience, we denote $f \in \mathcal{C}_R$). In this example, we study the worst-case performance of the projected subgradient method for solving (OPT):

$$x_i = x_{i-1} - h_{i-1} f'(x_{i-1}), \tag{1}$$

where $f'(x_{i-1}) \in \partial f(x_{i-1})$ is a subgradient of f at x_{i-1}, and $h_i \in \mathbb{R}$ is some step size parameter. For the worst-case performance measure, we chose to use the criterion

$$\min_{0\leq i\leq N} f(x_i) - f(x_\star),$$

with $x_\star$ an optimal solution to (OPT), and N being the number of iterations. Finally, in order to have a bound worst-case measure, we need to consider an *initial condition*; we chose to consider that the initial iterate x_0 to satisfy the following quality measure:

$$\|x_0 - x_\star\| \leq 1,$$

with the initial distance being arbitrarily set to 1 (see homogeneity relations [3, Section 3.5]).

3.1 Performance estimation problems

The key idea underlying the performance estimation approach relies in using the definition of the *worst-case behavior*. That is, the worst-case behavior of

$$\begin{aligned} \max_{f,\{x_i\},x_\star} \quad & \min_{0\leq i\leq N} f(x_i) - f(x_\star), \\ \text{s.t.} \quad & f \in \mathcal{C}_R(\mathbb{R}^d) \\ & x_0 \text{ satisfies some initialization conditions: } \|x_0 - x_\star\|^2 \leq 1 \\ & x_i \text{ is computed by (1) for all } 1 \leq i \leq N, \\ & x_\star \text{ is a minimizer of } f(x). \end{aligned} \tag{PEP(d)}$$

For treating (PEP(d)), we use semidefinite programming. All the modelling steps for going from (PEP(d)) to a semidefinite program for more complicated settings are detailed in [3] and [4].

The first step taken in that direction is to use a discrete version of (PEP(d)), replacing the *infinite-dimensional* variable and constraint $f \in \mathcal{C}_R(\mathbb{R}^d)$ by an *interpolation constraint* using only coordinates x_i, subgradients g_i and function values f_i of the iterates and of an optimal point:

$$\exists f \in \mathcal{C}_R : \; g_i \in \partial f(x_i) \text{ and } f_i = f(x_i) \text{ for all } i \in I \overset{\text{(Definition)}}{\Leftrightarrow} \{(x_i, g_i, f_i)\}_{i\in I} \text{ is } \mathcal{C}_R\text{-interpolable},$$

with $I = \{0, 1, \ldots, N, *\}$. One can show that this constraint can equivalently be formulated as[1]

$$\{(x_i, g_i, f_i)\}_{i\in I} \text{ is } \mathcal{C}_R\text{-interpolable} \Leftrightarrow f_i \geq f_j + \langle g_j, x_i - x_j\rangle \text{ for all } i, j \in I, \text{ and } \|g_i\|^2 \leq R^2 \text{ for all } i \in I.$$

[1] Interpolation conditions for other classes of functions can be found in [4, Section 3].

Therefore, assuming without loss of generality that $x_\star = g_\star = 0$ and that $f_\star = 0$, the problem (PEP(d)) can be reformulated as

$$\begin{aligned} \max_{\{x_i,g_i,f_i\}_{i\in I}} \ & \min_{0\leq i\leq N} f_i, \qquad \text{(discrete-PEP}(d)\text{)} \\ \text{s.t. } & f_i \geq f_j + \langle g_j, x_i - x_j\rangle \text{ for all } i,j \in I, \text{ and } \|g_i\|^2 \leq R^2 \text{ for all } i \in I, \\ & x_i, g_i \in \mathbb{R}^d \text{ for all } i \in I, \\ & x_0 \text{ satisfies some initialization conditions: } \|x_0 - x_\star\|^2 \leq 1, \\ & x_i = x_{i-1} - h_{i-1}g_{i-1} \text{ for all } 0 \leq i \leq N-1,, \\ & g_\star = 0. \end{aligned}$$

For solving (discrete-PEP(d)), we introduce the following notations:

$$P = [g_0 \ g_1 \ \dots \ g_N \ x_0],$$

along with $\mathbf{g}_i = e_{1+i}$ (for $i = 0,\dots,N$), $\mathbf{x}_0 = e_{N+2}$, $\mathbf{x}_i = \mathbf{x}_{i-1} - h_{i-1}\mathbf{g}_{i-1}$ (for $i = 1,\dots,N$) and $\mathbf{g}_\star = \mathbf{x}_\star = 0$. Those notations allow conveniently writing for all $i \in \{0,1,\dots,N,*\}$

$$\begin{aligned} x_i &= P\mathbf{x}_i, \\ g_i &= P\mathbf{g}_i. \end{aligned}$$

Using the previous notations, we note that all scalars products and norms present in the formulation (discrete-PEP(d)) can be written by combining entries of the matrix $G = P^\top P$ which is positive semidefinite by construction (notation $G \succeq 0$). Indeed, we have the following equalities:

$$\langle g_j, x_i - x_j\rangle = \mathbf{g}_i^\top G(\mathbf{x}_i - \mathbf{x}_j), \quad \|x_0 - x_\star\|^2 = \mathbf{x}_0^\top G\mathbf{x}_0, \text{and } \|g_i\|^2 = \mathbf{g}_i^\top G\mathbf{g}_i.$$

In addition, we have the following equivalence:

$$G \succeq 0, \ \text{rank } G \leq d \Leftrightarrow G = P^\top P \text{ with } P \in \mathbb{R}^{d\times(N+2)},$$

which allows writing (discrete-PEP(d)) as a rank-constrained semidefinite program (SDP).

$$\begin{aligned} \max_{G\succeq 0,\{f_i\}_{i=0,\dots,N},\tau} \ & \tau, \qquad \text{(SDP-PEP}(d)\text{)} \\ \text{s.t. } & f_i \geq f_j + \langle g_j, x_i - x_j\rangle \text{ for all } i,j \in I, \text{ and } \|g_i\|^2 \leq R^2 \text{ for all } i \in I, \\ & \tau \leq f_i \text{ for all } i \in I, \\ & \|x_0 - x_\star\|^2 \leq 1, \\ & \text{rank}\, G \leq d. \end{aligned}$$

For obtaining a formulation that is both *tractable* and *valid for all dimensions*, one can relax the rank constraint from (SDP-PEP(d)), and solve the corresponding simplified SDP. That is, instead of considering solving (SDP-PEP(d)) for all values of d, we solve (SDP-PEP(d)) only for $d = N+2$ (see [4, Remark 3]). The worst-case guarantee obtained by solving (PEP($N+2$)) is valid for any value of the dimension parameter d, and is guaranteed to be *exact* (i.e., or non-improvable) as long as $d \geq N+2$ (the so-called *large-scale* assumption). In addition, it can be solved using standard SDP solvers such as [7, 8, 9].

Good practice *rank deflection constraints*

Due to current techniques for solving semidefinite programs, the presence of constraints enforcing *rank-deficiency* of the Gram matrix may critically deteriorate the quality of the numerical solutions. For avoiding that, the user should evaluate as few function values and gradients as possible (for limiting the size of the Gram matrix), avoid replicates (avoid evaluating two times the same gradient at the same point), and generally avoid constraints enforcing linear dependence between two vectors. Common examples include

- (algorithmic constraints imposing rank-deficiency) a constraint $\|x_1 - x_0 + f'(x_0)\|^2 = 0$ enforces the equality $x_1 = x_0 - f'(x_0)$. You should instead consider substituting x_1 by $x_0 - f'(x_0)$; this is done automatically by the toolbox by defining x_1 as $x_0 - f'(x_0)$ (see example from Section 3.2).
- (interpolation constraints imposing rank-deficiency) When performing several subgradient evaluations at the same point, the corresponding subgradients may in general be different (if the subdifferential is not a singleton). However, if you evaluate several times the gradient of a differentiable function at the same point, smoothness *implicitly* imposes a rank deficiency on the Gram matrix, as it is equivalent to $\|g_1 - g_2\|^2 = 0$, where $g_1, g_2 \in \partial f(x)$.
- (inexactness model imposing rank-deficiency — see examples and [13]) Consider an initial iterate x_0 and a search direction given by a vector d_0 satisfying a relative accuracy criterion: $\|d_0 - f'(x_0)\| \leq \varepsilon \|f'(x_0)\|$. In the case $\varepsilon = 0$, the model imposes $\|d_0 - f'(x_0)\| \leq 0$ and hence $d_0 = f'(x_0)$. It is far better to consider substituting d_0 by $f'(x_0)$ instead of using the constraint $\|d_0 - f'(x_0)\|^2 = 0$.

3.2 Setting up the PEP within PESTO: full example

In this section, we exemplify the approach for studying N steps of a subgradient method for minimizing a convex function with bounded subgradients. We chose to use the constant step size rule $h_i = \frac{1}{\sqrt{N+1}}$ and arbitrarily consider the class $\mathcal{C}_R$ with $R = 1$. The example is detailed in the following sections.

```
% (0) Initialize an empty PEP
P=pep();

% (1) Set up the objective function
param.R=1;  % 'radius'-type constraint on the subgradient norms: ||g||≤1

% F is the objective function
F=P.DeclareFunction('ConvexBoundedGradient',param);

% (2) Set up the starting point and initial condition
x0=P.StartingPoint();            % x0 is some starting point
[xs,fs]=F.OptimalPoint();        % xs is an optimal point, and fs=F(xs)
P.InitialCondition((x0-xs)^2≤1); % Add an initial condition ||x0-xs||^2≤ 1

% (3) Algorithm and (4) performance measure
N=5; % number of iterations
h=ones(N,1)*1/sqrt(N+1); % step sizes

x=x0;

% Note: the worst-case performance measure used in the PEP is the
%       min_i (PerformanceMetric_i) (i.e., the best value among all
%       performance metrics added into the problem. Here, we use it
%       in order to find the worst-case value for min_i [F(x_i)-F(xs)]

% we create an array to save all function values (so that we can evaluate
% them afterwards)
f_saved=cell(N+1,1);
for i=1:N
    [g,f]=F.oracle(x);
    f_saved{i}=f;
    P.PerformanceMetric(f-fs);
    x=x-h(i)*g;
end

[g,f]=F.oracle(x);
f_saved{N+1}=f;
P.PerformanceMetric(f-fs);

% (5) Solve the PEP
P.solve();

% (6) Evaluate the output
for i=1:N+1
    f_saved{i}=double(f_saved{i});
end
f_saved
% The result should be (and is) 1/sqrt(N+1).
```

3.3 Basic objects and algebraic operations

There are four essential types of objects implemented within the toolbox:

1. functions, for which we refer to 3.4. Functions can be created, added, subtracted, scaled, and evaluated. There are two basic ways to create functions: firstly, by relying on the `DeclareFunction` method of a PEP object (see Section 3.2, step (0) Initialization of a PEP), and secondly by performing basic operations with other functions (sums, differences scaling, etc.). In the following example, we create and add two convex functions (more functional classes are described in the sequel).

```
% We declare two convex functions: f1 and f2.
f1=P.DeclareFunction('Convex');
f2=P.DeclareFunction('Convex');

% We create a new function F that is the sum of f1 and f2.
F=f1+f2;
```

   Let `x0` be some initial point. In order to evaluate the function, there are three standard ways. First, if only the subgradient of F at x_0 is of interest, one can use the following.

```
% Evaluating a subgradient of F at x0.
g0=F.subgradient(x0);
```

   If only the function value $F(x_0)$ is of interest, one can use the following alternative.

```
% Evaluating  F(x0).
F0=F.value(x0);
```

   Finally, if both a subgradient and a function value are of interest, we advise the user to use the following construction (which is better than combining the previous ones) performing both evaluations simultaneously.

```
% Evaluating  F(x0) and a subgradient of F at x0.
[g0,F0]=F.oracle(x0);
```

2. Vectors, which can be created, added, subtracted or multiplied (inner product) with each other or with a constant (also, divisions by nonzero constants are accepted). Once the PEP object is solved, vectors can also be evaluated. The basic operations for creating a vector are the following

   - by evaluating a subgradient of a function (see previous point),
   - by generating a *starting point*, that is, generating a point with no constraint (yet) on its position. This operation can be repeated to generate as much starting points as needed.

     ```
     x0=P.StartingPoint(); % x0 is some starting point
     ```

   - Also, it is possible to generate an optimal point of a given function.

     ```
     [xs,fs]=F.OptimalPoint(); % xs is an optimal point, and fs=F(xs)
     ```

   - Finally, it is possible to define new vectors by combining other ones. For example, for describing the iterations of an algorithm.

     ```
     x=x-F.subgradient(x); % subgradient step (step size 1)
     ```

     Note an alternate form for the previous code is as follows

     ```
     x=gradient_step(x,F,1); % subgradient step (step size 1)
     ```

   It is also possible to compute inner products of pairs of vectors, resulting in scalar values. This operation is essential for defining (among others) initial conditions, performance measures, and interpolation conditions.

```
% scalar_value1 is squared distance between x and the optimal point xs.
scalar_value1=(x-xs)^2;

% scalar_value2 is the inner product of a subgradient of F at x and x
scalar_value2=F.subgradient(x)*x;
```

Finally, once the corresponding PEP has been solved, vectors (and scalars) involved in this PEP can be evaluated using the `double` command. For example, the following evaluations are valid:

```
double(scalar_value1), double(x0-xs), double((x0-xs)^2), double(scalar_value2)
```

3. Scalars (constants, function values f_i's or inner products $\langle g_i, x_i \rangle$'s), which can be added, subtracted with each others (also, divisions by nonzero constants are accepted). Scalars can also be used to generate constraints (see next point). Once the PEP object is solved, scalars can also be evaluated.

4. Constraints (see also Section 4.1), which can be created by linearly combining scalar values in an (in)equality. In addition to the interpolation constraints, the two most common examples involve initial conditions and performance measures. The following code is valid and add two *initialization* constraints to the PEP:

```
P.InitialCondition((x0-xs)^2<=1);% Add an initial condition ||x0-xs||^2<= 1
P.InitialCondition(F0-Fs<=1);    % Add an initial condition F0-Fs<= 1
```

In PESTO, the performance measure is assumed to be of the form $\min_k\{m_k(G,\{f_i\}_i)\}$, where each $m_k(.)$ is a performance measure (e.g., in the previous subgradient example, we used $\min_{0\leq k\leq N} f_k - f_\star$). The command `PerformanceMetric` allows to create new $m_k(.)$'s.

```
P.PerformanceMetric((x-xs)^2);       % Add a performance measure ||x-xs||^2
P.PerformanceMetric(F.value(x)-Fs); % Add a performance measure F(x)-Fs
```

Note that as in the example (SDP-PEP(d)), the performance metrics are encoded as new constraints involving the objective function τ. The default behavior of `PerformanceMetric` can also be changed to replace the previous objectives, as follows.

```
P.PerformanceMetric((x-xs)^2);  % Add a performance measure ||x-xs||^2
P.PerformanceMetric((F.gradient(x))^2,'min'); % Add a performance measure ...
    ||F'(x)||^2, same behavior as P.PerformanceMetric((F.gradient(x))^2)
% At this point, the performance measure is min{ ||x-xs||^2, ||F'(x)||^2}.
% Let's change that to F(x)-Fs.
P.PerformanceMetric(F.value(x)-Fs,'replace');   % Replace all previously ...
    defined performance measures by F(x)-Fs
```

3.4 Functional classes

In PESTO, interpolation constraints are hidden to the users, and are specifically handled by routines in the `Functions_classes` directory. The list of functional classes for which interpolation constraints are handled in the toolbox is presented in Table 1. For details about the corresponding input parameters, type `help ClassName` in the Matlab prompt (e.g., `help Convex`).

Function class	PESTO routine name	Tightness
Convex functions	`Convex`	✔
Convex functions (bounded subdifferentials)	`ConvexBoundedGradient`	✔
Convex indicator functions (bounded domain)	`ConvexIndicator`	✔
Convex support functions (bounded subdifferentials)	`ConvexSupport`	✔
Smooth strongly convex functions	`SmoothStronglyConvex`	✔
Smooth (possibly nonconvex) functions	`Smooth`	✔
Smooth convex functions (bounded subdifferentials)	`SmoothConvexBoundedGradient`	✔
Strongly convex functions (bounded domain)	`StronglyConvexBoundedDomain`	✔

Table 1: Default functional classes within PESTO. Some classes are overlapping and are present only for promoting a better readability of the code. The corresponding interpolation conditions are developed in [4, Section 3.1].

Good to know *Interpolation and hidden assumptions*

The functions are only required to be interpolated at the points they were evaluated. This conception is of utmost importance when performing PEP-based worst-case analyses, as this may incorporate *hidden assumptions*. Common examples include:

- (existence of optimal point) not evaluating the function at an optimal point is equivalent not to assume the existence of an optimal point. Hence, the worst-case guarantees will be valid even when no optimal point exists.
- (feasible initial point) In the case of constrained minimization, not evaluating the corresponding indicator function at an initial point is equivalent not to assume that this point is feasible (i.e., we do not require the existence of a subgradient of the indicator function at that point). Hence, the worst-case guarantees will be valid even for infeasible initial points.

Note that those remark are also generically valid when performing convergence proofs. As PEPs can be seen as black-boxes proof generator, it is of utmost importance to be aware of the assumptions being made.

As an illustration of the previous remark, the following codes can be used to study the worst-case performances of the projected gradient method for minimizing a (constrained) smooth strongly convex function. In the first case, we require x_0 to be feasible, by evaluating the indicator function at x_0 (i.e., we require the indicator function to have a subgradient at x_0, and hence force x_0 to be feasible).

Example 1 In this example, x_0 is feasible.

```
% In this example, we use a projected gradient method for
% solving the constrained smooth strongly convex minimization problem
%   min_x F(x)=f_1(x)+f_2(x);
%   for notational convenience we denote xs=argmin_x F(x);
% where f_1(x) is L-smooth and mu-strongly convex and where f_2(x) is
% a convex indicator function.
%
% We show how to compute the worst-case value of F(xN)-F(xs) when xN is
% obtained by doing N steps of the method starting with an initial
% iterate satisfying F(x0)-F(xs)≤1.

% (0) Initialize an empty PEP
P=pep();

% (1) Set up the objective function
paramf1.mu=.1;  % Strong convexity parameter
paramf1.L=1;    % Smoothness parameter
f1=P.DeclareFunction('SmoothStronglyConvex',paramf1);
f2=P.DeclareFunction('ConvexIndicator');
F=f1+f2; % F is the objective function

% (2) Set up the starting point and initial condition
x0=P.StartingPoint();        % x0 is some starting point
[xs,fs]=F.OptimalPoint();    % xs is an optimal point, and fs=F(xs)
[g0,f0]=F.oracle(x0);

P.InitialCondition(f0-fs≤1); % Add an initial condition f0-fs≤1

% (3) Algorithm
gam=1/paramf1.L;        % step size
N=1;        % number of iterations

x=x0;
for i=1:N
    xint=gradient_step(x,f1,gam);
    x=projection_step(xint,f2);
end
xN=x;
fN=F.value(xN);

% (4) Set up the performance measure
P.PerformanceMetric(fN-fs);

% (5) Solve the PEP
P.solve()

% (6) Evaluate the output
double(fN-fs)   % worst-case objective function accuracy

% Result should be (and is) max((1-paramf1.mu*gam)^2,(1-paramf1.L*gam)^2)
```

Example 2 In this example, x_0 is not required to be feasible.

```
% In this example, we use a projected gradient method for
% solving the constrained smooth strongly convex minimization problem
%   min_x F(x)=f_1(x)+f_2(x);
%   for notational convenience we denote xs=argmin_x F(x);
% where f_1(x) is L-smooth and mu-strongly convex and where f_2(x) is
% a convex indicator function.
%
% We show how to compute the worst-case value of F(xN)-F(xs) when xN is
% obtained by doing N steps of the method starting with an initial
% iterate satisfying ||x0-xs||^2≤1.

% (0) Initialize an empty PEP
P=pep();

% (1) Set up the objective function
paramf1.mu=.1;  % Strong convexity parameter
paramf1.L=1;    % Smoothness parameter
f1=P.DeclareFunction('SmoothStronglyConvex',paramf1);
f2=P.DeclareFunction('ConvexIndicator');
F=f1+f2; % F is the objective function

% (2) Set up the starting point and initial condition
x0=P.StartingPoint();        % x0 is some starting point
[xs,fs]=F.OptimalPoint();    % xs is an optimal point, and fs=F(xs)

P.InitialCondition((x0-xs)^2≤1); % Add an initial condition ||x0-xs||^2≤ 1

% (3) Algorithm
gam=1/paramf1.L;        % step size
N=1;        % number of iterations

x=x0;
for i=1:N
    xint=gradient_step(x,f1,gam);
    x=projection_step(xint,f2);
end
xN=x;
fN=F.value(xN);

% (4) Set up the performance measure
P.PerformanceMetric(fN-fs);

% (5) Solve the PEP
P.solve()

% (6) Evaluate the output
double(fN-fs)   % worst-case objective function accuracy

% Result should be (and is) max((1-paramf1.mu*gam)^2,(1-paramf1.L*gam)^2)
```

3.5 First-order information recovery

As for interpolation conditions, the different models for first-order information recovery (oracles) are hidden to the users, and are specifically handled by routines within the `Primitive_oracles` directory. There are essentially two types of oracles available at the moment, which are summarized in Table 2. For details about the corresponding input parameters, type `help OracleName` in the Matlab prompt (e.g., `help subgradient`).

Type	PESTO routine name	Tightness
Gradient/subgradient	`subgradient`	✔
Inexact gradient/subgradient (relative inaccuracy)	`inexactsubgradient`	✔
Inexact gradient/subgradient (absolute inaccuracy)	`inexactsubgradient`	✔

Table 2: First-order information recovery within PESTO.

3.6 Standard algorithmic steps

In the same philosophy as for functional classes (Section 3.4) and oracles (Section 3.5), the implementation of several standard algorithmic operations are hidden to the users, and are handled by routines within the `Primitive_steps` directory. The list of primitive algorithmic operations is presented in Table 3. For details about the corresponding input parameters, type `help StepName` in the Matlab prompt (e.g., `help gradient_step`).

Algorithmic step	PESTO routine name	Tightness
Gradient/subradient step	`gradient_step`	✔
Projection step	`projection_step`	✔
Proximal step	`proximal_step`	✔
Conditional/Frank-Wolfe/linear optimization step	`linearoptimization_step`	✔
Line search	`exactlinesearch_step`	✖
Inexact proximal step	`inexact_proximal_step`	✔
Mirror/Bregman step	`mirror`	✔
Proximal mirror/Bregman step	`mirror_prox`	✔

Table 3: Standard algorithmic steps within PESTO. Note that no tightness guarantees are provided when using exact line searches (i.e., the code will only provide upper bounds in that case, which are often tight but probably not always [14, 13]).

3.7 Solving the PEP

By default `pep.solve` will display the PEP (SDP) size and the solver output. Verbosity can be controlled using the `verbose_pet` argument:

```
P.solve(0) %for no output
P.solve(1) %for default display
P.solve(2) %for detailed display
```

Without additional specifications, `pep.solve` calls the default Yalmip solver with no output. Changing the solver used and its display level can be done with an extra `solver_opt` argument (see Yalmip's guide for `sdpsetting`):

```
P.solve(1,sdpsettings('solver','SeDuMi')) %to use SeDuMi with default display
P.solve(1,sdpsettings('solver','mosek','mosek.MSK_DPAR_INTPNT_CO_TOL_PFEAS',1e-10)) ...
    %to use MOSEK with extra accuracy
```

3.8 Obtaining low-dimensional worst-case examples

The toolbox is featured with an additional option for trying to enforce the worst-case examples to be *as low-dimensional* as possible. This is done via a *trace heuristic*: using this option, the performance estimation problem is solved *twice*:

1. firstly, for computing the worst-case value.
2. Secondly, by imposing the value of the performance measure to be equal to the worst-case value and by minimizing the trace of the Gram matrix.

This option is activated by doing

```
P.TraceHeuristic(1) % activate the trace heuristic
```

Note again, that this option implies the performance estimation problem to be solved twice.

3.9 Working with operators

In PESTO, operators are treated exactly in the same way as functions; the only difference being that trying to evaluate function values with operators will result in `NaN`. The routines handling quadratic constraints and interpolation conditions for operators are provided within the `Operator_classes` directory. The list of operator classes that are handled in the toolbox is presented in Table 4. For details about the corresponding input parameters, type `help ClassName` in the Matlab prompt (e.g., `help StronglyMonotone`).

Operator class	PESTO routine name	Tightness
Monotone (maximally)	`Monotone`	✔
Strongly monotone (maximally)	`StronglyMonotone`	✔
Cocoercive	`Cocoercive`	✔
Lipschitz	`Lipschitz`	✔
Cocoercive and strongly monotone	`CocoerciveStronglyMonotone`	✖
Lipschitz and strongly monotone	`LipschitzStronglyMonotone`	✖

Table 4: Default operator classes within PESTO. Some classes are overlapping and are present only for promoting a better readability of the code. The corresponding interpolation conditions are developed in [15, Section 2]. Also note that no interpolations condition hold for the classes of *cocoercive and strongly monotone* and *Lipschitz and strongly monotone* operators. Using them might provide in non-tight numerical results.

4 Advanced operations

Although the structure of the toolbox and the basic operations (see Section 3) already allow for a certain flexibility for studying a variety of first-order schemes, some *advanced* operations may be used in order to model a larger panel of methods and functional classes.

4.1 Adding add-hoc constraints

In Section 3.3, we introduced the `InitialCondition` procedure for introducing scalar constraints. Another possibility is to use the `AddConstraint` method. There are essentially no differences between the two methods; the only reason for having both is readability. The following two codes are equivalent.

```
P.InitialCondition((x0-xs)^2≤1);% Add an initial condition ||x0-xs||^2≤ 1
```

```
P.AddConstraint((x0-xs)^2≤1);% Add an initial condition ||x0-xs||^2≤ 1
```

It is also possible to add linear matrix inequalities (LMIs) to impose a matrix M to be positive semi-definite $M \succeq 0$, using the routine `AddLMIConstraint`. This routine takes a cell array as input, allowing to define the matrix M element by element.

```
M = cell(10);
...                         % Defining the elements of M
P.AddLMIConstraint(M);  % Add an LMI constraint for M
```

4.2 Adding points to be interpolated

For modelling purposes, it can be useful to explicitly create new vectors, and link them via a function, and a coordinate/subgradient relation. More precisely, consider two vectors `x` and `g`, one scalar `f` and the following function F:

```
% We declare one convex function
F=P.DeclareFunction('Convex');
```

In order to force `g` and `f` to be respectively a subgradient and the function value of F at `x`, we use the `AddComponent` routine:

```
F.AddComponent(x,g,f); % g=grad of F at x, f=F(x)
```

Note that in some context (e.g., when implementing new algorithmic steps or first-order oracles), it can be useful to create new vectors or scalars with no constraints on them. This can be done using the `Point` class, as follows.

```
x=Point('Point'); % x is a vector
f=Point('Scalar');% f is a scalar; atlernative form: f=Point('Function value')
```

This is for example used for implementing the projection, proximal and the linear optimization steps of the toolbox. As an example, let us consider performing a proximal (or implicit) step on F from some point x_0, with step size γ:

$$x = x_0 - \gamma \partial F(x).$$

This is implemented in the `proximal_step` routine of PESTO. Let us have a look inside it.

```
function [x] = proximal_step(x0,func,gamma)
% [x] = proximal_step(x0,func,gamma)
%
% This routine performs a proximal step of step size gamma, starting from
% x0, and on function func. That is, it performs:
%       x=x0-gamma*g, where g is a (sub)gradient of func at x.
%       (implicit/proximal scheme).
%
% Input: - starting point x0
%        - function func on which the (sub)gradient will be evaluated
%        - step size gamma of the proximal step
```

```
%
% Output: x=x0-gamma*g, where g is a (sub)gradient of func at x.

g=Point('Point');
x=x0-gamma*g;
f=Point('Function value');
func.AddComponent(x,g,f);
end
```

4.3 Adding new primitive oracles and primitive algorithmic steps

Adding new primitive oracles and algorithmic steps within PESTO is fundamentaly very simple: just add a new routines with appropriate input/output arguments. The lists of existing such routines can be found in Section 3.5 and 3.6, and in the directories `Primitive_steps` and `Primitive_oracles` of the toolbox.

4.4 Adding new functional classes

First of all, we saw in Section 3.4 how to create new functions within some predefined classes. As an example, the following lines create a smooth strongly convex function.

```
paramF.mu=.1;    % Strong convexity parameter
paramF.L=1;     % Smoothness parameter
F=P.DeclareFunction('SmoothStronglyConvex',paramF);
```

Essentially, when creating a function, we instantiate a function object containing a list (initially empty) of points on which its corresponding interpolation conditions should hold. For encoding the interpolation condition, we use a very simple approach: each function object also refers to an *interpolation routine* (all interpolation routines are presented in the directory `Functions_classes` of the PESTO toolbox). In order to create new functions, it is also possible to directly create a instantiate a function object and associate it to a specific interpolation routine.

As an example, the interpolation routine for the class of smooth strongly convex function is `SmoothStronglyConvex.m`, and the previous code for generating a smooth strongly convex function can equivalently be written as

```
paramF.mu=.1;    % Strong convexity parameter
paramF.L=1;     % Smoothness parameter
F=P.AddObjective(@(pt1,pt2)SmoothStronglyConvex(pt1,pt2,paramF.mu,paramF.L));
```

In other words, each *interpolation routine* is a method taking two points (`pt1` and `pt2`) in input, as well as as many parameters as needed (here, μ and L), and providing the interpolation constraint corresponding to those two points in output (we assume that interpolation conditions are always required for all pairs of points). In order to create a new function, one has create a *function handle* depending only on the two points (`pt1` and `pt2`), by fixing the values of the parameters.

Concerning the implementation of interpolation routines, note that both points `pt1` and `pt2` are structures with three fields corresponding to the coordinate vector: `pt1.x`, its corresponding (sub)gradient: `pt1.g` and its function value `pt1.f`. All those elements should be treated as standard vectors or scalars of the PESTO toolbox. For example, `SmoothStronglyConvex.m` contains the following code.

```
function cons=SmoothStronglyConvex(pt1,pt2,mu,L)
assert(mu≥0 & L≥0 & L≥mu,'Constants provided to the functional class are not valid');
if ¬(pt1.x.isEqual(pt2.x) && pt1.g.isEqual(pt2.g) && pt1.f.isEqual(pt2.f))
    if L≠Inf
            cons=((pt1.f-pt2.f+pt1.g*(pt2.x-pt1.x)+...
                1/(2*(1-mu/L))*(1/L*(pt1.g-pt2.g)*(pt1.g-pt2.g).'+...
                mu*(pt1.x-pt2.x)*(pt1.x-pt2.x).'-...
                2*mu/L*(pt1.x-pt2.x)*(pt1.g-pt2.g).'))≤0);
    else
            cons=((pt1.f-pt2.f+pt1.g*(pt2.x-pt1.x)+mu/2*(pt1.x-pt2.x)^2)≤0);
```

```
    end
else
    cons=[];
end
end
```

Another example: `Convex.m` contains the following code.

```
function cons=Convex(pt1,pt2)
if ¬(pt1.x.isEqual(pt2.x) && pt1.g.isEqual(pt2.g) && pt1.f.isEqual(pt2.f))
    cons=((pt1.f-pt2.f+pt1.g*(pt2.x-pt1.x))≤0);
else
    cons=[];
end
end
```

4.5 Tags and evaluations

When evaluating a function value, a (sub)gradient, or both, it is possible to *tag* the corresponding values, in order to be able to easily recover them. As examples,

In some cases, tags allows recovering hidden pieces of information. For example, when minimizing $F(x) = f_1(x) + f_2(x)$ with both f_1 and f_2 being non-smooth convex functions and $x_\star$ being an optimal point of F. How do we efficiently recover two vectors $g_1 \in \partial f_1(x_\star)$ and $g_2 \in \partial f_2(x_\star)$ such that $g_1 + g_2 = 0$?

```
f1=P.DeclareFunction('Convex');
f2=P.DeclareFunction('Convex');
F=f1+f2; % F is the objective function

[xs,fs]=F.OptimalPoint('opt'); % xs is an optimal point, and fs=F(xs)

% note that we tag the point xs as 'opt' to be able to re-evaluate it
% easily (providing the oracle routine with this tag allows to recover
% previously evaluated points).

% the next step evaluates the oracle at the tagged point 'opt' (xs) for
% recovering the values of g1s and g2s; this allows to guarantee that
% g1s+g2s=0;
[g1s,¬]=f1.oracle('opt');
[g2s,¬]=f2.oracle('opt');
```

Note that the `double` command, allowing to evaluate a vector or a scalar after solving the PEP does not allow evaluating gradients and function values that were not saved in a variable, or appropriately tagged. For example, the following lines are valid and equivalent

```
double(g1s), double(g2s),
double(f1.gradient('opt')), double(f2.gradient('opt'))
```

(note that evaluating the corresponding function values also work.

```
double(f1.value('opt')), double(f2.value('opt'))
```

Finally, note that *tags* can also be specified when evaluating function and gradient values with the `oracle`, `subgradient` or `value` routines; the three following lines have the same results.

```
F.oracle(x0,'x0');
F.subgradient(x0,'x0');
F.value(x0,'x0');
```

That is, each of those lines evaluate the function and/or gradient at x_0 and tag the evaluation. The evaluated gradient and function values can be recovered using one of the following way:

```
[g0,F0]=F.oracle('x0');
g0=F.subgradient('x0');
F0=F.value('x0');
```

Good to know *Specifying both a tag a a point*

When calling an evaluation routine (e.g., *oracle*, *gradient*, etc.) for evaluating (sub)gradients and/or function values, with both a point and a tag all routine will prioritize tags! Meaning that if another point with the same tag was already evaluated, the output will correspond to the evaluation of the other point. As an example:

```
[g0,F0]=F.oracle(x0,'x0');
[g1,F1]=F.oracle(x1,'x0');
```

will output $g_1 = g_0$ and $f_1 = f_0$ even if $x_1 \neq x_0$. In other words, it will first check in the database if the point corresponding to the tag was already evaluated before performing an actual new evaluation.

5 Applications

More examples and demonstration files are available within the toolbox (in the directories `Examples` and `Examples_CDC`). We provide the overview below; note that we included analyses done through PEPs from different authors (details in the example files) [1, 16, 2, 17, 14, 18, 19, 20, 21, 13, 4, 3, 22, 23, 15, 24, 25], and IQCs [26, 27]

5.1 Complete list of examples

The toolbox include the following examples:

1. methods for unconstrained convex minimization:
   - proximal point algorithm (see e.g., [28]), fast/accelerated proximal point algorithms (see [29]), subgradient methods (see e.g., [30, 31]), subgradient with exact line search (see [14]), gradient method (see e.g., [31, 32]), gradient with exact line-search (see e.g., [13, 33]), heavy-ball method (see e.g., [34]), fast gradient method (see e.g., [35]), conjugate gradient method (see e.g., [14]), optimized gradient method (see e.g., [18, 1, 2, 17]), optimized gradient method with line search (see e.g., [14]), optimized gradient method for gradient norm (see e.g., [19]), fast/accelerated gradient method for smooth strongly convex minimization (see e.g., [31, 36]), triple momentum method (see e.g., [26]), robust momentum method (see e.g., [27]), relatively inexact fast gradient, gradient with exact line-search in inexact directions (see e.g., [13, 33]).
2. Methods for composite convex minimization:
   - proximal point method (see e.g., [28]), projected and proximal gradient method (see e.g., [31, 32]), fast/accelerated projected/proximal gradient method (see e.g., [37, 38], proximal optimized gradient method (see e.g., [4]), conditional gradient/Frank-Wolfe (see e.g., [39, 40]), Douglas-Rachford (see e.g., [41, 42, 43]), fast/accelerated Douglas-Rachford (see e.g., [44]), three operator splitting (see e.g., [45]), Bregman gradient descent/NoLips (see e.g., [46, 47, 48]), Bregman proximal point method (see e.g., [49]), improved interior gradient algorithm (IGA, [50]).
3. Methods for nonconvex minimization
   - gradient descent, NoLips (see e.g., [51]).
4. Methods for stochastic convex minimization:
   - SAGA [52], Point SAGA [53], SAGA with Sampled Negative Momentum (SSNM) [54], stochastic gradient decent (see e.g., [55]).
5. Methods for monotone inclusions (see e.g., [56, 57])
   - proximal point method, fast/accelerated proximal point method [21], Douglas-Rachford [41, 58] (see e.g. [42]), three operator splitting [45].
6. Methods for fixed-point problems (based on [59, 60])
   - Krasnoselskii-Mann [61], Halpern iterations [62].
7. Verifying potential functions: (based on [23])
   - potential on gradient method fast/accelerated gradient method based on that in [36] and [23].
8. Geometry of (possibly smooth and strongly) convex functions
   - Let f and g be two smooth strongly convex functions. What is the maximum distance between the optimum of $f+g$ and the average between the optimum of f and that of g?

9. Adaptive methods:
    - Polyak steps and variants using PEPs in [63].
10. Inexact proximal methods
    - Inexact versions of proximal point, forward-backward splitting, and Douglas-Rachford studied via PEPs in [64].
11. Decentralized optimization methods
    - Decentralized Gradient Descent (DGD, see e.g. [65]), studied via PEPs in [25].

The following pages contain the code for two of them.

5.2 Example 1: Douglas-Rachford splitting for monotone inclusion

```
% In this example, we use a Douglas-Rachford splitting (DRS)
% method for solving a monotone inclusion problem
%   find x st   0 \in Ax + Bx
% where A is L-Lipschitz and monotone and B is (maximally) mu-strongly
% monotone. We denote by JA and JB the respective resolvents of A and B.
%
% One iteration of the algorithm starting from point w is as follows:
%       x = JB( w )
%       y = JA( 2 * x - w )
%       z = w - theta * ( x - y )
% and then we choose as the next iterate the value of z.
%
% Given two initial points w0 and w1, we show how to compute the worst-case
% contraction factor ||z0 - z1||/||w0 - w1|| obtained after doing one
% iteration of DRS from respectively w0 and w1.
% Note that we allow the user to choose a stepsize alpha in the resolvent.
%
% This setting is studied in
% [1] Walaa M. Moursi, and Lieven Vandenberghe. "Douglas-Rachford
%     Splitting for the Sum of a Lipschitz Continuous and a Strongly
%     Monotone Operator." (2019)
% and the methodology using PEPs is presented in
% [2] Ernest K. Ryu, Adrien B. Taylor, C. Bergeling, and P. Giselsson.
%      "Operator Splitting Performance Estimation: Tight contraction
%       factors and optimal parameter selection." (2018).
% since the results of [2] tightened that of [1] we compare with [2] below.

% (0) Initialize an empty PEP
P=pep();

% (1) Set up the class of monotone inclusions
paramA.L  =  1; paramA.mu = 0; % A is 1-Lipschitz and 0-strongly monotone
paramB.mu = .1;                % B is .1-strongly monotone

A = P.DeclareFunction('LipschitzStronglyMonotone',paramA);
B = P.DeclareFunction('StronglyMonotone',paramB);

% (2) Set up the starting points
w0=P.StartingPoint();
w1=P.StartingPoint();
P.InitialCondition((w0-w1)^2≤1);  % Normalize the initial distance ||w0-ws||^2 ≤ 1

% (3) Algorithm
alpha = 1.3;        % step size (in the resolvents)
theta = .9;         % overrelaxation

x0 = proximal_step(w0,B,alpha);
y0 = proximal_step(2*x0-w0,A,alpha);
z0 = w0-theta*(x0-y0);

x1 = proximal_step(w1,B,alpha);
y1 = proximal_step(2*x1-w1,A,alpha);
z1 = w1-theta*(x1-y1);

% (4) Set up the performance measure: ||z0-z1||^2
P.PerformanceMetric((z0-z1)^2);

% (5) Solve the PEP
P.solve()

% (6) Evaluate the output
double((z0-z1)^2)   % worst-case contraction factor
```

5.3 Example 2: properties of smooth strongly convex functions

The file `Examples/08_Properties of (non)convex functions/A_Minimizer of a sum.m` illustrates how the toolbox can be used to explore properties of (strongly) convex functions that are not directly related to optimization algorithms. In this example, we consider the following problem.

Given two functions f and g both μ-strongly convex and L-smooth, how far can the minimizer $x_\star(f+g)$ of the sum $f+g$ be from $\frac{1}{2}(x_\star(f)+x_\star(g))$ (where $x_\star(f)$ and $x_\star(g)$ are respectively the minimizers of f and g). Using the code, one can easily verify that

(i) the solution is invariant if μ and L are multiplied by the same constant, and hence that the solution only depends on the condition number $\kappa = L/\mu$;

(ii) the solution scales linearly with $\|x_\star(g) - x_\star(f)\|$ which we thus assume to be 1.

The code computes the worst-case distance $\left\|x_\star(f+g) - \frac{x_\star(f)+x_\star(g)}{2}\right\|$ for various values of κ and plots its dependence on κ.

```
% This example shows how the PESTO toolbox can also be used to explore
% properties of (strongly) convex functions that are not directly related
% to optimization algorithms:
% we consider the following problem: Given two functions f, g both
% mu-strongly convexe and L-smooth, how far can the minimizer xsfg of (f+g)
% be from the mean of the minimizers xsf and xsg of respectively f and g.
%
% One can verify that the answer to this problem
% (i) is invariant if mu and L are multiplied by a same constant, and hence
% only depends on the condition number kappa = L/mu
% (ii) scales linearly with ||xsf-xsg||, which we will thus assume to be 1
%
% The code computes the worst-case distance for various values of kappa and
% plots its dependence on kappa.


% number of tests and vector of kappa to be tested
n_test = 80;
kappa = 1.025.^(1:n_test);
verbose = 0; % verbose mode of the toolbox


for k = 1:n_test;

    % (0) Information about current test
    disp('-----------------------------------------------------')
    disp(['Test ', num2str(k) , ' of ', num2str(n_test)])
    disp(['condition number = ' , num2str(round(kappa(k)*10^3)/10^3)])
    disp('-----------------------------------------------------')

    % (1) Declaration of the "PEP"
    P=pep();

    % (2) functions parameters (using scale_invariance)
    param.mu=1; % Strong convexity parameter
    param.L=kappa(k);       % Smoothness parameter

    % (3) functions declarations
    f= P.DeclareFunction('SmoothStronglyConvex',param);
    g= P.DeclareFunction('SmoothStronglyConvex',param);
    fg = f+g;

    % (4) decleration of the minimizers
    [xsf,fs] = f.OptimalPoint();
    [xsg,gs] = g.OptimalPoint();
    [xsfg,fgs] = fg.OptimalPoint();
```

```
47
48      % (5) Contraint on the minimizer
49      P.AddConstraint((xsf-xsg)^2≤1);
50      % note: the problem will naturally force (xsf-xsg)^2 = 1
51
52      % (6) Criterion to be maximized
53      P.PerformanceMetric((xsfg-(xsf+xsg)/2 )^2 );
54
55      % (7) Solving the Pep
56      P.solve(verbose);
57
58      % (8) Evaluation and storage the output
59      distance_sq(k) =double((xsfg-(xsf+xsg)/2 )^2);
60
61  end
62
63  % representation of the results
64  distance = sqrt(distance_sq)
65
66  figure
67  plot(kappa,distance_sq)
68  xlabel('condition number');
69  ylabel('||x^s_{fg} - \frac{1}{2}(x^s_f+x^s_g)||^2')
70  title('square distance to average minimizers f, g')
71
72  figure
73  plot(kappa,distance)
74  xlabel('condition number');
75  ylabel('||x^s_{fg} - \frac{1}{2}(x^s_f+x^s_g)||')
76  title('distance to average minimizers f, g')
77
78
79  disp('************************************************************')
80  disp('Computation ended')
81  disp('The result should show the square distance is asymptotically')
82  disp('linear in the condition number')
83  disp('************************************************************')
```

References

[1] Y. Drori and M. Teboulle. Performance of first-order methods for smooth convex minimization: a novel approach. *Mathematical Programming*, 145(1-2):451–482, 2014.

[2] Y. Drori. *Contributions to the Complexity Analysis of Optimization Algorithms*. PhD thesis, Tel-Aviv University, 2014.

[3] A. B. Taylor, J. M. Hendrickx, and F. Glineur. Smooth strongly convex interpolation and exact worst-case performance of first-order methods. *Mathematical Programming*, 161(1-2):307–345, 2017.

[4] A. B. Taylor, J. M. Hendrickx, and F. Glineur. Exact worst-case performance of first-order methods for composite convex optimization. *SIAM Journal on Optimization*, 27(3):1283–1313, 2017.

[5] A. B. Taylor. *Convex Interpolation and Performance Estimation of First-order Methods for Convex Optimization*. PhD thesis, Université catholique de Louvain, 2017.

[6] J. Löfberg. YALMIP : A toolbox for modeling and optimization in MATLAB. In *Proceedings of the CACSD Conference*, 2004.

[7] J. F. Sturm. Using SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones. *Optimization Methods and Software*, 11–12:625–653, 1999.

[8] APS Mosek. The MOSEK optimization software. *Online at http://www.mosek.com*, 54, 2010.

[9] K.-C. Toh, M.J. Todd, and R.H. Tütüncü. Sdpt3—a matlab software package for semidefinite programming, version 1.3. *Optimization methods and software*, 11(1-4):545–581, 1999.

[10] A. Taylor, J. Hendrickx, and F. Glineur. Performance Estimation Toolbox (PESTO): automated worst-case analysis of first-order optimization methods. In *Proceedings of the 56th IEEE Conference on Decision and Control (CDC 2017)*, 2017.

[11] L. Lessard, B. Recht, and A. Packard. Analysis and design of optimization algorithms via integral quadratic constraints. *SIAM Journal on Optimization*, 26(1):57–95, 2016.

[12] A. Taylor, B. Van Scoy, and L. Lessard. Lyapunov functions for first-order methods: Tight automated convergence guarantees. In *International Conference on Machine Learning (ICML)*, 2018.

[13] E. de Klerk, F. Glineur, and A. B. Taylor. On the worst-case complexity of the gradient method with exact line search for smooth strongly convex functions. *Optimization Letters*, 11(7):1185–1199, 2017.

[14] Y. Drori and A.B. Taylor. Efficient first-order methods for convex minimization: a constructive approach. *Mathematical Programming*, 184(1):183–220, 2020.

[15] E.K. Ryu, A.B. Taylor, C. Bergeling, and P. Giselsson. Operator splitting performance estimation: Tight contraction factors and optimal parameter selection. *SIAM Journal on Optimization*, 30(3):2251–2271, 2020.

[16] Y. Drori and M. Teboulle. An optimal variant of Kelley's cutting-plane method. *Mathematical Programming*, 160(1):321–351, 2016.

[17] Y. Drori. The exact information-based complexity of smooth convex minimization. *Journal of Complexity*, 39:1–16, 2017.

[18] D. Kim and J. A. Fessler. Optimized first-order methods for smooth convex minimization. *Mathematical Programming*, 159(1-2):81–107, 2016.

[19] D. Kim and J.A. Fessler. Optimizing the efficiency of first-order methods for decreasing the gradient of smooth convex functions. *Journal of Optimization Theory and Applications*, 188(1):192–219, 2021.

[20] D. Kim and J. A. Fessler. On the convergence analysis of the optimized gradient method. *Journal of optimization theory and applications*, 172(1):187–205, 2017.

[21] D. Kim. Accelerated proximal point method for maximally monotone operators. *Mathematical Programming*, pages 1–31, 2021.

[22] A. B. Taylor, J. M. Hendrickx, and F. Glineur. Exact worst-case convergence rates of the proximal gradient method for composite convex minimization. *Journal of Optimization Theory and Applications*, 178(2):455–476, 2018.

[23] A. Taylor and F. Bach. Stochastic first-order methods: non-asymptotic and computer-aided analyses via potential functions. In *Conference on Learning Theory (COLT)*, 2019.

[24] G. Gu and J. Yang. Optimal nonergodic sublinear convergence rate of proximal point algorithm for maximal monotone inclusion problems. *preprint arXiv:1904.05495*, 2019.

[25] S. Colla and J. M. Hendrickx. Automated worst-case performance analysis of decentralized gradient descent. *preprint arXiv:2103.14396*, 2021.

[26] B. Van Scoy, R. A. Freeman, and K. M. Lynch. The fastest known globally convergent first-order method for minimizing strongly convex functions. *IEEE Control Systems Letters*, 2(1):49–54, 2018.

[27] S. Cyrus, B. Hu, B. Van Scoy, and L. Lessard. A robust accelerated optimization algorithm for strongly convex functions. In *2018 Annual American Control Conference (ACC)*, pages 1376–1381. IEEE, 2018.

[28] O. Güler. On the convergence of the proximal point algorithm for convex minimization. *SIAM Journal on Control and Optimization*, 29(2):403–419, 1991.

[29] O. Güler. New proximal point algorithms for convex minimization. *SIAM Journal on Optimization*, 2(4):649–664, 1992.

[30] N. Z. Shor, Krzysztof C. Kiwiel, and Andrzej Ruszcayński. *Minimization methods for non-differentiable functions*. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[31] Y. Nesterov. *Lectures on Convex Optimization*. Springer Optimization and Its Applications. Springer International Publishing, 2018.

[32] B. T. Polyak. *Introduction to Optimization*. Optimization Software New York, 1987.

[33] E. De Klerk, F. Glineur, and A.B. Taylor. Worst-case convergence analysis of inexact gradient and newton methods through semidefinite programming performance estimation. *SIAM Journal on Optimization*, 30(3):2053–2082, 2020.

[34] B. T. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.

[35] Y. Nesterov. A method of solving a convex programming problem with convergence rate $O(1/k^2)$). *Soviet Mathematics Doklady*, 27:372–376, 1983.

[36] N. Bansal and A. Gupta. Potential-function proofs for gradient methods. *Theory of Computing*, 15(1):1–32, 2019.

[37] Y. Nesterov. Gradient methods for minimizing composite functions. *Mathematical Programming*, 140(1):125–161, 2013.

[38] A. Beck and M. Teboulle. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM journal on imaging sciences*, 2(1):183–202, 2009.

[39] M. Frank and P. Wolfe. An algorithm for quadratic programming. *Naval research logistics quarterly*, 3(1-2):95–110, 1956.

[40] M. Jaggi. Revisiting frank-wolfe: Projection-free sparse convex optimization. In *International Conference on Machine Learning (ICML)*, pages 427–435, 2013.

[41] J. Douglas and H. H. Rachford. On the numerical solution of heat conduction problems in two and three space variables. *Transactions of the American Mathematical Society*, 82:421–439, 1956.

[42] H. H. Bauschke and W. M. Moursi. On the Douglas–Rachford algorithm. *Mathematical Programming*, 164(1-2):263–284, 2017.

[43] P. Giselsson and S. Boyd. Linear convergence and metric selection for douglas-rachford splitting and admm. *IEEE Transactions on Automatic Control*, 62(2):532–544, 2017.

[44] P. Patrinos, L. Stella, and A. Bemporad. Douglas–Rachford splitting: Complexity estimates and accelerated variants. In *53rd IEEE Conference on Decision and Control*, pages 4234–4239. IEEE, 2014.

[45] D. Davis and W. Yin. A three-operator splitting scheme and its optimization applications. *Set-valued and variational analysis*, 25(4):829–858, 2017.

[46] H. H. Bauschke, J. Bolte, and M. Teboulle. A descent lemma beyond Lipschitz gradient continuity: first-order methods revisited and applications. *Mathematics of Operations Research*, 42(2):330–348, 2016.

[47] H. Lu, R. M. Freund, and Y. Nesterov. Relatively smooth convex optimization by first-order methods, and applications. *SIAM Journal on Optimization*, 28(1):333–354, 2018.

[48] R.-A. Dragomir, A.B. Taylor, A. d'Aspremont, and J. Bolte. Optimal complexity and certification of bregman first-order methods. *Mathematical Programming*, pages 1–43, 2021.

[49] J. Eckstein. Nonlinear proximal point algorithms using bregman functions, with applications to convex programming. *Mathematics of Operations Research*, 18(1):202–226, 1993.

[50] A. Auslender and M. Teboulle. Interior gradient and proximal methods for convex and conic optimization. *SIAM Journal on Optimization*, 16(3):697–725, 2006.

[51] J. Bolte, S. Sabach, M. Teboulle, and Y. Vaisbourd. First order methods beyond convexity and Lipschitz gradient continuity with applications to quadratic inverse problems. *SIAM Journal on Optimization*, 28(3):2131–2151, 2018.

[52] A. Defazio, F. Bach, and S. Lacoste-Julien. SAGA: A fast incremental gradient method with support for non-strongly convex composite objectives. In *Advances in Neural Information Processing Systems (NIPS)*, pages 1646–1654, 2014.

[53] A. Defazio. A simple practical accelerated method for finite sums. In *Advances in Neural Information Processing Systems (NIPS)*, pages 676–684, 2016.

[54] K. Zhou, Q. Ding, F. Shang, J. Cheng, D. Li, and Z.-Q. Luo. Direct acceleration of saga using sampled negative momentum. In *Proceedings of Machine Learning Research*, volume 89, pages 1602–1610, 2019.

[55] F. R. Bach and E. Moulines. Non-asymptotic analysis of stochastic approximation algorithms for machine learning. In *Advances in Neural Information Processing Systems (NIPS)*, pages 451–459, 2011.

[56] H. H. Bauschke and P. L. Combettes. *Convex analysis and monotone operator theory in Hilbert spaces*, volume 408. Springer, 2011.

[57] E. K. Ryu and S. Boyd. Primer on monotone operator methods. *Appl. Comput. Math*, 15(1):3–43, 2016.

[58] W.M. Moursi and L. Vandenberghe. Douglas–rachford splitting for the sum of a lipschitz continuous and a strongly monotone operator. *Journal of Optimization Theory and Applications*, 183(1):179–198, 2019.

[59] F. Lieder. On the convergence rate of the Halpern-iteration. *Optimization Letters*, 15(2):405–418, 2021.

[60] F. Lieder. *Projection Based Methods for Conic Linear Programming Optimal First Order Complexities and Norm Constrained Quasi Newton Methods*. PhD thesis, Universitats-und Landesbibliothek der Heinrich-Heine-Universitat Dusseldorf, 2018.

[61] W. R. Mann. Mean value methods in iteration. *Proceedings of the American Mathematical Society*, 4(3):506–510, 1953.

[62] B. Halpern. Fixed points of nonexpanding maps. *Bulletin of the American Mathematical Society*, 73(6):957–961, 1967.

[63] M. Barré, A. Taylor, and A. d'Aspremont. Complexity guarantees for Polyak steps with momentum. In *Conference on Learning Theory (COLT)*, 2020.

[64] M. Barré, A. Taylor, and F. Bach. Principled analyses and design of first-order methods with inexact proximal operators. *preprint arXiv:2006.06041*, 2020.

[65] A. Nedic and A. Ozdaglar. Distributed subgradient methods for multi-agent optimization. *Automatic Control, IEEE Transactions on*, 54:48 – 61, 02 2009.