Performance Estimation Toolbox (PESTO): User Manual*

Adrien B. Taylor[†], Julien M. Hendrickx[‡], François Glineur[§]

Current version: December 21, 2018

[†]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: 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.

External 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.

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.

Recent updates

12/2018 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 The example section was completed with much more material. In particular, we added applications to operator splitting methods and fixed-point iterations.

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:

   - Adrien Taylor, Julien Hendrickx, and François 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:

   - 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*, 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

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

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

$$\max_{\{x_i, g_i, f_i\}_{i \in I}} \min_{0 \leq i \leq N} f_i, \qquad \text{(discrete-PEP}(d)\text{)}$$

$$\text{s.t. } f_i \geq f_j + \langle g_j, x_i - x_j \rangle \text{ for all } i, j \in I, \text{ and } \|g_i\|^2 \leq R^2 \text{ for all } i \in I,$$

$$x_i, g_i \in \mathbb{R}^d \text{ for all } i \in I,$$

$$x_0 \text{ satisfies some initialization conditions: } \|x_0 - x_\star\|^2 \leq 1,$$

$$x_i = x_{i-1} - h_{i-1} g_{i-1} \text{ for all } 0 \leq i \leq N - 1,,$$

$$g_\star = 0.$$

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 $\mathbf{g}_\star = \mathbf{x}_\star = 0$. Those notations allow conveniently writing for all $i \in \{0, 1, \ldots, N, *\}$

$$x_i = P\mathbf{x}_i,$$

$$g_i = P\mathbf{g}_i.$$

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:

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

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).

$$\max_{G \succeq 0, \{f_i\}_{i=0,\ldots,N}, \tau} \tau, \qquad \text{(SDP-PEP}(d)\text{)}$$

$$\text{s.t. } f_i \geq f_j + \langle g_j, x_i - x_j \rangle \text{ for all } i, j \in I, \text{ and } \|g_i\|^2 \leq R^2 \text{ for all } i \in I,$$

$$\tau \leq f_i \text{ for all } i \in I,$$

$$\|x_0 - x_\star\|^2 \leq 1,$$

$$\text{rank } G \leq d.$$

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 5.4.2) 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.

```matlab
% (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).

```
1  % We declare two convex functions: f1 and f2.
2  f1=P.DeclareFunction('Convex');
3  f2=P.DeclareFunction('Convex');
4
5  % We create a new function F that is the sum of f1 and f2.
6  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.

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

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

```
1  % Evaluating  F(x0).
2  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.

```
1  % Evaluating  F(x0) and a subgradient of F at x0.
2  [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.

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

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

     ```
     1  [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.

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

     Note an alternate form for the previous code is as follows

     ```
     1  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.

```
1  % scalar_value1 is squared distance between x and the optimal point xs.
2  scalar_value1=(x-xs)^2;
3
4  % scalar_value2 is the inner product of a subgradient of F at x and x
5  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:

```
1  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:

```
1  P.InitialCondition((x0-xs)^2<=1);% Add an initial condition ||x0-xs||^2<= 1
2  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 $\min_{0 \le k \le N} f_k - f_\star$). The command **PerformanceMetric** allows to create new $m_k(.)$'s.

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

```matlab
1  % In this example, we use a projected gradient method for
2  % solving the constrained smooth strongly convex minimization problem
3  %    min_x F(x)=f_1(x)+f_2(x);
4  %    for notational convenience we denote xs=argmin_x F(x);
5  % where f_1(x) is L-smooth and mu-strongly convex and where f_2(x) is
6  % a convex indicator function.
7  %
8  % We show how to compute the worst-case value of F(xN)-F(xs) when xN is
9  % obtained by doing N steps of the method starting with an initial
10 % iterate satisfying F(x0)-F(xs)≤1.
11
12 % (0) Initialize an empty PEP
13 P=pep();
14
15 % (1) Set up the objective function
16 paramf1.mu=.1;  % Strong convexity parameter
17 paramf1.L=1;    % Smoothness parameter
18 f1=P.DeclareFunction('SmoothStronglyConvex',paramf1);
19 f2=P.DeclareFunction('ConvexIndicator');
20 F=f1+f2; % F is the objective function
21
22 % (2) Set up the starting point and initial condition
23 x0=P.StartingPoint();        % x0 is some starting point
24 [xs,fs]=F.OptimalPoint();    % xs is an optimal point, and fs=F(xs)
25 [g0,f0]=F.oracle(x0);
26
27 P.InitialCondition(f0-fs≤1); % Add an initial condition f0-fs≤1
28
29 % (3) Algorithm
30 gam=1/paramf1.L;        % step size
31 N=1;         % number of iterations
32
33 x=x0;
34 for i=1:N
35     xint=gradient_step(x,f1,gam);
36     x=projection_step(xint,f2);
37 end
38 xN=x;
39 fN=F.value(xN);
40
41 % (4) Set up the performance measure
42 P.PerformanceMetric(fN-fs);
43
44 % (5) Solve the PEP
45 P.solve()
46
47 % (6) Evaluate the output
48 double(fN-fs)    % worst-case objective function accuracy
49
50 % 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.

```matlab
% 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`	✖

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 [13, 14]).

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:

```
1  P.solve(0) %for no output
2  P.solve(1) %for default display
3  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`):

```
1 P.solve(1,sdpsettings('solver','SeDuMi')) %to use SeDuMi with default display
2 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

```
1 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 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:

```
1  % We declare one convex function
2  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:

```
1  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.

```
1  x=Point('Point'); % x is a vector
2  f=Point('Scalar');% f is a scalar; alternative 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.

```
1  function [x] = proximal_step(x0,func,gamma)
2  % [x] = proximal_step(x0,func,gamma)
3  %
4  % This routine performs a proximal step of step size gamma, starting from
5  % x0, and on function func. That is, it performs:
6  %       x=x0-gamma*g, where g is a (sub)gradient of func at x.
7  %       (implicit/proximal scheme).
8  %
9  % Input: - starting point x0
10 %        - function func on which the (sub)gradient will be evaluated
11 %        - step size gamma of the proximal step
12 %
13 % Output: x=x0-gamma*g, where g is a (sub)gradient of func at x.
14 %
15 g=Point('Point');
16 x=x0-gamma*g;
17 f=Point('Function value');
```

```
18  func.AddComponent(x,g,f);
19
20  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.

```
1  paramF.mu=.1;    % Strong convexity parameter
2  paramF.L=1;      % Smoothness parameter
3  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

```
1  paramF.mu=.1;    % Strong convexity parameter
2  paramF.L=1;      % Smoothness parameter
3  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.

```
1  function cons=SmoothStronglyConvex(pt1,pt2,mu,L)
2  assert(mu≥0 & L≥0 & L≥mu,'Constants provided to the functional class are not valid');
3  if ¬(pt1.x.isEqual(pt2.x) && pt1.g.isEqual(pt2.g) && pt1.f.isEqual(pt2.f))
4      if L≠Inf
5          cons=((pt1.f-pt2.f+pt1.g*(pt2.x-pt1.x)+...
6              1/(2*(1-mu/L))*(1/L*(pt1.g-pt2.g)*(pt1.g-pt2.g).'+...
7              mu*(pt1.x-pt2.x)*(pt1.x-pt2.x).'-...
8              2*mu/L*(pt1.x-pt2.x)*(pt1.g-pt2.g).'))≤0);
9      else
10         cons=((pt1.f-pt2.f+pt1.g*(pt2.x-pt1.x)+mu/2*(pt1.x-pt2.x)^2)≤0);
11     end
12 else
13     cons=[];
14 end
```

page 17

```
15   end
```

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

```
1   function cons=Convex(pt1,pt2)
2   if ¬(pt1.x.isEqual(pt2.x) && pt1.g.isEqual(pt2.g) && pt1.f.isEqual(pt2.f))
3       cons=((pt1.f-pt2.f+pt1.g*(pt2.x-pt1.x))≤0);
4   else
5       cons=[];
6   end
7
8   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 $g_1 + g_2 = 0$?

```
1  f1=P.DeclareFunction('Convex');
2  f2=P.DeclareFunction('Convex');
3  F=f1+f2; % F is the objective function
4
5  [xs,fs]=F.OptimalPoint('opt'); % xs is an optimal point, and fs=F(xs)
6
7  % note that we tag the point xs as 'opt' to be able to re-evaluate it
8  % easily (providing the oracle routine with this tag allows to recover
9  % previously evaluated points).
10
11 % the next step evaluates the oracle at the tagged point 'opt' (xs) for
12 % recovering the values of g1s and g2s; this allows to guarantee that
13 % g1s+g2s=0;
14 [g1s,¬]=f1.oracle('opt');
15 [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

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

(note that evaluating the corresponding function values also work.

```
1  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.

```
1  F.oracle(x0,'x0');
2  F.subgradient(x0,'x0');
3  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:

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

5 Applications

More examples and demonstration files are available within the toolbox (in the directories `Examples` and `Examples_CDC`). In the following pages, we provide codes for analyzing (or verifying the analyses) the following applications.

1. Convex minimization

   - Proximal point algorithm

2. Nonsmooth convex minimization

   - Subgradient methods

3. Smooth convex minimization:

   - Optimized gradient method for smooth convex minimization [16].
   - Optimized gradient method for gradient norm for smooth convex minimization [17]
   - Conjugate gradient for smooth convex minimization [13].

4. Smooth strongly convex minimization:

   - Steepest descent for smooth strongly convex minimization [14].
   - Steepest descent with inexact search directions for smooth strongly convex minimization [14].
   - Triple momentum method for smooth strongly convex minimization [18].
   - Robust momentum method for smooth strongly convex minimization [19].

5. Composite convex minimization and operator splitting

   - Douglas-Rachford splitting for solving a monotone inclusion problem (results to be compared with [15]).
   - Douglas-Rachford splitting for minimizing the sum of two functions (results to be compared with [20]).
   - Davis-Yin operator three-operator splitting (PEP version in [15]).

6. Fixed-point iterations for non-expansive mappings

   - Halpern-iteration for non-expansive mappings (PEP version in [22]).
   - Krasnolskii-Mann (PEP version in [23]).

7. Geometry of smooth 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 ?

5.1   Convex minimization

5.1.1   Proximal point algorithm

This example is due to [4]. The interested reader can easily play with the code (see the `Examples`
directory of the toolbox) for incorporating additional elements such as momentum or inexactness.

```matlab
% In this example, we use a proximal point method for solving the
% non-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 a proximal method starting with an initial
% iterate satisfying ||x0-xs||<=1.

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

% (1) Set up the objective function
F=P.DeclareFunction('Convex'); % 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
N=4;          % number of iterations
gam=1;      % step size

x=x0;
for i=1:N
    x=proximal_step(x,F,gam);
end
xN=x;

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

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

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

% The result should be (and is) 1/(4*N*gam)
% see Taylor, Adrien B., Julien M. Hendrickx, and Francois Glineur.
%     "Exact Worst-case Performance of First-order Methods for Composite
%     Convex Optimization." to appear in SIAM Journal on Optimization
%     (2017)
```

page 21

5.2 Nonsmooth convex minimization

5.2.1 Subgradient methods

```matlab
1  % In this example, we use a subgradient method for
2  % solving the non-smooth convex minimization problem
3  %   min_x F(x); for notational convenience we denote xs=argmin_x F(x);
4  % where F(x) satisfies a Lipschitz condition; i.e., it has a bounded
5  % gradient ||g||<=R for all g being a subgradient of F at some point.
6  %
7  % We show how to compute the worst-case value of min_i F(xi)-F(xs) when xi is
8  % obtained by doing i steps of a subgradient method starting with an initial
9  % iterate satisfying ||x0-xs||<=1.
10
11 % (0) Initialize an empty PEP
12 P=pep();
13
14 % (1) Set up the objective function
15 param.R=1;  % 'radius'-type constraint on the subgradient norms: ||g||<=1
16
17 % F is the objective function
18 F=P.DeclareFunction('ConvexBoundedGradient',param);
19
20 % (2) Set up the starting point and initial condition
21 x0=P.StartingPoint();         % x0 is some starting point
22 [xs,fs]=F.OptimalPoint();     % xs is an optimal point, and fs=F(xs)
23 P.InitialCondition((x0-xs)^2<=1);% Add an initial condition ||x0-xs||^2<= 1
24
25 % (3) Algorithm and (4) performance measure
26 N=5; % number of iterations
27 h=ones(N,1)*1/sqrt(N+1); % step sizes
28
29 x=x0;
30
31 % Note: the worst-case performance measure used in the PEP is the
32 %       min_i (PerformanceMetric_i) (i.e., the best value among all
33 %       performance metrics added into the problem. Here, we use it
34 %       in order to find the worst-case value for min_i [F(x_i)-F(xs)]
35
36 % we create an array to save all function values (so that we can evaluate
37 % them afterwards)
38 f_saved=cell(N+1,1);
39 for i=1:N
40     [g,f]=F.oracle(x);
41     f_saved{i}=f;
42     P.PerformanceMetric(f-fs);
43     x=x-h(i)*g;
44 end
45
46 [g,f]=F.oracle(x);
47 f_saved{N+1}=f;
48 P.PerformanceMetric(f-fs);
49
50 % (5) Solve the PEP
51 P.solve()
52
53 % (6) Evaluate the output
54 for i=1:N+1
55     f_saved{i}=double(f_saved{i});
56 end
57 f_saved
58 % The result should be 1/sqrt(N+1).
```

5.3   Smooth convex minimization

Different examples are provided in the `Examples` directory of the toolbox, including the vanilla gradient method or the celebrated fast gradient method [24]. In this section, we provide three examples: the Optimized gradient method (OGM) [16] (first numerical version in [1, 2]), a greedy conjugate gradient-like method [13], and the Optimized gradient method for gradient norm (OGM-G) [17].

It was established in [25] that OGM enjoys the best possible worst-case guarantee on function values for smooth convex minimization. By using the following code, the reader can verify that both OGM and conjugate gradient achieves this lower bound.

5.3.1   Optimized gradient method

```matlab
1  % In this example, we use the optimized gradient method (OGM) for
2  % solving the L-smooth convex minimization problem
3  %   min_x F(x); for notational convenience we denote xs=argmin_x F(x).
4  %
5  % We show how to compute the worst-case value of F(xN)-F(xs) when xN is
6  % obtained by doing N steps of OGM starting with an initial iterate
7  % satisfying ||x0-xs||≤1.
8
9  % (0) Initialize an empty PEP
10 P=pep();
11
12 % (1) Set up the objective function
13 param.L=1;      % Smoothness parameter
14
15 F=P.DeclareFunction('SmoothStronglyConvex',param); % F is the objective function
16
17 % (2) Set up the starting point and initial condition
18 x0=P.StartingPoint();        % x0 is some starting point
19 [xs,fs]=F.OptimalPoint();        % xs is an optimal point, and fs=F(xs)
20 P.InitialCondition((x0-xs)^2<=1); % Add an initial condition ||x0-xs||^2≤ 1
21
22 % (3) Algorithm
23 gam=1/param.L;      % step size
24 N=5;        % number of iterations
25
26 x=cell(N+1,1);%we store all the x's in a cell (for convenience)
27 x{1}=x0;
28 y=x0;
29 theta=1;
30 for i=1:N
31     x{i+1}=gradient_step(y,F,gam);
32     theta_prev=theta;
33     if i<N
34         theta=(1+sqrt(4*theta^2+1))/2;
35     else
36         theta=(1+sqrt(8*theta^2+1))/2;
37     end
38     y=x{i+1}+(theta_prev-1)/theta*(x{i+1}-x{i})+theta_prev/theta*(x{i+1}-y);
39 end
40
41 % (4) Set up the performance measure
42 fN=F.value(y);                % g=grad F(x), f=F(x)
43 P.PerformanceMetric(fN-fs); % Worst-case evaluated as F(x)-F(xs)
44
45 % (5) Solve the PEP
46 P.solve()
47
48 % (6) Evaluate the output
49 double(fN-fs)   % worst-case objective function accuracy
50 % The result should be 1/2/theta^2
```

5.3.2   Conjugate gradient method (or greedy first-order method)

```matlab
1  % In this example, we use a greedy first-order method (GFOM), or conjugate
2  % gradient, for solving the L-smooth (possibly mu-strongly)
3  % convex minimization problem
4  %   min_x F(x); for notational convenience we denote xs=argmin_x F(x).
5  %
6  % We show how to compute the worst-case value of F(xN)-F(xs) when xN is
7  % obtained by doing N steps of the gradient method starting with an initial
8  % iterate satisfying ||x0-xs||<=1.
9
10
11 % (0) Initialize an empty PEP
12 P=pep();
13
14 % (1) Set up the objective function
15 param.L=1;       % Smoothness parameter
16 param.mu=0.0;    % Strong convexity parameter
17
18 % F is the objective function
19 F=P.DeclareFunction('SmoothStronglyConvex',param);
20
21 % (2) Set up the starting point and initial condition
22 x0=P.StartingPoint();        % x0 is some starting point
23 [xs,fs]=F.OptimalPoint();        % xs is an optimal point, and fs=F(xs)
24 P.InitialCondition((x0-xs)^2<=1); % Add an initial condition ||x0-xs||^2<= 1
25
26 % (3) Algorithm
27 N=5;         % number of iterations
28
29 x=cell(N+1,1);%we store all the x's in a cell (for convenience)
30 g=cell(N+1,1);%we store all the g's in a cell (for convenience)
31 x{1}=x0;
32 g{1}=F.gradient(x{1});
33 dirs{1}=g{1};
34 for i=1:N
35     [x{i+1}, g{i+1}] = exactlinesearch_step(x{i},F,dirs);
36     dirs{2+(i-1)*2}  = x{i+1} - x{1};
37     dirs{3+(i-1)*2}  = g{i+1};
38 end
39
40 % (4) Set up the performance measure
41 fN=F.value(x{N+1});                  % g=grad F(x), f=F(x)
42 P.PerformanceMetric(fN-fs); % Worst-case evaluated as F(x)-F(xs)
43
44 % (5) Solve the PEP
45 P.solve()
46
47 % (6) Evaluate the output
48 double(fN-fs)    % worst-case objective function accuracy
49 % The results are the same as those for the optimized gradient method.
```

5.3.3 Optimized gradient method for gradient norm

```
1   % In this example, we use the optimized gradient method for gradient norm
2   % (OGM-G) for solving the L-smooth convex minimization problem
3   %    min_x F(x); for notational convenience we denote xs=argmin_x F(x).
4   %
5   % We show how to compute the worst-case value of ||F'(xN)||^2 when xN is
6   % obtained by doing N steps of OGM-G starting with an initial
7   % iterate satisfying F(x0)-F(x*)≤1.
8
9
10  % (0) Initialize an empty PEP
11  P=pep();
12  L = 1;
13  % (1) Set up the objective function
14  param.L=L;        % Smoothness parameter
15
16  F=P.DeclareFunction('SmoothStronglyConvex',param); % F is the objective function
17
18  % (2) Set up the starting point and initial condition
19  x0=P.StartingPoint();           % x0 is some starting point
20  [xs,fs]=F.OptimalPoint();       % xs is an optimal point, and fs=F(xs)
21  [g0,f0]=F.oracle(x0);
22  P.InitialCondition(f0-fs≤1);    % Add an initial condition F(x0)-F(x*)≤1.
23
24  % (3) Algorithm
25  gam=1/param.L;        % step size
26  N=5;          % number of iterations
27
28  x=cell(N+1,1);%we store all the x's in a cell (for convenience)
29  x{1}=x0;
30  g{1}=g0;
31  y{1}=x0;
32  theta(1)=1;
33  for i=1:N
34      if i<N
35          theta(i+1)=(1+sqrt(4*theta(i)^2+1))/2;
36      else
37          theta(i+1)=(1+sqrt(4*theta(i)^2+1))/2;
38      end
39  end
40  th_ti = @(i)theta(N+1-i);
41
42  for i = 1:N
43      y{i+1} = x{i} - 1/L * g{i};
44      cc       = (2*th_ti(i)-1)/(2*th_ti(i-1)-1);
45      x{i+1} = y{i+1} + (th_ti(i-1)-1)/th_ti(i-1)*cc*(y{i+1}-y{i}) + cc * (y{i+1}-x{i});
46      g{i+1}   = F.gradient(x{i+1});
47  end
48
49  % (4) Set up the performance measure
50  % gN=F.gradient(x{N+1});                % g=grad F(x), f=F(x)
51  obj = (g{N+1})^2;
52  P.PerformanceMetric(obj); % Worst-case evaluated as F(x)-F(xs)
53
54  % (5) Solve the PEP
55  P.solve();
56
57  % (6) Evaluate the output
58  double(obj)   % worst-case objective function accuracy
59
60  % The result should be 2/theta(end)^2
```

5.4   Smooth strongly convex minimization

As for the previous cases, different examples are provided in the `Examples` directory of the toolbox. This section provides the code for four algorithms.

5.4.1   Gradient descent with exact line searches

```
1  % In this example, we use a gradient method with exact line search
2  % for solving the L-smooth mu-strongly convex minimization problem
3  %   min_x F(x); for notational convenience we denote xs=argmin_x F(x).
4  %
5  %   Starting from an iterate x0, the method performs at each iteration
6  %   an exact line search step in the steepest descent direction
7  %       gamma=argmin_gamma F(xi-gamma*gi), with gi the gradient of F at xi,
8  %   and performs the update
9  %       x{i+1}=xi-gamma*gi.
10 %
11 % We show how to compute the worst-case value of F(xN)-F(xs) when xN is
12 % obtained by doing N steps of the method starting with an initial
13 % iterate satisfying F(x0)-F(xs)≤1.
14
15 % (0) Initialize an empty PEP
16 P=pep();
17
18 % (1) Set up the objective function
19 param.mu=.1;      % Strong convexity parameter
20 param.L=1;        % Smoothness parameter
21
22 F=P.DeclareFunction('SmoothStronglyConvex',param);
23 % F is the objective function
24
25 % (2) Set up the starting point and initial condition
26 x0=P.StartingPoint();         % x0 is some starting point
27 [xs,fs]=F.OptimalPoint();         % xs is an optimal point, and fs=F(xs)
28 [g0,f0]=F.oracle(x0);
29 P.InitialCondition(f0-fs≤1); % Add an initial condition f0-fs≤ 1
30
31 % (3) Algorithm
32 N=2;
33 x=x0;
34 for i=1:N
35     [g,¬]=F.oracle(x);
36     x=exactlinesearch_step(x,F,g);
37 end
38
39 % (4) Set up the performance measure
40 [g,f]=F.oracle(x);
41 P.PerformanceMetric(f-fs); % Worst-case evaluated as F(x)-F(xs)
42
43 % (5) Solve the PEP
44 P.solve()
45
46 % (6) Evaluate the output
47 double(f-fs)    % worst-case objective function accuracy
48
49 % The result should be
50 %((param.L-param.mu)/(param.L+param.mu))^(2*N)
```

5.4.2   Inexact gradient descent with exact line search

```matlab
1  % In this example, we use an inexact gradient method with exact line search
2  % for solving the L-smooth mu-strongly convex minimization problem
3  %    min_x F(x); for notational convenience we denote xs=argmin_x F(x).
4  %
5  %    Starting from an iterate x0, the method performs at each iteration
6  %    an exact line search step in a direction d satisfying a relative
7  %    accuracy criterion
8  %    (gi is the gradient of F at xi)
9  %        ||d-gi||≤eps*||gi||  (**)
10 %    that is, the method evaluates
11 %        gamma=argmin_gamma F(xi-gamma*d) for d satisfying (**)
12 %    and performs the update
13 %        x{i+1}=xi-gamma*d.
14 %
15 % We show how to compute the worst-case value of F(xN)-F(xs) when xN is
16 % obtained by doing N steps of the method starting with an initial
17 % iterate satisfying F(x0)-F(xs)≤1.
18
19 % (0) Initialize an empty PEP
20 P=pep();
21
22 % (1) Set up the objective function
23 param.mu=.1;     % Strong convexity parameter
24 param.L=1;       % Smoothness parameter
25
26 F=P.DeclareFunction('SmoothStronglyConvex',param);
27 % F is the objective function
28
29 % (2) Set up the starting point and initial condition
30 x0=P.StartingPoint();        % x0 is some starting point
31 [xs,fs]=F.OptimalPoint();    % xs is an optimal point, and fs=F(xs)
32 [g0, f0]=F.oracle(x0);
33 P.InitialCondition(f0-fs≤1); % Add an initial condition f0-fs≤ 1
34
35 % (3) Algorithm
36 N=2;
37 eps=0.1;
38 x=x0;
39 for i=1:N
40     d=inexactsubgradient(x,F,eps,0);
41     x=exactlinesearch_step(x,F,d);
42 end
43 fN=F.value(x);
44 % (4) Set up the performance measure
45 P.PerformanceMetric(fN-fs); % Worst-case evaluated as ||g||^2
46
47 % (5) Solve the PEP
48 P.solve()
49
50 % (6) Evaluate the output
51 double(fN-fs)   % worst-case objective function accuracy
52
53 % The result should be
54 %((param.L*(1+eps)-param.mu*(1-eps))/(param.L*(1+eps)+param.mu*(1-eps)))^(2*N)
```

5.4.3   Triple momentum method

```matlab
1  % In this example, we use the triple momentum method for solving the
2  % L-smooth mu-strongly convex minimization problem
3  %    min_x F(x);
4  %    for notational convenience we denote xs=argmin_x F(x).
5  %
6  % We show how to compute the worst-case value of F(xN)-F(xs) when xN is
7  % obtained by doing N steps of the method starting with an initial
8  % iterate satisfying ||x0-xs||<=1.
9
10
11  % (0) Initialize an empty PEP
12  P=pep();
13
14  % (1) Set up the objective function
15  param.L=1;       % Smoothness parameter
16  param.mu=.1;     % Strong convexity parameter
17
18  F=P.DeclareFunction('SmoothStronglyConvex',param); % F is the objective function
19
20  % (2) Set up the starting point and initial condition
21  x0=P.StartingPoint();          % x0 is some starting point
22  [xs,fs]=F.OptimalPoint();      % xs is an optimal point, and fs=F(xs)
23  P.InitialCondition((x0-xs)^2<=1); % Add an initial condition ||x0-xs||^2<= 1
24
25  % (3) Algorithm
26  N=10;        % number of iterations
27
28  kappa=param.mu/param.L;
29  rho=(1-sqrt(kappa));
30
31  alpha=(1+rho)/param.L; beta=rho^2/(2-rho); gamma=rho^2/(1+rho)/(2-rho);
32  Δ=rho^2/(1-rho^2);
33
34  xsi=cell(N+1,1);% we store the iterates in a cell for convenience
35  xsi{1}=x0;
36  xsi{2}=x0;
37  y=x0;
38  for i=2:N+1
39      xsi{i+1}=(1+beta)*xsi{i}-beta*xsi{i-1}-alpha*F.gradient(y);
40      y=(1+gamma)*xsi{i+1}-gamma*xsi{i};
41      x=(1+Δ)*xsi{i+1}-Δ*xsi{i};
42  end
43
44  % (4) Set up the performance measure
45  fN=F.value(x);          % fN=F(xN)
46  P.PerformanceMetric(fN-fs); % Worst-case evaluated as F(x)-F(xs)
47
48  % (5) Solve the PEP
49  P.solve()
50
51  % (6) Evaluate the output
52  double(fN-fs)    % worst-case objective function accuracy
53
54  % Should at most match the standard guarantees
55  % f(xN)-f(xs)<= (1-sqrt(kappa))^(2*N)*param.L/2/kappa*double((x0-xs)^2)
56  % ||xN-xs||^2<= (1-sqrt(kappa))^(2*N)/kappa*double((x0-xs)^2)
```

5.4.4   Robust momentum method

```matlab
% In this example, we use the robust momentum method for solving the
% L-smooth mu-strongly convex minimization problem
%    min_x F(x);
%    for notational convenience we denote xs=argmin_x F(x).
% We show how to compute the rate of the Lyapunov function developed in
%(**) Cyrus, S., Hu, B., Van Scoy, B., & Lessard, L. "A robust accelerated
%      optimization algorithm for strongly convex functions." In 2018 Annual
%      American Control Conference (ACC) (pp. 1376-1381). IEEE.

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

% (1) Set up the objective function
L  = 1;  param.L=L;       % Smoothness parameter
mu = .2; param.mu=mu;     % Strong convexity parameter

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

% (2) Set up the starting point and initial condition
xm1=P.StartingPoint();              % xm1 is some starting point
x0=P.StartingPoint();               % x0 is some starting point
[xs,fs]=F.OptimalPoint();           % xs is an optimal point, and fs=F(xs)

% (3) Algorithm' parameters
kappa = param.L/param.mu;
lam   = .1;
rho   = lam*(1-1/kappa) + (1-lam)*(1-1/sqrt(kappa));

alpha  = kappa*(1-rho)^2*(1+rho)/L;
beta   = kappa*rho^3/(kappa-1);
gamma  = rho^3/((kappa-1)*(1-rho)^2*(1+rho));
lambda = mu^2*(kappa-kappa*rho^2-1)/(2*rho*(1-rho));
nnu    = (1+rho)*(1-kappa+2*kappa*rho-kappa*rho^2)/(2*rho);

% (4) Perform two iterations according to the notations in (**)
y0      = x0+gamma*(x0-xm1);
[g0,F0] = F.oracle(y0);
x1      = x0 + beta*(x0-xm1) - alpha*g0;
y1      = x1 + gamma*(x1-x0);
[g1,F1] = F.oracle(y1);
x2      = x1+beta*(x1-x0)-alpha*g1;

z0      = (x0-rho^2*xm1)/(1-rho^2);
z1      = (x1-rho^2*x0)/(1-rho^2);
z2      = (x2-rho^2*x1)/(1-rho^2);

% (5) Evaluate the Lyapunov function at the first and second iterations
q0 = (L-mu)*(F0-fs-mu/2*(y0-xs)^2)-1/2*(g0-mu*(y0-xs))^2;
q1 = (L-mu)*(F1-fs-mu/2*(y1-xs)^2)-1/2*(g1-mu*(y1-xs))^2;

initLyapunovValue   = lambda*(z1-xs)^2+q0;
finalLyapunovValue  = lambda*(z2-xs)^2+q1;

% (6) Normalize the initial value of the Lyapunov function to one and observe
% that the value of the second indeed corresponds to the rate.
P.InitialCondition(initLyapunovValue <= 1);      % (initial condition)
P.PerformanceMetric(finalLyapunovValue);         % (performance measure)

% (5) Solve the PEP
P.solve(1);
% the value of final should match rho^2
[double(finalLyapunovValue) rho^2]
```

5.5   Composite minimization and operator splitting

A series of examples for this setting are presented in the directory `Examples`. Those examples include a code to reproduce the analysis [26] for the proximal gradient method, a code for obtaining tight worst-case bounds for the celebrated fast iterative shrinkage-thresholding algorithm (FISTA) [27], and a code for studying conditional gradient methods (a.k.a. Frank-Wolfe [28]). We provide below examples for studying the Douglas-Rachford splitting in different settings.

5.5.1   Douglas-Rachford splitting for monotone inclusion

```
1   % In this example, we use a Douglas-Rachford splitting (DRS)
2   % method for solving a monotone inclusion problem
3   %   find x st   0 \in Ax + Bx
4   % where A is L-Lipschitz and monotone and B is (maximally) mu-strongly
5   % monotone. We denote by JA and JB the respective resolvents of A and B.
6   %
7   % One iteration of the algorithm starting from point w is as follows:
8   %       x = JB( w )
9   %       y = JA( 2 * x - w )
10  %       z = w - theta * ( x - y )
11  % and then we choose as the next iterate the value of z.
12  %
13  % Given two initial points w0 and w1, we show how to compute the worst-case
14  % contraction factor ||z0 - z1||/||w0 - w1|| obtained after doing one
15  % iteration of DRS from respectively w0 and w1.
16  % Note that we allow the user to choose a stepsize alpha in the resolvent.
17  %
18  % This setting is studied in
19  % (**) Ernest K. Ryu, Adrien B. Taylor, C. Bergeling, and P. Giselsson.
20  %       "Operator Splitting Performance Estimation: Tight contraction
21  %        factors and optimal parameter selection." arXiv:1812.00146, 2018.
22  %
23  % (0) Initialize an empty PEP
24  P=pep();
25
26  % (1) Set up the class of monotone inclusions
27  paramA.L  =  1; paramA.mu = 0; % A is 1-Lipschitz and 0-strongly monotone
28  paramB.mu = .1;                % B is .1-strongly monotone
29  A = P.DeclareFunction('LipschitzStronglyMonotone',paramA);
30  B = P.DeclareFunction('StronglyMonotone',paramB);
31
32  % (2) Set up the starting points
33  w0=P.StartingPoint(); w1=P.StartingPoint();
34  P.InitialCondition((w0-w1)^2<=1);  % Normalize the initial distance ||w0-ws||^2 <= 1
35
36  % (3) Algorithm
37  alpha = 1.3;        % step size (in the resolvents)
38  theta = .9;         % overrelaxation
39
40  x0 = proximal_step(w0,B,alpha);
41  y0 = proximal_step(2*x0-w0,A,alpha);
42  z0 = w0-theta*(x0-y0);
43
44  x1 = proximal_step(w1,B,alpha);
45  y1 = proximal_step(2*x1-w1,A,alpha);
46  z1 = w1-theta*(x1-y1);
47
48  % (4) Set up the performance measure: ||z0-z1||^2
49  P.PerformanceMetric((z0-z1)^2);
50  % (5) Solve the PEP
51  P.solve()
52  % (6) Evaluate the worst-case contraction factor; it should match the result from (**)
53  double((z0-z1)^2)
```

5.5.2 Douglas-Rachford splitting for composite minimization

```matlab
1  % In this example, we use a Douglas-Rachford splitting (DRS)
2  % method for solving the composite convex minimization problem
3  %    min_x F(x)=f_1(x)+f_2(x)
4  %    (for notational convenience we denote xs=argmin_x F(x);
5  % where f_1(x) is L-smooth and mu-strongly convex, and f_2 is convex.
6  %
7  % We show how to compute the worst-case value of ||wN-ws||^2 when wN is
8  % obtained by doing N steps of DRS starting with an initial iterate w0
9  % satisfying ||w0-ws||<1, and ws is some point to which the iterates of
10 % DRS converge.
11 %
12 % Note that the point ws may be defined in the following way:
13 % let g1s and g2s be subgradients of respectively f_1 and f_2 at xs such
14 % that g1s+g2s=0 (optimality conditions). Then, ws=xs+lambda*g2s (lambda is
15 % the step size used in DRS).
16
17 % (0) Initialize an empty PEP
18 P=pep();
19
20 % (1) Set up the objective function
21 paramf1.mu=.1;  % Strong convexity parameter
22 paramf1.L=1;       % Smoothness parameter
23 f1=P.DeclareFunction('SmoothStronglyConvex',paramf1);
24 f2=P.DeclareFunction('Convex');
25 F=f1+f2; % F is the objective function
26
27 % (2) Set up the starting point and initial condition
28 w0=P.StartingPoint(); % x0 is some starting point
29 [xs,fs]=F.OptimalPoint('opt'); % xs is an optimal point, and fs=F(xs)
30
31 % note that we tag the point xs as 'opt' to be able to re-evaluate it
32 % easily (providing the oracle routine with this tag allows to recover
33 % previously evaluated points).
34
35 % the next step evaluates the oracle at the tagged point 'opt' (xs) for
36 % recovering the values of g1s and g2s; this allows to guarantee that
37 % g1s+g2s=0;
38 [g1s,¬]=f1.oracle('opt');
39 [g2s,¬]=f2.oracle('opt');
40 lambda=2; ws=xs+lambda*g2s;
41
42 % Add an initial condition ||w0-ws||^2< 1
43 P.InitialCondition((w0-ws)^2-1<0);
44
45 % (3) Algorithm
46 N=5;              % number of iterations
47 gam=lambda;     % step size
48
49 w=w0;
50 for i=1:N
51     x=proximal_step(w,f2,gam);
52     y=proximal_step(2*x-w,f1,gam);
53     w=y-x+w;
54 end
55
56 % (4) Set up the performance measure
57 P.PerformanceMetric((w-ws)^2);
58
59 % (5) Solve the PEP
60 P.solve()
61
62 % (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).
```

5.5.3   Three-operator splitting

```matlab
1  % In this example, we use a Three-Operator Splitting method for solving
2  % a monotone inclusion problem
3  %    find x st   0 \in Ax + Bx + Cx
4  % where A is maximally monotone, B is cocoercive and C is the gradient of
5  % a smooth strongly convex function. We denote by JA and JB the respective
6  % resolvents of A and B.
7  %
8  % One iteration of the algorithm starting from point w is as follows:
9  %        x = JB( w )
10 %        y = JA( 2 * x - w - C x)
11 %        z = w - theta * ( x - y )
12 % and then we choose as the next iterate the value of z.
13 %
14 % Given two initial points w0 and w1, we show how to compute the worst-case
15 % contraction factor ||z0 - z1||/||w0 - w1|| obtained after doing one
16 % iteration of DRS from respectively w0 and w1. Note that we allow the
17 % user to choose a stepsize alpha in the resolvent and for C.
18 %
19 % (0) Initialize an empty PEP
20 P=pep();
21
22 % (1) Set up the class of monotone inclusions
23 paramB.beta = 1;           % B is 1-cocoercive
24 paramC.L = 1; paramC.mu=.1; % C is the gradient of a 1-smooth .1-str convex
25
26 A = P.DeclareFunction('Monotone');
27 B = P.DeclareFunction('Cocoercive',paramB);
28 C = P.DeclareFunction('SmoothStronglyConvex',paramC);
29
30 % (2) Set up the starting points
31 w0=P.StartingPoint(); w1=P.StartingPoint();
32 P.InitialCondition((w0-w1)^2<=1);  % Normalize the initial distance ||w0-ws||^2 <= 1
33
34 % (3) Algorithm
35 alpha = .9;      % step size (in the resolvents)
36 theta = 1.3;     % overrelaxation
37
38 x0 = proximal_step(w0,B,alpha);
39 y0 = proximal_step(2*x0-w0-alpha*C.evaluate(x0),A,alpha);
40 z0 = w0-theta*(x0-y0);
41
42 x1 = proximal_step(w1,B,alpha);
43 y1 = proximal_step(2*x1-w1-alpha*C.evaluate(x1),A,alpha);
44 z1 = w1-theta*(x1-y1);
45
46 % (4) Set up the performance measure: ||z0-z1||^2
47 P.PerformanceMetric((z0-z1)^2);
48
49 % (5) Solve the PEP
50 P.solve()
51
52 % (6) Evaluate the output
53 double((z0-z1)^2)   % worst-case contraction factor
```

5.6   Fixed-point iterations of nonexpansive mappings

Performance estimation problems have very recently been applied to the convergence analysis of fixed-point iterations of non-expansive mappings in [22, 23]. This kind of analysis can also be done with the toolbox and we give the corresponding examples below.

5.6.1   Halpern iterations

```matlab
% In this example, we use the Halpern iteration for finding a fixed point
% to the non-expansive operator A :
%   find x such that  x = Ax
%
% (**) Lieder, Felix. "On the Convergence Rate of the Halpern-Iteration."
%      (2017)


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

% (1) Set up the objective function
paramA.L=1;       % A is 1-Lipschitz (non-expansive)
A = P.DeclareFunction('Lipschitz',paramA);

% (2) Set up the starting point and initial condition
x0 = P.StartingPoint();        % x0 is some starting point
xs = fixedpoint(A);
P.InitialCondition((x0-xs)^2<=1); % Add an initial condition ||x0-xs||^2<= 1

% (3) Algorithm
N = 10;
lambda = @(k)(1/(k+2));
x=x0;
for i=1:N
    x = lambda(i-1) * x0 + (1-lambda(i-1)) * A.evaluate(x);
end
xN  = x;
AxN = A.evaluate(xN);
% (4) Set up the performance measure
P.PerformanceMetric((xN-AxN)^2); % Worst-case evaluated as F(x)-F(xs)

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

% (6) Evaluate the output
double((xN-AxN)^2)    % worst-case objective function accuracy

% The result should be (2/(N+1))^2 as in (**)
```

5.6.2   Krasnoselskii–Mann

```matlab
1  % In this example, we use Krasnoselskii-Mann iterations for finding a
2  % fixed point to the non-expansive operator A :
3  %    find x such that  x = Ax
4  %
5  % This scheme was first studied using PEPs in:
6  % (**) Felix Lieder. "Projection Based Methods for Conic Linear Programming
7  %         Optimal First Order Complexities and Norm Constrained Quasi Newton
8  %         Methods."  PhD thesis (2018)
9
10
11 % (0) Initialize an empty PEP
12 P=pep();
13
14 % (1) Set up the objective function
15 paramA.L=1;       % A is 1-Lipschitz (non-expansive)
16 A = P.DeclareFunction('Lipschitz',paramA);
17
18 % (2) Set up the starting point and initial condition
19 x0 = P.StartingPoint();      % x0 is some starting point
20 xs = fixedpoint(A);
21 P.InitialCondition((x0-xs)^2<=1); % Add an initial condition ||x0-xs||^2<= 1
22
23 % (3) Algorithm
24 N = 10;
25 lambda = @(k)(1/(k+2));
26 x=x0;
27 for i=1:N
28     x = lambda(i-1) * x + (1-lambda(i-1)) * A.evaluate(x);
29 end
30 xN  = x;
31 AxN = A.evaluate(xN);
32 % (4) Set up the performance measure
33 P.PerformanceMetric((xN-AxN)^2); % Worst-case squared residual
34
35 % (5) Solve the PEP
36 P.solve()
37
38 % (6) Evaluate the output
39 double((xN-AxN)^2)    % worst-case squared residual
```

5.7   Geometry of smooth strongly convex functions

The file `Examples/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.

```matlab
1   % number of tests and vector of kappa to be tested
2   n_test = 80;
3   kappa = 1.025.^(1:n_test);
4   verbose = 0; % verbose mode of the toolbox
5
6
7   for k = 1:n_test;
8
9       % (0) Information about current test
10      disp('--------------------------------------------------')
11      disp(['Test ', num2str(k) , ' of ', num2str(n_test)])
12      disp(['condition number = ' , num2str(round(kappa(k)*10^3)/10^3)])
13      disp('--------------------------------------------------')
14
15      % (1) Declaration of the "PEP"
16      P=pep();
17
18      % (2) functions parameters (using scale_invariance)
19      param.mu=1; % Strong convexity parameter
20      param.L=kappa(k);       % Smoothness parameter
21
22      % (3) functions declarations
23      f= P.DeclareFunction('SmoothStronglyConvex',param);
24      g= P.DeclareFunction('SmoothStronglyConvex',param);
25      fg = f+g;
26
27      % (4) decleration of the minimizers
28      [xsf,fs] = f.OptimalPoint();
29      [xsg,gs] = g.OptimalPoint();
30      [xsfg,fgs] = fg.OptimalPoint();
31
32      % (5) Contraint on the minimizer (have to be declared as "initial
33      %condition")
34      P.InitialCondition((xsf-xsg)^2<=1);
35      % note: the problem will naturally force (xsf-xsg)^2 = 1
36
37      % (6) Criterion to be maximized
38      P.PerformanceMetric((xsfg-(xsf+xsg)/2 )^2 );
39
40      % (7) Solving the Pep
41      P.solve(verbose);
42
43      % (8) Evaluation and storage the output
44      distance_sq(k) =double((xsfg-(xsf+xsg)/2 )^2);
45
46  end
```

```matlab
47
48
49  % representation of the results
50  distance = sqrt(distance_sq)
51
52  figure
53  plot(kappa,distance_sq)
54  xlabel('condition number');
55  ylabel('||x^s_{fg} - \frac{1}{2}(x^s_f+x^s_g)||^2')
56  title('square distance to average minimizers f, g')
57
58  figure
59  plot(kappa,distance)
60  xlabel('condition number');
61  ylabel('||x^s_{fg} - \frac{1}{2}(x^s_f+x^s_g)||')
62  title('distance to average minimizers f, g')
63
64
65  disp('***********************************************************')
66  disp('Computation ended')
67  disp('The result should show the square distance is asymptotically')
68  disp('linear in the condition number')
69  disp('***********************************************************')
```

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*, 27(3):1283–1313, 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] Adrien Taylor, Julien Hendrickx, and François 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] 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.

[12] Adrien Taylor, Bryan Van Scoy, and Laurent Lessard. Lyapunov functions for first-order methods: Tight automated convergence guarantees. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 4897–4906. PMLR, 10–15 Jul 2018.

[13] Yoel Drori and Adrien B Taylor. Efficient first-order methods for convex minimization: a constructive approach. *preprint arXiv:1803.05676*, 2018.

[14] 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*, 11(7):1185–1199, 2017.

[15] Ernest K Ryu, Adrien B Taylor, Carolina Bergeling, and Pontus Giselsson. Operator splitting performance estimation: Tight contraction factors and optimal parameter selection. *preprint arXiv:1812.00146*, 2018.

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

[17] Donghwan Kim and Jeffrey A Fessler. Optimizing the efficiency of first-order methods for decreasing the gradient of smooth convex functions. *preprint arXiv:1803.06600*, 2018.

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

[19] Saman Cyrus, Bin Hu, Bryan Van Scoy, and Laurent Lessard. A robust accelerated optimization algorithm for strongly convex functions. In *2018 Annual American Control Conference (ACC)*, pages 1376–1381. IEEE, 2018.

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

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

[22] Felix Lieder. On the convergence rate of the halpern-iteration. 2017.

[23] Felix 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.

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

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

[26] Adrien B Taylor, Julien M Hendrickx, and François Glineur. Exact worst-case convergence rates of the proximal gradient method for composite convex minimization. *Journal of Optimization Theory and Applications*, pages 1–22.

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

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