PEPFlow: A Python Library for the Workflow of Performance Estimation of Optimization Algorithms

Jaewook J. Suh
CMOR Department
Rice University
jacksuh@rice.edu

Bicheng Ying
Google Inc.
ybc@google.com

Xin Jiang
ISE Department
University of Houston
xinjiang@uh.edu

Edward D. H. Nguyen
ECE Department
Rice University
en18@rice.edu

Abstract

We present PEPFlow, a Python library designed to streamline the workflow for analyzing the convergence behavior of a variety of first-order optimization algorithms. The library builds on Performance Estimation Problems (PEPs), which reformulate the worst-case convergence guarantees as convex semidefinite programs (SDPs). Solving the SDP provides numerical evidence of convergence rates, and more importantly, its dual variables can be leveraged to construct analytical proofs. PEPFlow supports the entire workflow by automating SDP formulation, exploring relaxations, inspecting dual variables, and verifying proofs. Together, these features bridge numerical verification with rigorous analysis and substantially reduce manual effort. A pre-release version of PEPFlow is available at:

https://github.com/pepflow-lib/PEPFlow.

1 Introduction

First-order optimization methods are among the most widely used tools in modern optimization and machine learning, valued for their simplicity, scalability, and ease of implementation. Their versatility has made them popular with both practitioners and researchers. However, analyzing their convergence behavior remains technically demanding and often limited to experts.

Performance Estimation Problems (PEPs) [9, 33] provide a systematic framework for such analyses, and extensive research has been devoted to their development and applications [1–6, 8, 10, 11, 13–16, 18–24, 26–32, 34–37]. The worst-case convergence behavior of a first-order optimization method is formulated as a tractable convex optimization problem, referred to as *Primal PEP*, whose solution offers numerical verification of the algorithm's convergence guarantees. More importantly, the dual variables of Primal PEP provide the key ingredients for constructing analytical convergence proofs, thereby turning numerical evidence into rigorous theoretical guarantees.

Despite the power of the PEP framework, existing tools such as PEPit [12] focus primarily on automating the numerical setup, with limited support for the subsequent and more challenging proof stage. This leaves a gap between numerical evidence and analytical verification.

To address this limitation, we introduce PEPFlow, a Python library that assists with the complete PEP workflow. In addition to formulating and solving Primal PEP, PEPFlow provides tools for exploring its relaxations, inspecting dual solutions, and symbolically verifying proofs. Together, these features reduce the manual effort required in traditional PEP analysis and enable a seamless workflow from problem setup to convergence proof.

Our main contributions are as follows.

- We introduce PEPFlow, a Python library covering the PEP workflow end-to-end.
- We provide new functionality to support analytical proof construction, including an interactive dashboard for inspecting relaxations of Primal PEP and structured access to dual variables.
- We demonstrate the library through the example of gradient descent, with a particular focus on how it streamlines convergence analysis.

The remainder of the paper is organized as follows. Section 2 outlines the PEP workflow and illustrates how PEPFlow assists at each stage, Section 3 presents a working example and details the library's key features, and Section 4 concludes with future directions.

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: ScaleOPT: GPU-Accelerated and Scalable Optimization.

2 The Workflow of PEP and PEPFlow

This section overviews the PEP workflow and shows how PEPFlow assists at each stage, from formulating and solving Primal PEP to deriving analytical proofs, while highlighting key features in PEPFlow that streamline and automate the process.

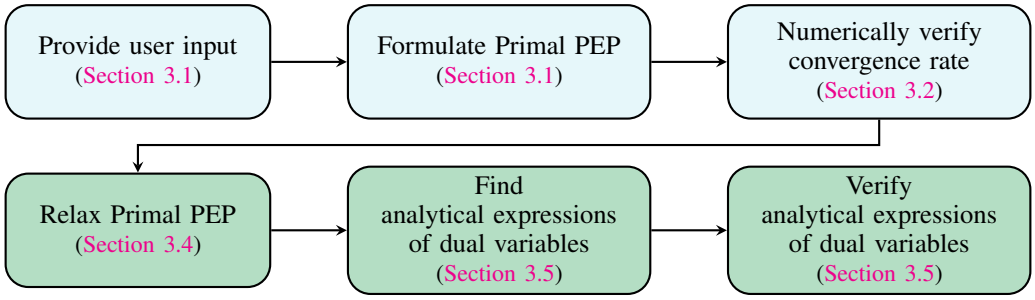


Figure 1: A diagram of the PEP workflow. Existing libraries such as PEPit [12] have features that assist users with the steps in blue. PEPFlow has new features that assist users with the steps in green.

A typical PEP workflow begins with the user specifying key ingredients: the problem/function class, algorithm update rule, performance metric, optimality criterion, and initial condition. From this information, one formulates the *Primal PEP*, a semidefinite program (SDP) whose solution characterizes the algorithm's worst-case convergence behavior. Solving Primal PEP for different iteration counts verifies convergence and reveals the numerical rate. PEPFlow automates this stage: it allows users to define the problem in a high-level interface, automatically generates Primal PEP, and passes it to efficient SDP solvers. This greatly reduces the technical burden on users and makes numerical experimentation straightforward.

After numerical evidence is obtained, the next step in the PEP workflow is to derive a formal convergence proof. This requires identifying analytical expressions for the dual variables (Lagrange multipliers) of Primal PEP and verifying them numerically. This typically requires an exact relaxation of Primal PEP, which reduces the number of constraints and hence the number of nontrivial dual variables. The resulting sparse dual solution helps reveal candidate formulas for the multipliers that can then be tested and verified. Once confirmed, substituting these expressions into the Lagrangian yields an equality that, when symbolically verified, constitutes a rigorous convergence proof. PEPFlow supports this stage in the workflow by providing tools to inspect dual solutions, detect sparsity patterns, and test/verify candidate analytical formulas, enabling a smooth transition from numerical exploration to rigorous proofs.

Overall, PEPFlow is designed to support users throughout the entire PEP workflow. While other Python libraries such as PEPit [12] provide useful functionality, their features cover only a subset of the workflow (see Figure 1). In contrast, PEPFlow offers tools for every stage and, uniquely, introduces capabilities that assist users in establishing formal convergence proofs for a wide class of optimization algorithms. In particular, PEPFlow provides two critical new features:

- An interactive dashboard that allows users to search for exact relaxations of Primal PEP.
- Direct access to the mathematical objects required to derive and verify analytical expressions of the dual variables.

In the next section, we will present a working example of the PEP workflow, highlighting the parts that PEPFlow can automate or facilitate and those that still require user intervention.

3 A Working Example of PEPFlow

3.1 Primal PEP Problem Formulation

What sets PEPFlow apart from other PEP packages is its ability not only to automate the formulation process but also to support the derivation of analytical convergence proofs. To showcase this capability, we begin by introducing the key idea of PEP: formulating a tractable convex problem to evaluate the worst-case performance of first-order methods, using the gradient descent (GD) method as an example. Although PEPFlow automates this process, understanding the underlying formulation is crucial for deriving the analytical convergence proof of GD.

To begin, recall that GD is an iterative algorithm for minimizing a differentiable function $f\colon \mathbb{R}^d \to \mathbb{R}$, where f belongs to a given function class $\mathcal{F}$. Analyzing the convergence of GD requires specifying an initial condition on the starting point x_0 and selecting a performance measure that quantifies the progress of GD, which typically involves a minimizer of f, denoted by $x_\star$. With these ingredients, we can formulate the worst-case behavior of GD as an optimization problem:

$$\begin{array}{ll} \text{maximize} & \text{some performance measure} \\ \text{subject to} & f \text{ belongs to some function class } \mathcal{F}, \\ & \{x_n\}_{n=1}^N \text{ is generated by GD}, \\ & x_\star \text{ is a minimizer of } f, \\ & x_0 \text{ satisfies some initial condition}. \end{array} \tag{1}$$

To make this problem more concrete, we here consider as an example that f is an L-smooth convex function, the performance measure is $f(x_N) - f(x_\star)$, and the initial condition is $\|x_0 - x_*\|^2 \leq \rho^2$ with $\rho > 0$. Instead of working directly with the functional constraint $f \in \mathcal{F}$, one can equivalently enforce a set of so-called interpolation conditions that characterize L-smooth convex functions [9, 33]. These conditions are crucial in PEP, as they translate abstract functional properties into explicit inequalities relating the function values, gradients, and points visited by GD, yielding a finite-dimensional optimization problem. Defining the index sets $\mathcal{I}_N^\star := \{0, 1, \ldots, N, \star\}$ and $\mathcal{J}_N^\star := \{(i,j) \in \mathcal{I}_N^\star \times \mathcal{I}_N^\star \mid i \neq j\}$, we can reformulate (1) into the following nonconvex quadratically constrained quadratic program (QCQP):

$$\begin{array}{ll} \text{maximize} & f_N - f_\star \\ \text{subject to} & f_i \geq f_j + g_j^\mathsf{T}(x_i - x_j) + \frac{1}{2L}\|g_i - g_j\|_2^2, \quad (i,j) \in \mathcal{J}_N^\star \\ & x_i = x_{i-1} - \frac{1}{L}\sum_{j=0}^{i-1} g_j, \quad i \in 1, \ldots, N \\ & g_\star = 0 \\ & \|x_0 - x_\star\|_2^2 \leq \rho^2, \end{array} \tag{2}$$

where the decision variables are $\{x_i, f_i, g_i\}_{i \in \mathcal{I}_N^\star} \subseteq \mathbb{R}^d \times \mathbb{R} \times \mathbb{R}^d$, and for simplicity the stepsize in GD is chosen as $\frac{1}{L}$. The nonconvex QCQP can be further reformulated into a convex SDP by introducing the following change of variables

$$G := H^\mathsf{T} H \in \mathbb{S}_+^{N+3}, \quad \text{where} \quad H := [x_0 \quad g_0 \quad g_1 \quad \cdots \quad g_N \quad x_*] \in \mathbb{R}^{d \times (N+3)},$$
$$F := \left(f_0, f_1, \ldots, f_N, f_*\right) \in \mathbb{R}^{N+2}.$$

We introduce the following notation to select $\{x_i, f_i, g_i\}_{i \in \mathcal{I}_N^\star}$ from H and F:

$$\begin{array}{ll} \boldsymbol{f}_\star := e_{N+3} \in \mathbb{R}^{N+2}, & \boldsymbol{f}_i := e_{i+1} \in \mathbb{R}^{N+2},\ i \in \{0, \ldots, N\} \\ \boldsymbol{g}_\star := 0 \in \mathbb{R}^{N+3}, & \boldsymbol{g}_i := e_{i+2} \in \mathbb{R}^{N+3},\ i \in \{0, \ldots, N\} \\ \boldsymbol{x}_0 := e_1 \in \mathbb{R}^{N+3}, & \boldsymbol{x}_\star := e_{N+3} \in \mathbb{R}^{N+3}, \\ \boldsymbol{x}_i := \boldsymbol{x}_{i-1} - \frac{1}{L}\boldsymbol{g}_{i-1} \in \mathbb{R}^{N+3},\ i \in \{1, \ldots, N\}. & \end{array} \tag{3}$$

where e_i is the ith basis vector. This system of notation satisfies

$$x_i = H\boldsymbol{x}_i, \qquad f_i = F\boldsymbol{f}_i, \qquad g_i = H\boldsymbol{g}_i,$$

for all $i \in \mathcal{I}_N^\star$. Furthermore, for all $(i,j) \in \mathcal{J}_N^\star$, define

$$\begin{array}{ll} A_{i,j} := (\boldsymbol{x}_i - \boldsymbol{x}_j) \odot (\boldsymbol{x}_i - \boldsymbol{x}_j) \in \mathbb{S}_+^{N+3}, & B_{i,j} := \boldsymbol{g}_i \odot (\boldsymbol{x}_i - \boldsymbol{x}_j) \in \mathbb{S}^{N+3}, \\ C_{i,j} := (\boldsymbol{g}_i - \boldsymbol{g}_j) \odot (\boldsymbol{g}_i - \boldsymbol{g}_j) \in \mathbb{S}_+^{N+3}, & D_{i,j} := \boldsymbol{f}_j - \boldsymbol{f}_i \in \mathbb{R}^{N+2}, \end{array} \tag{4}$$

where $\odot$ denotes the symmetric outer product: $x \odot y = \frac{1}{2}(xy^\mathsf{T} + yx^\mathsf{T})$. These notations satisfy

$$\|x_i - x_j\|_2^2 = \mathbf{tr}\left(GA_{i,j}\right), \qquad g_j^\mathsf{T}(x_i - x_j) = \mathbf{tr}\left(GB_{i,j}\right), \qquad \|g_i - g_j\|_2^2 = \mathbf{tr}\left(GC_{i,j}\right).$$

With all the introduced notation, we can finally rewrite the nonconvex QCQP (2) as an SDP

$$\begin{array}{ll} \text{maximize} & F^\mathsf{T} D_{\star,N} \\ \text{subject to} & F^\mathsf{T} D_{i,j} + \mathbf{tr}\left(GB_{i,j}\right) + \frac{1}{2L}\mathbf{tr}\left(GC_{i,j}\right) \leq 0, \quad (i,j) \in \mathcal{J}_N^\star \\ & \mathbf{tr}\left(GA_{0,\star}\right) \leq \rho^2 \\ & G \succeq 0, \end{array} \tag{Primal PEP}$$

where the decision variables are $F \in \mathbb{R}^{N+2}$ and $G \in \mathbb{S}^{N+3}$. As an SDP, Primal PEP can be solved efficiently by numerical solvers. Moreover, Primal PEP can be easily implemented using `PEPFlow`, as illustrated below where we specifically consider $L = 1$ and $\rho = 1$ without loss of generality.

The following code block demonstrates how to use `PEPFlow` to formulate Primal PEP. Moreover, `PEPFlow` uses this information to automatically construct Primal PEP, which can be translated into a CVXPY problem solvable by various numerical solvers.

```python
import pepflow as pf

ctx_gd = pf.PEPContext("ctx_gd").set_as_current()
L, N = 1, 9  # Set the Lipschitz constant and the number of iterations.

f = pf.SmoothConvexFunction(is_basis=True, L=L, tags=["f"]) # Declare a function
x = pf.Vector(is_basis=True, tags=["x_0"]) # Define an initial point.
x_star = f.set_stationary_point("x_star")  # Define an optimal solution.
init_dist = (x - x_star) ** 2

for i in range(1, N+1):
    x = x - 1 / L * f.grad(x)  # Implement Gradient Descent.
    x.add_tag(f"x_{i}")   # Add tag for this point.

# Set the initial constraint ('lt' means less than).
pb = pf.PEPBuilder(ctx_gd)
pb.add_initial_constraint(init_dist.lt(1,name="initial_condition"))
# Set the performance metric.
pb.set_performance_metric(f(x) - f(x_star))
pb.solve_primal() # or pb.solve_dual() to get the pep problem result.
```

3.2 Numerical Verification of the Convergence Rate

By solving Primal PEP numerically for different iteration counts N and examining the resulting objective values, we can verify the convergence of GD and estimate its convergence rate. Plotting these objective values yields the following figure. The first three values indicate $\frac{1}{2} = 0.500$, $\frac{1}{6} \approx 0.167$, and $\frac{1}{10} = 0.100$. This suggests the pattern $\frac{1}{4N+2}$ for the optimal values of Primal PEP.

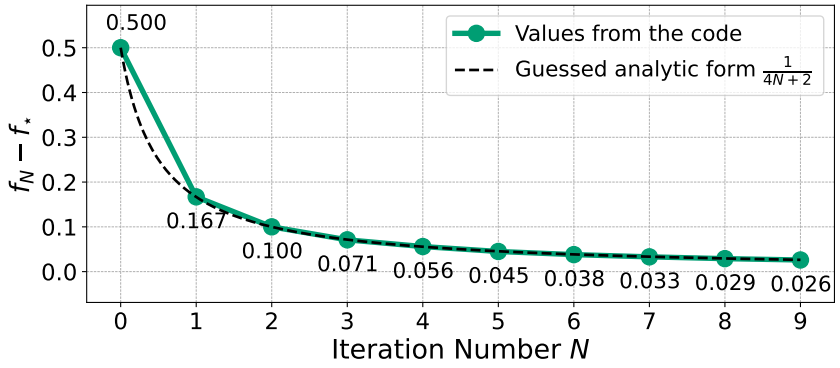


Figure 2: Optimal values of Primal PEP for different N, rounded to three decimal places.

3.3 Lagrangian of Primal PEP and Obtaining a Proof of Convergence

The key to obtaining an analytical proof of convergence is to examine the Lagrangian of Primal PEP:

$$\begin{aligned}\mathcal{L}(F, G, \lambda, \tau, S) = F^\mathsf{T} D_{\star,N} - \sum_{(i,j)\in\mathcal{J}_N^\star} \lambda_{ij}\big(F^\mathsf{T} D_{i,j} + \mathbf{tr}(GB_{i,j}) + \tfrac{1}{2L}\,\mathbf{tr}(GC_{i,j})\big) \\ - \tau\big(\mathbf{tr}(GA_{\star,0}) - \rho^2\big) + \mathbf{tr}(GS),\end{aligned} \tag{5}$$

where $\lambda = \{\lambda_{i,j}\} \subset \mathbb{R}$, $\tau \in \mathbb{R}$, and $S \in \mathbb{S}^{N+3}$ are the Lagrange multipliers (dual variables). Let $(\tilde{\lambda}, \tilde{\tau}, \tilde{S})$ be a dual feasible point. The Lagrangian $\mathcal{L}$ evaluated at this feasible point equals

$$F^\mathsf{T} D_{\star,N} - \tilde{\tau}\,\mathbf{tr}(GA_{\star,0}) = \sum_{(i,j)\in\mathcal{J}_N^\star} \tilde{\lambda}_{ij}\big(F^\mathsf{T} D_{i,j} + \mathbf{tr}(GB_{i,j}) + \tfrac{1}{2L}\,\mathbf{tr}(GC_{i,j})\big) - \mathbf{tr}(G\tilde{S}).$$

Using the definitions we established earlier, this can be rewritten as

$$f_N - f_\star - \tilde{\tau}\|x_0 - x_\star\|_2^2 = \sum_{(i,j)\in\mathcal{J}_N^\star} \tilde{\lambda}_{ij}\Big(f_j - f_i + g_j^\mathsf{T}(x_i - x_j) + \tfrac{1}{2L}\|g_i - g_j\|_2^2\Big) - \mathbf{tr}(G\tilde{S}). \tag{6}$$

Because f is L-smooth and convex, the term $f_j - f_i + g_j^\mathsf{T}(x_i - x_j) + \frac{1}{2L}\|g_i - g_j\|_2^2$ is nonpositive. Moreover, the term $\mathbf{tr}(G\tilde{S})$ is nonnegative as G and $\tilde{S}$ are positive semidefinite. It then follows that

$$f(x_N) - f(x_\star) \le \tilde{\tau}\|x_0 - x_\star\|_2^2.$$

Consequently, if we can find the analytical expression of a dual feasible point $(\tilde{\lambda}, \tilde{\tau}, \tilde{S})$ and verify (6), we can formally establish an upper bound on the worst-case convergence behavior of GD.

3.4 Exact Relaxation of Primal PEP

We first focus on finding analytical expressions for $\tilde{\lambda}$. Since the number of variables $\tilde{\lambda}_{i,j}$ grows quadratically with N, it is desirable to reduce their count. To this end, we attempt to set certain $\tilde{\lambda}_{i,j}$ to zero, or equivalently, to remove some interpolation constraints, without altering the optimal value. In other words, our goal is to identify an equivalent relaxation of Primal PEP.

`PEPFlow` provides an interactive dashboard that enables users to manually deactivate selected interpolation constraints. After choosing which constraints to remove, the user can re-solve the relaxed Primal PEP to check whether the relaxation remains exact. The goal is to identify relaxations where the active constraints exhibit a recognizable pattern as the number of iterations N increases. See Figure 3 for an example illustrating this process for GD.

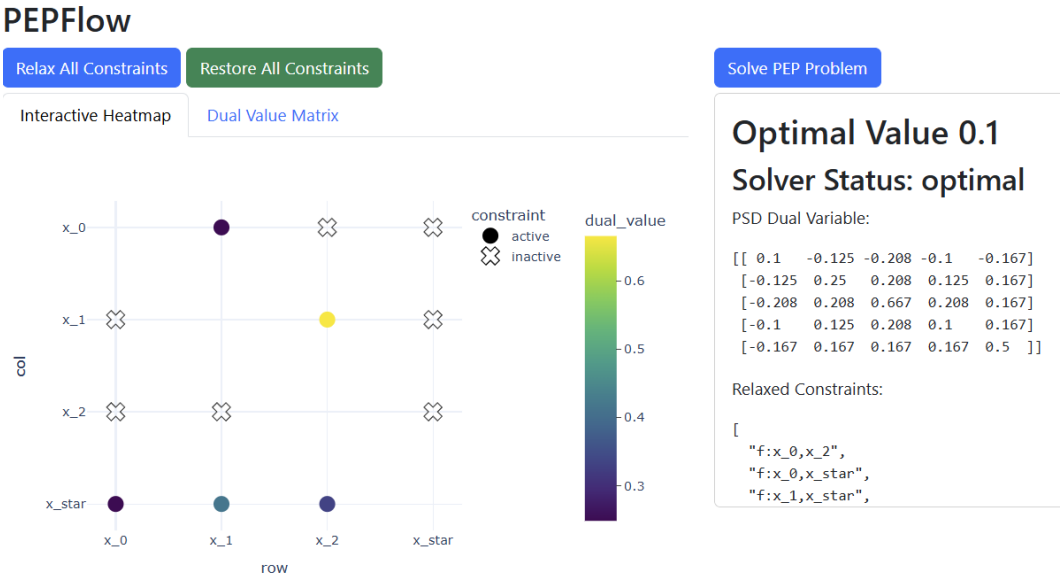


Figure 3: An interactive constraint relaxation panel in `PEPFlow` for GD with $N = 2$.

3.5 Finding and Verifying Analytical Expressions of Dual Variables

After obtaining an exact relaxation of Primal PEP, the next step is to derive and numerically validate analytical expressions for the dual variables. This is carried out through a trial-and-error style iterative process in which the user alternates between proposing candidate formulas for the dual variables and verifying, across different iteration counts N, that these formulas match the solutions returned by the numerical solver. The dual variables fall into three categories, $\tilde{\tau}$, $\tilde{\lambda}$, and $\tilde{S}$, for which analytical expressions must be identified and confirmed. Throughout the remainder of this section, we set $L = 1$ and $\rho = 1$ for simplicity.

`PEPFlow` supports this stage by providing convenient access to solver outputs, utilities to inspect dual variables, and tools to test candidate analytical formulas, thereby streamlining the iterative process for the user.

Finding a closed-form expression for $\tilde{\tau}$ amounts to identifying a pattern in the objective values of Primal PEP as the number of iterations N varies. This was already observed when we numerically verified the convergence rate of GD in Section 3.2, which suggests $\tilde{\tau} = \frac{1}{4N+2}$. The rest of this subsection demonstrates how `PEPFlow` assists users in verifying the other dual variables, $\tilde{\lambda}$ and $\tilde{S}$.

3.5.1 Finding and Verifying Analytical Expressions of $\tilde{\lambda}$

Recall that the dual variables $\tilde{\lambda}$ are associated with the interpolation conditions of Primal PEP, and the purpose of finding a suitable relaxation is to reduce the number of nonzero $\tilde{\lambda}_{i,j}$'s. Here we review the relaxation identified using the interactive dashboard, illustrated for $N = 2$ in Figure 3. This previously known relaxation restricts only the following dual variables to be nonzero:

$$\left\{\tilde{\lambda}_{i-1,i} \mid i = 0, \ldots, N-1\right\} \cup \left\{\tilde{\lambda}_{\star,i} \mid i = 0, \ldots, N\right\}.$$

Examining these dual variables after solving Primal PEP numerically for various N gives:

```
[[0.    0.25  0.    0.   ]
 [0.    0.    0.667 0.   ]
 [0.    0.    0.    0.   ]
 [0.25  0.417 0.333 0.   ]]
```

(a) $N = 2$

```
[[0.    0.167 0.    0.    0.   ]
 [0.    0.    0.4   0.    0.   ]
 [0.    0.    0.    0.75  0.   ]
 [0.    0.    0.    0.    0.   ]
 [0.167 0.233 0.35  0.25  0.   ]]
```

(b) $N = 3$

We can then guess the following analytical expressions:

$$\tilde{\lambda}_{i-1,i} = \frac{i}{2N+1-i},\ i = 1, \dots, N, \qquad \tilde{\lambda}_{\star,i} = \begin{cases} \tilde{\lambda}_{0,1} & i = 0 \\ \tilde{\lambda}_{i,i+1} - \tilde{\lambda}_{i-1,i} & i = 1, \dots, N-1 \\ 1 - \tilde{\lambda}_{i-1,i} & i = N. \end{cases} \tag{7}$$

Details on numerical verification of the proposed expression with `NumPy` are in Appendix B.1.

A key benefit of `PEPFlow` in this stage of the PEP workflow is that it presents the dual variables λ in a structured and transparent manner. As shown in Figure 3, users can readily see the correspondence between the dual variables and their associated interpolation conditions. This clarity is made possible by the tagging system implemented in `PEPFlow`, which allows users to assign tags to different mathematical objects.

3.5.2 Verifying $\tilde{S}$ is positive semidefinite

`PEPFlow` automates the calculations needed for the final numerical verification step: to identify a positive semidefinite (PSD) matrix $\tilde{S}$ that satisfies (6). Actually, given $\tilde{\lambda}$ as in (7) and $\tilde{\tau} = \frac{1}{4N+2}$, $\tilde{S}$ is uniquely determined by dual feasibility. Thus, the PEP workflow is complete once the symbolic $\tilde{S}$ is shown to be PSD. Further details are provided in Appendix B.2.

Although the technical details of this verification are deferred to Appendix B.2, we highlight here two key features of `PEPFlow` that automate much of the process:

- Although $\tilde{S}$ is uniquely determined once $\tilde{\tau}$ and $\tilde{\lambda}$ are fixed, its analytical expression is often intricate and requires careful examination of $\{(x_i, g_i)\}_{i \in \mathcal{I}_N^\star}$ (or $\{(\boldsymbol{x}_i, \boldsymbol{g}_i)\}_{i \in \mathcal{I}_N^\star}$). `PEPFlow` automates the extraction of the pairs $\{(x_i, g_i)\}_{i \in \mathcal{I}_N^\star}$, as demonstrated in the code below.

```
x_list = ctx.tracked_point(f)  # The points associated with function f.
g_x_list = ctx.tracked_grad(f) # The gradients associated with function f.
```

  With the above lists of iterates and gradients, the user can arbitrarily form their linear combinations and inner products, operations essential to verify $\tilde{S}$ is PSD.
- When a user defines a candidate expression, `S_guess`, using linear combinations and inner products of above lists of iterates and gradients, `PEPFlow` provides a function that translates that analytical expression into its matrix representation based on (3) and (4). This otherwise tedious procedure is fully automated and can be carried out with a single line of code:

```
S_guess_matrix_expression = S_guess.eval(ctx).matrix
```

Once we numerically verify an analytical expression of a dual feasible point (for various N), the subsequent symbolic verification is mechanical; an example is provided in Appendix B.3. This completes the end-to-end workflow of PEP.

4 Conclusion and Future Work

This paper introduces `PEPFlow`, a library designed to streamline the end-to-end workflow of the Performance Estimation Problems (PEP) for verifying and deriving convergence guarantees of first-order optimization methods. Beyond formulating and solving the Primal PEP, `PEPFlow` uniquely assists in deriving and verifying analytical convergence proofs. Through the example of gradient descent (GD), we show how its interactive tools, structured access to dual variables, and symbolic routines streamline the transition from numerical results to rigorous analysis.

`PEPFlow` is still under active development, with steady progress toward broader capabilities. Many infrastructure improvements remain—for example, automating the search for suitable relaxations of the Primal PEP—with the ultimate goal of fully "automating" the design and analysis of first-order methods. Beyond infrastructure, the development of `PEPFlow` also sparks new research directions; for instance, studying how patterns in dual solutions can reveal structural properties of algorithms.

References

[1] M. BARRÉ, A. B. TAYLOR, AND F. BACH, *Principled analyses and design of first-order methods with inexact proximal operators*, Mathematical Programming, 201 (2023), pp. 185–230.

[2] N. BOUSSELMI, J. M. HENDRICKX, AND F. GLINEUR, *Interpolation Conditions for Linear Operators and Applications to Performance Estimation Problems*, SIAM Journal on Optimization, 34 (2024), pp. 3033–3063.

[3] S. P. BOYD, T. PARSHAKOVA, E. K. RYU, AND J. J. SUH, *Optimization algorithm design via electric circuits*, Neural Information Processing Systems, (2024).

[4] S. DAS GUPTA, R. M. FREUND, X. A. SUN, AND A. TAYLOR, *Nonlinear conjugate gradient methods: Worst-case convergence rates via computer-assisted analyses*, Mathematical Programming, (2024).

[5] S. DAS GUPTA, B. P. G. VAN PARYS, AND E. K. RYU, *Branch-and-bound performance estimation programming: A unified methodology for constructing optimal optimization methods*, Mathematical Programming, 204 (2024), pp. 567–639.

[6] E. DE KLERK, F. GLINEUR, AND A. B. TAYLOR, *Worst-case convergence analysis of inexact gradient and newton methods through semidefinite programming performance estimation*, SIAM Journal on Optimization, 30 (2020), pp. 2053–2082.

[7] S. DIAMOND AND S. BOYD, *CVXPY: A Python-embedded modeling language for convex optimization*, Journal of Machine Learning Research, 17 (2016), pp. 1–5.

[8] R.-A. DRAGOMIR, A. B. TAYLOR, A. D'ASPREMONT, AND J. BOLTE, *Optimal complexity and certification of Bregman first-order methods*, Mathematical Programming, 194 (2022), pp. 41–83.

[9] Y. DRORI AND M. TEBOULLE, *Performance of first-order methods for smooth convex minimization: A novel approach*, Mathematical Programming, 145 (2014), pp. 451–482.

[10] E. GORBUNOV, N. LOIZOU, AND G. GIDEL, *Extragradient method: $O(1/K)$ last-iterate convergence for monotone variational inequalities and connections with cocoercivity*, International Conference on Artificial Intelligence and Statistics, (2022).

[11] B. GOUJAUD, A. DIEULEVEUT, AND A. TAYLOR, *On fundamental proof structures in first-order optimization*, in Conference on Decision and Control, 2023, pp. 3023–3030.

[12] B. GOUJAUD, C. MOUCER, F. GLINEUR, J. HENDRICKX, A. TAYLOR, AND A. DIEULEVEUT, *PEPit: Computer-assisted worst-case analyses of first-order optimization methods in Python*, Mathematical Programming Computation, 16 (2024), pp. 337–367.

[13] U. JANG, S. D. GUPTA, AND E. K. RYU, *Computer-assisted design of accelerated composite optimization methods: OptISTA*, Mathematical Programming, (2025).

[14] D. KIM, *Accelerated proximal point method for maximally monotone operators*, Mathematical Programming, 190 (2021), pp. 57–87.

[15] D. KIM AND J. A. FESSLER, *Optimized first-order methods for smooth convex minimization*, Mathematical Programming, 159 (2016), pp. 81–107.

[16] D. KIM AND J. A. FESSLER, *Optimizing the efficiency of first-order methods for decreasing the gradient of smooth convex functions*, Journal of Optimization Theory and Applications, 188 (2021), pp. 192–219.

[17] N. KRUCHTEN, A. SEIER, AND C. PARMER, *An interactive, open-source, and browser-based graphing library for Python*, 2024.

[18] S. LEE AND D. KIM, *Fast extra gradient methods for smooth structured nonconvex-nonconcave minimax problems*, Neural Information Processing Systems, (2021).

[19] F. LIEDER, *On the convergence rate of the Halpern-iteration*, Optimization Letters, 15 (2021), pp. 405–418.

[20] C. MOUCER, A. TAYLOR, AND F. BACH, *A systematic approach to Lyapunov analyses of continuous-time models in convex optimization*, SIAM Journal on Optimization, 33 (2023), pp. 1558–1586.

[21] E. D. H. NGUYEN, J. J. SUH, X. JIANG, AND S. MA, *Swapping objectives accelerates Davis-Yin splitting*, arXiv:2506.23475, (2025).

[22] C. PARK AND E. K. RYU, *Optimal first-order algorithms as a function of inequalities*, Journal of Machine Learning Research, 25 (2024), pp. 1–66.

[23] J. PARK AND E. K. RYU, *Exact optimal accelerated complexity for fixed-point iterations*, International Conference on Machine Learning, (2022).

[24] J. PARK AND E. K. RYU, *Accelerated infeasibility detection of constrained optimization and fixed-point iterations*, International Conference on Machine Learning, (2023).

[25] C. PARMER, P. DUVAL, AND A. JOHNSON, *A data and analytics web app framework for Python, no JavaScript required.*, Nov. 2024.

[26] E. K. RYU, A. B. TAYLOR, C. BERGELING, AND P. GISELSSON, *Operator splitting performance estimation: Tight contraction factors and optimal parameter selection*, SIAM Journal on Optimization, 30 (2020), pp. 2251–2271.

[27] F. SCHAIPP, A. HÄGELE, A. TAYLOR, U. SIMSEKLI, AND F. BACH, *The surprising agreement between convex optimization theory and learning-rate scheduling for large model training*, International Conference on Machine Learning, (2025).

[28] A. TAYLOR AND F. BACH, *Stochastic first-order methods: Non-asymptotic and computer-aided analyses via potential functions*, Conference on Learning Theory, (2019).

[29] A. TAYLOR AND Y. DRORI, *An optimal gradient method for smooth strongly convex minimization*, Mathematical Programming, 199 (2023), pp. 557–594.

[30] A. TAYLOR, B. VAN SCOY, AND L. LESSARD, *Lyapunov functions for first-order methods: Tight automated convergence guarantees*, International Conference on Machine Learning, (2018).

[31] A. B. TAYLOR, J. M. HENDRICKX, AND F. GLINEUR, *Exact worst-case performance of first-order methods for composite convex optimization*, SIAM Journal on Optimization, 27 (2017), pp. 1283–1313.

[32] A. B. TAYLOR, J. M. HENDRICKX, AND F. GLINEUR, *Performance estimation toolbox (PESTO): Automated worst-case analysis of first-order optimization methods*, in Conference on Decision and Control, 2017.

[33] A. B. TAYLOR, J. M. HENDRICKX, AND F. GLINEUR, *Smooth strongly convex interpolation and exact worst-case performance of first-order methods*, Mathematical Programming, 161 (2017), pp. 307–345.

[34] M. UPADHYAYA, S. BANERT, A. B. TAYLOR, AND P. GISELSSON, *Automated tight Lyapunov analysis for first-order methods*, Mathematical Programming, (2024).

[35] M. UPADHYAYA, A. B. TAYLOR, S. BANERT, AND P. GISELSSON, *AutoLyap: A Python package for computer-assisted Lyapunov analyses for first-order methods*, (2025).

[36] T. YOON, J. KIM, J. J. SUH, AND E. K. RYU, *Optimal Acceleration for Minimax and Fixed-Point Problems is Not Unique*, International Conference on Machine Learning, (2024).

[37] T. YOON AND E. K. RYU, *Accelerated algorithms for smooth convex-concave minimax problems with $\mathcal{O}(1/k^2)$ rate on squared gradient norm*, International Conference on Machine Learning, (2021).

A Design of PEPFlow library

The PEPFlow library is designed for interactive use. As explained in the main text, the workflow for solving a performance estimation problem (PEP) involves multiple steps. From the user's perspective, interaction with PEPFlow is structured around three main entry points, as depicted in Figure 5.

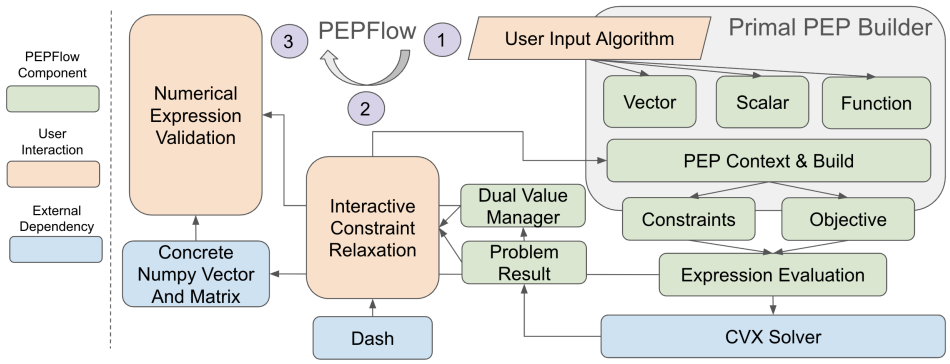


Figure 5: PEPFlow

The first entry point of PEPFlow allows the user to define an optimization algorithm. To model the algorithm, PEPFlow provides three key components: Vector, Scalar, and Function. These serve as abstract representations for quantities defined by the algorithm, such as x_i, $f(x_i)$, or $\nabla f(x_i)$. The relationship between iterates, like $x_{i+1} = x_i - \eta \nabla f(x_i)$, is encoded directly in the Vector object's construction. To encode these relationships, Vector objects support inner product space operations like linear combinations and inner products. Similarly, Scalar objects support basic arithmetic. The Function class has several derived subclasses for different function types (e.g., smooth convex, strongly convex), and each Function object stores the interpolation constraints associated with its type. These three components are designed to interact with one another. For instance, applying a Function to a Vector returns a Scalar, and the inner product of two Vector objects also yields a Scalar.

As these components are abstract representations, their concrete realizations as vectors or matrices are context-dependent. Therefore, we introduce a crucial concept: the PEPContext. The PEPContext is an object that groups all related Vector, Scalar, and Function components, and it is this context that ultimately determines their concrete numerical values. We give a concrete example below:

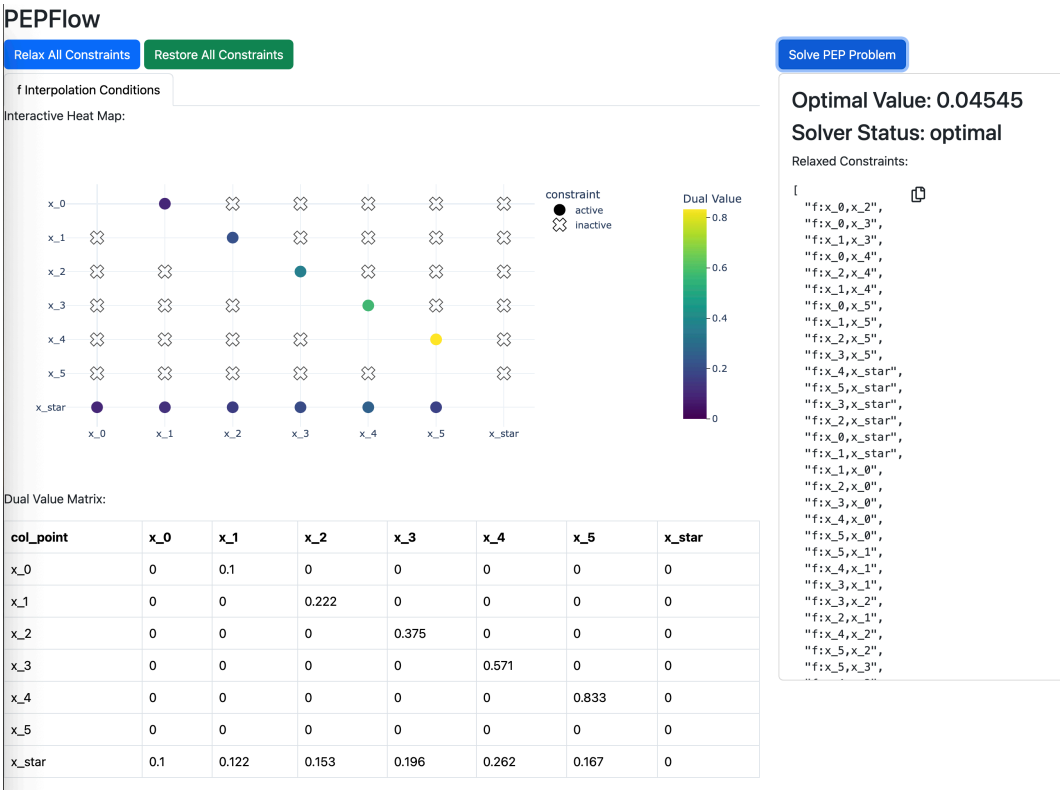
```
>>> import pepflow as pf
>>> ctx = pf.PEPContext("pep").set_as_current()
>>> p1 = pf.Vector(is_basis=True)
>>> print(p1.eval(ctx))
array([1])      # We only have one basis vector in contex
>>> p2 = pf.Vector(is_basis=True)
>>> print(p1.eval(ctx))  # Note it is 'p1' again
array([1, 0])   # Because we have two basis vectors now
```

The PEPContext object automatically tracks all Vector and Function components as they are defined. Therefore, once the user finishes constructing the algorithm, the PEPContext holds all the necessary information to formulate the PEP. This formulation is handled by the PEPBuilder class, which encapsulates everything required to build Primal PEP, including initialization and interpolation constraints, as well as their relaxations. The core of the problem solver is as follow.

```
... # PEPBuilder preparing all performance metric and constraints
solver = ps.CVXSolver(
    objective=objective,
    constraints=[c for c in constraints if c not in relaxed_constraints],
    context=context,
)
problem = solver.build_problem()
result = problem.solve(**kwargs)
...
```

Since PEPFlow is not a solver, it interfaces with external optimization libraries such as CVXPY [7]. This connection is handled by an expression evaluation component. After the PEPBuilder class generates the abstract Constraint and objective objects (depending on PEPContext), the evaluator translates them into the concrete matrix and vector representations that are compatible with CVXPY. That is essentially what the function `solver.build_problem()` does behind the scenes.

While Primal PEP is solvable at this point, this only completes the initial phase of the PEP workflow. The next step involves strategically relaxing the interpolation constraints as much as possible without altering the optimal value. This process requires user intuition and cannot be automated. Therefore, PEPFlow provides an interactive web interface—compatible with both scripts and Jupyter Notebooks—that allows the user to explore and apply these relaxations manually. The following screenshot is the example usage of interactive relaxation:



Dual Value Matrix:

col_point	x_0	x_1	x_2	x_3	x_4	x_5	x_star
x_0	0	0.1	0	0	0	0	0
x_1	0	0	0.222	0	0	0	0
x_2	0	0	0	0.375	0	0	0
x_3	0	0	0	0	0.571	0	0
x_4	0	0	0	0	0	0.833	0
x_5	0	0	0	0	0	0	0
x_star	0.1	0.122	0.153	0.196	0.262	0.167	0

Figure 6: Interactive constraint relaxation panel in PEPFlow.

Users can toggle interpolation constraints between active and inactive by clicking the corresponding points in the left panel. Clicking the "Solve PEP Problem" button then re-solves a relaxation of Primal PEP that only uses the user-selected set of active constraints. By observing changes in the optimal value, the user can assess the impact of toggling each constraint. PEPFlow leverages the Plotly [17] and Dash [25] library to allow for interactive usage. The connection between the Dash app and PEPFlow is through the DualManager. For each Constraint, there is a unique associated name so that Plotly can use the string to manipulate the relaxation of constraints.

After the necessary interpolation constraints are determined, the final stage of the PEP workflow remains—finding and verifying the analytical expressions of the dual variables. Details about how PEPFlow can assist with this process are covered in Section 3.5.

Remark. The PEPFlow library is under active development. Its API and internal utilities are subject to change in future releases.

A.1 Comparison with PEPit

The design of PEPFlow's Primal PEP modeling interface was influenced by PEPit [12], an important precursor library that also models Primal PEP. Despite this shared foundation, their core philosophies differ significantly, as PEPFlow is designed to support the entire interactive workflow, extending beyond PEPit's focus on Primal PEP formulation alone.

PEPFlow is designed for interactive use, featuring a dynamic and extensible interface. This design facilitates straightforward integration with other libraries like Dash and allows users to easily access underlying data, such as NumPy representations, or retrieve information like the dual variables of tagged constraints. PEPit, on the other hand, is a more closed ecosystem, which makes it challenging for users to extract its internal data structures for interoperability with other tools. Furthermore, PEPit is designed for static script execution and lacks context management. As a result, when used in an interactive environment like a Jupyter Notebook, its global state can be easily corrupted by the nonlinear execution of cells. Consequently, this design makes it challenging to extend PEPit to support interactive constraint relaxation, a feature we consider essential to the PEP workflow.

B Omitted details in Section 3.5

B.1 Details of the implementation to verify analytical expressions of $\tilde{\lambda}$

To numerically verify whether our proposed analytical expression is correct, we can use, for example, the following code snippet:

```
lambda_candidate = np.zeros((N + 2, N + 2))

## Additional constraint 1 (consecutive)
for j in range(N + 1):
    for i in range(N + 2):
        if j == i + 1:
            lambda_candidate[i, j] = j / (2 * N + 1 - j)

## Additional constraint 2 (between x_*)
for j in range(N + 1):
    if j == N:
        lambda_candidate[N + 1, j] = 1 - lambda_candidate[j - 1, j]
    else:
        lambda_candidate[N + 1, j] = (
            lambda_candidate[j, j + 1] - lambda_candidate[j - 1, j]
        )

np.allclose(lambda_candidate, lambda_solution, atol=1e-4)
```

B.2 Details of the implementation for numerically verifying that $\tilde{S}$ is PSD

Recall that our goal is to find feasible $\tilde{\lambda}$ and $\tilde{S}$ that satisfy (6), with $\tilde{\tau} = \frac{1}{4N+2}$. Since we have a candidate for $\tilde{\lambda}$, the remaining step is to find a compatible analytic expression for a feasible $\tilde{S}$. Since $\tilde{S}$ must satisfy (6), we can show that it should satisfy[1]

$$\tilde{S} = \sum_{i,j \in \mathcal{I}_N^\star} \tilde{\lambda}_{ij} \left(B_{ij} + \frac{1}{2L} C_{i,j} \right) + \tilde{\tau} A_{\star,0}. \tag{8}$$

Thus, the analytical form is already determined since $\tilde{\tau}$ and $\tilde{\lambda}$ are determined. The remaining condition from dual feasibility that we need to show is that $\tilde{S}$ is PSD.

There are various way to show a matrix is PSD. Here, we show that $\tilde{S}$ is PSD by providing a concrete decomposition. We first print the numerical values of $\tilde{S}$ we obtain from the code:

[1] We use the fact that, since the equality (6) holds for all arbitrary G, the coefficients of G on the left-hand side and the right-hand side must coincide.

```
S_solution =
[[ 0.1    -0.125 -0.208 -0.167 -0.1  ]
 [-0.125   0.25   0.208  0.167  0.125]
 [-0.208   0.208  0.667  0.167  0.208]
 [-0.167   0.167  0.167  0.5    0.167]
 [-0.1     0.125  0.208  0.167  0.1  ]]
```

Figure 7: Numerical values of $\tilde{S}$ obtained from the code, when $N = 2$.

Leveraging the numerical values and using some heuristics, we are able to find the following decomposition:

$$S_{\text{guess}} = \frac{1}{4N+2}(\boldsymbol{z}_N - \boldsymbol{x}_\star) \odot (\boldsymbol{z}_N - \boldsymbol{x}_\star) + \frac{1}{2}\sum_{i=0}^{N}\sum_{j=i+1}^{N}\left((2N+1)\tilde{\lambda}_{\star,i} - 1\right)\tilde{\lambda}_{\star,j}C_{i,j}, \tag{9}$$

where

$$\boldsymbol{z}_N = \boldsymbol{x}_0 - (2N+1)\sum_{i=0}^{N}\tilde{\lambda}_{\star,i}\boldsymbol{g}_i.$$

We numerically verify this decomposition with the following code. As mentioned in Section 3.5.2, we leverage `PEPFlow` to automatically obtain the list corresponding to $\{x_i\}_{i\in\mathcal{I}_N^\star}$ and $\{g_i\}_{i\in\mathcal{I}_N^\star}$.

```
x_list = ctx.tracked_point(f)  # The points associated with function f.
g_x_list = ctx.tracked_grad(f) # The gradients associated with function f.
```

Now, we create the term corresponding to (9). First, we calculate the term $\frac{1}{4N+2}(\boldsymbol{z}_N - \boldsymbol{x}_\star) \odot (\boldsymbol{z}_N - \boldsymbol{x}_\star)$ in (9). Leveraging `PEPFlow`, the required calculation can be written intuitively, as shown below.

```
z = x_list[0]- (2*N+1) * sum(lambda_candidate[N+1,i] * g_x_list[i] for i in range(N + 1))
S_guess = 1 / (4*N+2) * (z - x_list[-1])**2
```

Similarly, we calculate the remaining summation term in (9).

```
S_guess += 1/2 * sum(lambda_candidate[N+1,j] * ((2*N+1) * lambda_candidate[N+1,i] - 1)
        * (g_x_list[i] - g_x_list[j])**2 for i in range(N+1) for j in range(i+1,N+1) )
```

As mentioned in Section 3.5.2, `PEPFlow` provides a function that translates the analytical expression `S_guess` into its matrix representation `S_guess_matrix_expression`, which is based on (3) and (4). This translation allows us to compare our guessed decomposition `S_guess` with the numerical values in Figure 7, which are the values of $\tilde{S}$ obtained from the code.

```
S_guess_matrix_expression = S_guess.eval(ctx).matrix
```

Finally, we check that our candidate decomposition is equal to $\tilde{S}$ obtained from the code. The following code outputs `True`. This concludes the numerical verification of the decomposition in (9).

```
np.allclose(S_guess_matrix_expression, S_solution, atol=1e-4)
```

We could still obtain `True` for $N = 0, \ldots, 10$, which strongly suggests that the decomposition in (9) is correct. Once we have a convincing candidate, the remaining formal verification can be done by carefully comparing the coefficients, which is an elementary calculation. Details are provided in Appendix B.3.

B.3 Details of symbolically verifying that $\tilde{S}$ is PSD

In Appendix B.2, we numerically verified that $\tilde{S}$ in (8) and S_{guess} in (9) are equal. Recalling that $\tilde{\tau} = \frac{1}{4N+2}$, this means that the following equality holds for $N = 0, \ldots, 10$, up to a certain numerical precision:

$$\begin{aligned}
&\sum_{i,j\in\mathcal{I}_N^\star}\tilde{\lambda}_{ij}\left(B_{ij} + \frac{1}{2}C_{i,j}\right) + \tilde{\tau}A_{\star,0} \\
&= \tilde{\tau}(\boldsymbol{z}_N - \boldsymbol{x}_\star) \odot (\boldsymbol{z}_N - \boldsymbol{x}_\star) + \frac{1}{2}\sum_{i=0}^{N}\sum_{j=i+1}^{N}\left(\frac{1}{2\tilde{\tau}}\tilde{\lambda}_{\star,i} - 1\right)\tilde{\lambda}_{\star,j}C_{i,j}
\end{aligned}$$

Recalling the definition (4), this suggests (10) in the lemma below. This is essentially an alternative argument corresponding to half of the proof of [9, Theorem 3.1], which relies on [9, Lemma 3.2] and [9, Lemma 3.3], and spans more than seven pages.

Lemma 1. *Let $N \geq 0$. Define $\tilde{\lambda}_{i,j}$ for $i, j \in \mathcal{I}_N^\star = \{0, 1, \ldots, N, \star\}$ as in (7), i.e.,*

$$\tilde{\lambda}_{i-1,i} = \frac{i}{2N+1-i},\ i = 1, \ldots, N, \qquad \tilde{\lambda}_{\star,i} = \begin{cases} \tilde{\lambda}_{0,1} & i = 0 \\ \tilde{\lambda}_{i,i+1} - \tilde{\lambda}_{i-1,i} & i = 1, \ldots, N-1 \\ 1 - \tilde{\lambda}_{i-1,i} & i = N. \end{cases} \tag{7}$$

when $j = i-1$ or $i = \star$, and 0 otherwise. Suppose that $\{x_i\}_{i=0,1,\ldots,N}$ satisfies $x_i = x_0 - \sum_{j=0}^{i-1} g_j$ and $g_\star = 0$. Then the following equality holds:

$$\begin{aligned} &\sum_{i,j \in \mathcal{I}_N^\star} \tilde{\lambda}_{ij} \left(g_j^\mathsf{T}(x_i - x_j) + \frac{1}{2} \|g_i - g_j\|^2 \right) + \tilde{\tau} \|x_0 - x_\star\|^2 \\ &= \tilde{\tau} \left\| x_0 - x_\star - \frac{1}{2\tilde{\tau}} \sum_{i=0}^{N} \tilde{\lambda}_{\star,i} g_i \right\|^2 + \frac{1}{2} \sum_{i=0}^{N} \sum_{j=i+1}^{N} \left(\frac{1}{2\tilde{\tau}} \tilde{\lambda}_{\star,i} - 1 \right) \tilde{\lambda}_{\star,j} \|g_i - g_j\|^2 . \end{aligned} \tag{10}$$

Proof. Although the statement appears rather complicated to prove, the underlying strategy is straightforward:

substitute all definitions and assumptions into (10) and
compare the coefficients on both sides via a careful case division of the indices.

The remaining proof is simply a detailed breakdown of the above process to make it easier for humans to follow.

By substituting the definitions and assumptions, showing (10) reduces to showing the equality below:

$$\begin{aligned} &\sum_{i=1}^{N} \tilde{\lambda}_{i-1,i} \left(g_i^\mathsf{T} g_{i-1} + \frac{1}{2} \|g_{i-1} - g_i\|^2 \right) + \tilde{\tau} \|x_0 - x_\star\|^2 \\ &\quad + \sum_{i=0}^{N} \tilde{\lambda}_{\star,i} \left(g_i^\mathsf{T} \left(x_\star - x_0 + \sum_{j=0}^{i-1} g_j \right) + \frac{1}{2} \|g_i\|^2 \right) \\ &= \tilde{\tau} \left\| x_0 - x_\star - \frac{1}{2\tilde{\tau}} \sum_{i=0}^{N} \tilde{\lambda}_{\star,i} g_i \right\|^2 + \frac{1}{2} \sum_{i=0}^{N} \sum_{j=i+1}^{N} \left(\frac{1}{2\tilde{\tau}} \tilde{\lambda}_{\star,i} - 1 \right) \tilde{\lambda}_{\star,j} \|g_i - g_j\|^2 \end{aligned} \tag{11}$$

This equality can be verified in the following steps:

(i) Compare the terms containing $x_0 - x_\star$.

$$\begin{aligned} &\sum_{i=1}^{N} \tilde{\lambda}_{i-1,i} \left(g_i^\mathsf{T} g_{i-1} + \frac{1}{2} \|g_{i-1} - g_i\|^2 \right) + \sum_{i=0}^{N} \tilde{\lambda}_{\star,i} \left(g_i^\mathsf{T} \sum_{j=0}^{i-1} g_j + \frac{1}{2} \|g_i\|^2 \right) \\ &= \frac{1}{4\tilde{\tau}} \left\| \sum_{i=0}^{N} \tilde{\lambda}_{\star,i} g_i \right\|^2 + \frac{1}{2} \sum_{i=0}^{N} \sum_{j=i+1}^{N} \left(\frac{1}{2\tilde{\tau}} \tilde{\lambda}_{\star,i} - 1 \right) \tilde{\lambda}_{\star,j} \|g_i - g_j\|^2 \end{aligned} \tag{12}$$

(ii) Compare the coefficients of the cross terms $g_i^\mathsf{T} g_j$.

Observe that the cross terms in the right hand side can be simplified to:

$$\begin{aligned}
&\frac{1}{2\tilde{\tau}}\sum_{i=0}^{N}\sum_{j=0}^{i-1}\tilde{\lambda}_{\star,i}\tilde{\lambda}_{\star,j}g_i^{\mathsf{T}}g_j-\sum_{i=0}^{N}\sum_{j=i+1}^{N}\left(\frac{1}{2\tilde{\tau}}\tilde{\lambda}_{\star,i}-1\right)\tilde{\lambda}_{\star,j}g_i^{\mathsf{T}}g_j\\
&=\frac{1}{2\tilde{\tau}}\left(\sum_{i=0}^{N}\sum_{j=0}^{i-1}\tilde{\lambda}_{\star,i}\tilde{\lambda}_{\star,j}g_i^{\mathsf{T}}g_j-\sum_{i=0}^{N}\sum_{j=i+1}^{N}\tilde{\lambda}_{\star,i}\tilde{\lambda}_{\star,j}g_i^{\mathsf{T}}g_j\right)+\sum_{i=0}^{N}\sum_{j=i+1}^{N}\tilde{\lambda}_{\star,j}g_i^{\mathsf{T}}g_j\\
&=\sum_{i=0}^{N}\sum_{j=0}^{i-1}\tilde{\lambda}_{\star,i}g_i^{\mathsf{T}}g_j.
\end{aligned}$$

Substituting into (12), we obtain

$$\begin{aligned}
&\sum_{i=1}^{N}\tilde{\lambda}_{i-1,i}\left(\frac{1}{2}\left\|g_{i-1}\right\|^2+\frac{1}{2}\left\|g_i\right\|^2\right)+\sum_{i=0}^{N}\tilde{\lambda}_{\star,i}\left(\frac{1}{2}\left\|g_i\right\|^2\right)\\
&=\frac{1}{4\tilde{\tau}}\sum_{i=0}^{N}\tilde{\lambda}_{\star,i}^2\left\|g_i\right\|^2+\frac{1}{2}\sum_{i=0}^{N}\sum_{j=i+1}^{N}\left(\frac{1}{2\tilde{\tau}}\tilde{\lambda}_{\star,i}-1\right)\tilde{\lambda}_{\star,j}\left(\left\|g_i\right\|^2+\left\|g_j\right\|^2\right)
\end{aligned}\tag{13}$$

(iii) Compare the coefficients of the squared terms $\|g_i\|^2$.

Now, we take a look at the coefficient of $\|g_i\|^2$ for each $i=0,\ldots,N$. By reorganizing the left-hand side, we obtain:

$$\begin{aligned}
&\sum_{i=1}^{N}\tilde{\lambda}_{i-1,i}\left(\frac{1}{2}\left\|g_{i-1}\right\|^2+\frac{1}{2}\left\|g_i\right\|^2\right)+\sum_{i=0}^{N}\tilde{\lambda}_{\star,i}\left(\frac{1}{2}\left\|g_i\right\|^2\right)\\
&=\frac{1}{2}\sum_{i=0}^{N-1}\tilde{\lambda}_{i,i+1}\left\|g_i\right\|^2+\frac{1}{2}\sum_{i=1}^{N}\tilde{\lambda}_{i-1,i}\left\|g_i\right\|^2+\frac{1}{2}\sum_{i=0}^{N}\tilde{\lambda}_{\star,i}\left\|g_i\right\|^2\\
&=\frac{1}{2}\left(\tilde{\lambda}_{0,1}+\tilde{\lambda}_{\star,0}\right)\|g_0\|^2+\frac{1}{2}\sum_{i=1}^{N-1}\left(\tilde{\lambda}_{i,i+1}+\tilde{\lambda}_{i-1,i}+\tilde{\lambda}_{\star,i}\right)\|g_i\|^2+\frac{1}{2}\left(\tilde{\lambda}_{N-1,N}+\tilde{\lambda}_{\star,N}\right)\|g_N\|^2\\
&=\sum_{i=0}^{N-1}\tilde{\lambda}_{i,i+1}\left\|g_i\right\|^2+\frac{1}{2}\left\|g_N\right\|^2,
\end{aligned}\tag{14}$$

where we obtain the last equation from the definition of $\tilde{\lambda}_{\star,i}$ in (7).

By reorganizing the double summation on the right-hand side, we obtain:

$$\begin{aligned}
&\frac{1}{2}\sum_{i=0}^{N}\sum_{j=i+1}^{N}\left(\frac{1}{2\tilde{\tau}}\tilde{\lambda}_{\star,i}-1\right)\tilde{\lambda}_{\star,j}\left(\left\|g_i\right\|^2+\left\|g_j\right\|^2\right)\\
&=\frac{1}{2}\sum_{i=0}^{N}\sum_{j=i+1}^{N}\left(\frac{1}{2\tilde{\tau}}\tilde{\lambda}_{\star,i}-1\right)\tilde{\lambda}_{\star,j}\left\|g_i\right\|^2+\frac{1}{2}\sum_{i=0}^{N}\sum_{j=i+1}^{N}\left(\frac{1}{2\tilde{\tau}}\tilde{\lambda}_{\star,i}-1\right)\tilde{\lambda}_{\star,j}\left\|g_j\right\|^2\\
&=\frac{1}{2}\sum_{i=0}^{N}\sum_{j=i+1}^{N}\left(\frac{1}{2\tilde{\tau}}\tilde{\lambda}_{\star,i}-1\right)\tilde{\lambda}_{\star,j}\left\|g_i\right\|^2+\frac{1}{2}\sum_{i=1}^{N}\sum_{j=0}^{i-1}\left(\frac{1}{2\tilde{\tau}}\tilde{\lambda}_{\star,j}-1\right)\tilde{\lambda}_{\star,i}\left\|g_i\right\|^2\\
&=\frac{1}{2}\left(\frac{1}{2\tilde{\tau}}\tilde{\lambda}_{\star,0}-1\right)\sum_{j=1}^{N}\tilde{\lambda}_{\star,j}\left\|g_0\right\|^2+\frac{1}{2}\sum_{i=1}^{N}\left(\sum_{j=0}^{N}\frac{1}{2\tilde{\tau}}\tilde{\lambda}_{\star,i}\tilde{\lambda}_{\star,j}-\sum_{j=i+1}^{N}\tilde{\lambda}_{\star,j}-\sum_{j=0}^{i-1}\tilde{\lambda}_{\star,i}\right)\left\|g_i\right\|^2\\
&\quad-\frac{1}{2}\sum_{i=1}^{N}\frac{1}{2\tilde{\tau}}\tilde{\lambda}_{\star,i}^2\left\|g_i\right\|^2,
\end{aligned}$$

where we obtain the second equality by rearranging the order of summation.[2]

[2] $\sum_{i=0}^{N}\sum_{j=i+1}^{N}a_{i,j}=\sum_{j=0}^{N}\sum_{i=j+1}^{N}a_{j,i}=\sum_{i=1}^{N}\sum_{j=0}^{i-1}a_{j,i}$

Next, from the definition of $\tilde{\lambda}_{\star,i}$ in (7), it follows that:

$$\sum_{j=0}^{i} \tilde{\lambda}_{\star,j} = \begin{cases} \tilde{\lambda}_{i,i+1} & \text{if } i = 0, \dots, N-1 \\ 1 & \text{if } i = N, \end{cases}$$

and so

$$\sum_{j=i+1}^{N} \tilde{\lambda}_{\star,j} = \sum_{j=0}^{N} \tilde{\lambda}_{\star,j} - \sum_{j=0}^{i} \tilde{\lambda}_{\star,j} = 1 - \tilde{\lambda}_{i,i+1}.$$

Substituting the gathered observations into (13), we can check that the right-hand side of (13) can be rewritten as:

$$\begin{aligned} &\frac{1}{4\tilde{\tau}} \sum_{i=0}^{N} \tilde{\lambda}_{\star,i}^2 \left\|g_i\right\|^2 + \frac{1}{2} \sum_{i=0}^{N} \sum_{j=i+1}^{N} \left(\frac{1}{2\tilde{\tau}} \tilde{\lambda}_{\star,i} - 1\right) \tilde{\lambda}_{\star,j} \left(\left\|g_i\right\|^2 + \left\|g_j\right\|^2\right) \\ &= \left(\frac{1}{2}\left(\frac{1}{2\tilde{\tau}} \tilde{\lambda}_{\star,0} - 1\right)(1 - \tilde{\lambda}_{\star,0}) + \frac{1}{4\tilde{\tau}} \tilde{\lambda}_{\star,0}^2\right) \left\|g_0\right\|^2 \\ &\quad + \sum_{i=1}^{N} \frac{1}{2}\left(\frac{1}{2\tilde{\tau}} \tilde{\lambda}_{\star,i} - (1 - \tilde{\lambda}_{i,i+1}) - i\tilde{\lambda}_{\star,i}\right) \left\|g_i\right\|^2 + \frac{1}{2}\left(\frac{1}{2\tilde{\tau}} \tilde{\lambda}_{\star,N} - N\tilde{\lambda}_{\star,N}\right) \left\|g_N\right\|^2. \end{aligned}$$

Now, recalling that $\tilde{\tau} = \frac{1}{4N+2}$ and $\tilde{\lambda}_{\star,0} = \tilde{\lambda}_{0,1} = \frac{1}{2N}$ from (7), for $i = 0$ we have:

$$\frac{1}{2}\left(\frac{1}{2\tilde{\tau}} \tilde{\lambda}_{\star,0} - 1\right)(1 - \tilde{\lambda}_{\star,0}) + \frac{1}{4\tilde{\tau}} \tilde{\lambda}_{\star,0}^2 = \frac{1}{2}(2N + 1 + 1)\, \tilde{\lambda}_{\star,0} - \frac{1}{2} = \tilde{\lambda}_{0,1}.$$

Next, recalling that $\tilde{\lambda}_{i-1,i} = \frac{i}{2N+1-i}$ from (7), for $i = 1, \dots, N-1$ we have:

$$\begin{aligned} &\frac{1}{2}\left(\frac{1}{2\tilde{\tau}} \tilde{\lambda}_{\star,i} - (1 - \tilde{\lambda}_{i,i+1}) - i\tilde{\lambda}_{\star,i}\right) \\ &= \frac{1}{2}(2N + 1 - i)\left(\tilde{\lambda}_{i,i+1} - \tilde{\lambda}_{i-1,i}\right) - \frac{1}{2}(1 - \tilde{\lambda}_{i,i+1}) \\ &= \frac{1}{2}\underbrace{(2N - i)\tilde{\lambda}_{i,i+1}}_{=i+1} + \frac{1}{2}\tilde{\lambda}_{i,i+1} - \frac{1}{2}\underbrace{(2N + 1 - i)\tilde{\lambda}_{i-1,i}}_{=i} - \frac{1}{2} + \frac{1}{2}\tilde{\lambda}_{i,i+1} = \tilde{\lambda}_{i,i+1}. \end{aligned}$$

Finally, for $i = N$:

$$\frac{1}{2}\left(\frac{1}{2\tilde{\tau}} \tilde{\lambda}_{\star,N} - N\tilde{\lambda}_{\star,N}\right) = \frac{1}{2}(2N + 1 - N)\, \tilde{\lambda}_{\star,N} = \frac{1}{2}(N + 1)\left(1 - \frac{N}{N+1}\right) = \frac{1}{2}.$$

Since this coincides with the coefficient on the left-hand side in (14), we conclude the proof. □

This completes the PEP workflow.