Performance Estimation Toolbox (PESTO): User Manual*

Adrien B. Taylor†, Julien M. Hendrickx†, François Glineur†

Version: June 21, 2017

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.

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.

*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.

†Université catholique de Louvain, ICTEAM Institute/CORE, {adrien.taylor, julien.hendrickx, francois.glineur}@uclouvain.be

Acknowledgements

The authors would like to thank François Gonze from UCLouvain and Yoel Drori from Google Inc. for their feedbacks on preliminary versions of the toolbox.

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. An appropriate conference paper was submitted very soon. In the meantime, the methodology papers can be used:
   - Adrien B Taylor, Julien M Hendrickx, and François Glineur. Smooth strongly convex interpolation and exact worst-case performance of first-order methods. *Mathematical Programming*, 161(1-2):307–345, 2017
   - Adrien B Taylor, Julien M Hendrickx, and François Glineur. Exact worst-case performance of first-order methods for composite convex optimization. *SIAM Journal on Optimization*, 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 [10] for obtaining bounds on the worst-case guarantees. This alternative approach is specialized for obtaining asymptotic linear rates of 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

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

with x_* 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:

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

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

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

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]

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

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

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

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

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

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

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

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:

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

In addition, we have the following equivalence:

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

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

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

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

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 example from Section 5.3) 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 and evaluated. There are two basic ways to create functions: first, by relying on the `DeclareFunction` method of a PEP object (see Section 3.2, step (0) Initialization of a PEP), and second by summing other functions. 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 $\min_{0\leq k\leq N} f_k - f_*$). 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 F0-Fs
```

Note that as in the example (SDP-PEP(d)), the performance metrics are encoded as new constraints involving the objective function τ.

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
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
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
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`

Table 3: Standard algorithmic steps within PESTO.

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 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.

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

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

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 the 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. As an 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_* being an optimal point of F. How do we efficiently recover two vectors $g_1 \in \partial f_1(x_*)$ and $g_2 \in \partial f_2(x_*)$ 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');
```

5 Further Examples

More examples and demonstration files are available within the toolbox (in the directories `Examples` and `Examples_CDC`). The detailed analyses of the examples provided hereafter (optimized gradient method, steepest descent with exact line searches and Douglas-Rachford splitting) can respectively be found in [1, 11, 12], [13] and [14].

5.1 Optimized gradient method

```
% In this example, we use the optimized gradient method (OGM) for
% solving the L-smooth convex minimization problem
%   min_x F(x); for notational convenience we denote xs=argmin_x F(x).
%
% We show how to compute the worst-case value of F(xN)-F(xs) when xN is
% obtained by doing N steps of the gradient method starting with an initial
% iterate satisfying ||x0-xs||≤1.

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

% (1) Set up the objective function
param.L=1;      % Smoothness parameter

F=P.DeclareFunction('SmoothStronglyConvex',param); % 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/param.L;      % step size
N=5;        % number of iterations

x=cell(N+1,1);%we store all the x's in a cell (for convenience)
x{1}=x0;
y=x0;
theta=1;
for i=1:N
    x{i+1}=gradient_step(y,F,gam);
    theta_prev=theta;
    if i<N
        theta=(1+sqrt(4*theta^2+1))/2;
    else
        theta=(1+sqrt(8*theta^2+1))/2;
    end
    y=x{i+1}+(theta_prev-1)/theta*(x{i+1}-x{i})+theta_prev/theta*(x{i+1}-y);
end

% (4) Set up the performance measure
fN=F.value(y);                % g=grad F(x), f=F(x)
P.PerformanceMetric(fN-fs); % Worst-case evaluated as F(x)-F(xs)

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

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

% The result should be 1/2/theta^2
```

5.2 Exact line searches

```
% In this example, we use a gradient method with exact line search
% for solving the L-smooth mu-strongly convex minimization problem
%   min_x F(x); for notational convenience we denote xs=argmin_x F(x).
%
%   Starting from an iterate x0, the method performs at each iteration
%   an exact line search step in the steepest descent direction
%       gamma=argmin_gamma F(xi-gamma*gi), with gi the gradient of F at xi,
%   and performs the update
%       x{i+1}=xi-gamma*gi.
%
% 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
param.mu=.1;    % Strong convexity parameter
param.L=1;      % Smoothness parameter

F=P.DeclareFunction('SmoothStronglyConvex',param);
% 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
N=2;
x=x0;
for i=1:N
    [g,¬]=F.oracle(x);
    x=exactlinesearch_step(x,F,g);
end

% (4) Set up the performance measure
[g,f]=F.oracle(x);
P.PerformanceMetric(f-fs); % Worst-case evaluated as F(x)-F(xs)

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

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

% The result should be
%((param.L-param.mu)/(param.L+param.mu))^(2*N)
```

5.3 Exact line search in inexact search direction

```
% In this example, we use an inexact gradient method with exact line search
% for solving the L-smooth mu-strongly convex minimization problem
%   min_x F(x); for notational convenience we denote xs=argmin_x F(x).
%
%   Starting from an iterate x0, the method performs at each iteration
%   an exact line search step in a direction d satisfying a relative
%   accuracy criterion
%   (gi is the gradient of F at xi)
%       ||d-gi||≤eps*||gi|| (**)
%   that is, the method evaluates
%       gamma=argmin_gamma F(xi-gamma*d) for d satisfying (**)
%   and performs the update
%       x{i+1}=xi-gamma*d.
%
% 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
param.mu=.1;    % Strong convexity parameter
param.L=1;      % Smoothness parameter

F=P.DeclareFunction('SmoothStronglyConvex',param);
% 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
N=2;
eps=0.1;
x=x0;
for i=1:N
    d=inexactsubgradient(x,F,eps,0);
    x=exactlinesearch_step(x,F,d);
end
fN=F.value(x);
% (4) Set up the performance measure
P.PerformanceMetric(fN-fs); % Worst-case evaluated as ||g||^2

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

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

% The result should be
%((param.L*(1+eps)-param.mu*(1-eps))/(param.L*(1+eps)+param.mu*(1-eps)))^(2*N)
```

5.4 Douglas-Rachford splitting

```
% In this example, we use a Douglas-Rachford splitting (DRS)
% method for solving the composite 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 f_2 is convex.
%
% We show how to compute the worst-case value of ||wN-ws||^2 when wN is
% obtained by doing N steps of DRS starting with an initial iterate w0
% satisfying ||w0-ws||≤1, and ws is some point to which the iterates of
% DRS converge.
%
% Note that the point ws may be defined in the following way:
% let g1s and g2s be subgradients of respectively f_1 and f_2 at xs such
% that g1s+g2s=0 (optimality conditions). Then, ws=xs+lambda*g2s (lambda is
% the step size used in DRS).

% (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('Convex');
F=f1+f2; % F is the objective function

% (2) Set up the starting point and initial condition
w0=P.StartingPoint(); % x0 is some starting point
[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');
lambda=2; ws=xs+lambda*g2s;

% Add an initial condition ||w0-ws||^2≤ 1
P.InitialCondition((w0-ws)^2-1≤0);

% (3) Algorithm
N=5;            % number of iterations
gam=lambda;     % step size

w=w0;
for i=1:N
    x=proximal_step(w,f2,gam);
    y=proximal_step(2*x-w,f1,gam);
    w=y-x+w;
end

% (4) Set up the performance measure
P.PerformanceMetric((w-ws)^2);

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

% (6) Evaluate the output
```

```
63 double((w-ws)^2)   % worst-case distance to fixed point ws
64
65 % The result should be (and is)
66 % max(1/(1+paramf1.mu*gam),gam*paramf1.L/(1+gam*paramf1.L))^(2*N)
67 %
68 % see (Theorem 2): Giselsson, Pontus, and Stephen Boyd. "Linear
69 %                  convergence and metric selection in Douglas-Rachford
70 %                  splitting and ADMM."
71 %                  IEEE Transactions on Automatic Control (2016).
```

References

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

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

[4] Adrien B Taylor, Julien M Hendrickx, and François Glineur. Exact worst-case performance of first-order methods for composite convex optimization. *SIAM Journal on Optimization*, 2017.

[5] Adrien 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] Kim-Chuan Toh, Michael J Todd, and Reha H Tütüncü. Sdpt3—a matlab software package for semidefinite programming, version 1.3. *Optimization methods and software*, 11(1-4):545–581, 1999.

[10] Laurent Lessard, Benjamin Recht, and Andrew Packard. Analysis and design of optimization algorithms via integral quadratic constraints. *SIAM Journal on Optimization*, 26(1):57–95, 2016.

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

[12] Donghwan Kim and Jeffrey A Fessler. On the convergence analysis of the optimized gradient method. *Journal of Optimization Theory and Applications*, pages 1–19, 2015.

[13] Etienne de Klerk, François Glineur, and Adrien B Taylor. On the worst-case complexity of the gradient method with exact line search for smooth strongly convex functions. *Optimization Letters*, 2016.

[14] Pontus Giselsson and Stephen Boyd. Linear convergence and metric selection in douglas-rachford splitting and admm. *IEEE Transactions on Automatic Control*, 2016.