```
  Activating project at `~/.julia/environments/v1.12`
Precompiling packages...
   1021.1 ms  ✓ PlutoUI
  1 dependency successfully precompiled in 1 seconds. 34 already precompiled.
```

`PEPit.jl`: computer-assisted worst-case analysis of first-order optimization algorithms in Julia

Shuvomoy Das Gupta

Joint work with Python PEPit team: B. Goujaud, C. Moucer, F. Glineur, J. Hendrickx, A. Taylor, A. Dieuleveut.

JuMP-dev 2026

JuMP-dev 2026, Edinburgh

Some personal update

Since the last time we met at JuMP-dev 2023:

- I finished my PhD in 2024.
- I was a postdoc at Columbia IEOR during 2024-2025.
- In Fall 2025, I started a faculty position at Rice University in Houston, TX.

What is `PEPit.jl`?

- `PEPit.jl` is a native Julia implementation of the Performance Estimation Programming (PEP) methodology [Drori–Teboulle 2014; Taylor–Hendrickx–Glineur 2017a,b].
- Functionally, `PEPit.jl` is equivalent to the Python package `PEPit` [Goujaud et al. 2024].
- **The core idea in PEP.** Model the analysis of first-order optimization algorithms as higher-level optimization problems called performance estimation problems (PEPs).
    - These PEPs are convex semidefinite programs (SDPs)!
    - We solve these SDPs numerically to obtain tight worst-case bounds for known algorithms.
    - We can also use PEP to discover new algorithms under suitable conditions!

Why `PEPit.jl`?

- The `Julia` + `JuMP` ecosystem is very conducive to open-source reproducible research.
- `JuMP` supports 50+ commercial and open-source solvers.
    - PEP can benefit from interaction between different types of solvers.
- `Julia` is a fast language, `JuMP` provides solver-independent callbacks (lazy constraints, user-cuts, heuristic solutions).
    - PEP often requires exploiting problem-specific structures to extract useful information.
    - `Julia` + `JuMP` can allow us to write custom code to exploit structures present in PEPs.
- `JuMP` already has many useful packages, such as `Dualization.jl`, `ParametricOptInterface.jl`, and `BilevelJuMP.jl`, that can potentially exploit problem-specific structure in PEPs.
- Certain types of PEPs can be naturally nonlinear/nonconvex: `Julia` + `JuMP` can be a natural fit for these types of PEPs.

PEP workflow

A performance estimation problem (PEP) turns the worst-case convergence bound of an optimization algorithm into a convex optimization problem.

Core PEP workflow

1. **Input.** Specify the mathematical setting (input to `PEPit.jl`):
    - function class, e.g. $f \in \mathcal{F}_{\mu,L}$,
    - algorithm, e.g. $x_{i+1} = x_i - \alpha \nabla f(x_i)$,
    - performance metric, e.g. $\|x_i - x_\star\|^2$,
    - initialization, e.g. $\|x_0 - x_\star\|^2 \leq 1$.

2. **Sample.** Sample only the objects experienced by the method:
    - points x_i,
    - gradients g_i,
    - function values f_i.
3. **Interpolation.** Replace the unknown infinite-dimensional function/operator by *interpolation* constraints.
4. **Gramian transformation.** Lift inner products and norm-squared terms into a Gram matrix: $G_{ij} = \langle v_i, v_j \rangle$ and $G \succeq 0$. Function values are embedded in vector $F = [\{f_i\}_i]$.
5. **Solve.** Solve the resulting SDP!

A simple example

- **Optimization problem.** Suppose we are interested in solving $\min_x f(x)$, where $f : \mathbb{R}^d \to \mathbb{R}$.
- **Function type.** We consider $f \in \mathcal{F}_{\mu,L}$.
    - $\mathcal{F}_{\mu,L}$: the class of L-smooth and μ-strongly convex functions (functions that are not too flat or sharp)
- **Algorithm to be studied.** The algorithm is gradient descent:

$$x_{i+1} = x_i - \alpha \nabla f(x_i),\ i = 0, 1, 2, \dots, \quad \text{(GD)}$$

- We start with some x_0 and then update the iterates using (GD).

Main question

- **Question. [Taylor 2024]** What is the smallest τ such that

$$\frac{\|x_{i+1} - x_\star\|^2}{\|x_i - x_\star\|^2} \leq \tau$$

for every dimension d, every $f \in \mathcal{F}_{\mu,L}$, every starting point x_0, and the gradient step $x_{i+1} = x_i - \alpha \nabla f(x_i)$?

- Note that, without loss of generality, we can set $i \leftarrow 0$.

Abstract optimization problem

- Find the smallest τ such that

$$\|x_1 - x_\star\|^2 \leq \tau \|x_0 - x_\star\|^2$$

for every dimension $d \in \mathbb{N}$, every L-smooth and μ-strongly convex function $f \in \mathcal{F}_{\mu,L}$, every starting point x_0, the algorithmic step $x_1 = x_0 - \alpha \nabla f(x_0)$, and $x_\star \in \arg\min_x f(x)$.

- The problem can be written as the *abstract* optimization problem:

$$\begin{aligned} \tau = \text{maximize}_{f,x_0,x_1,x_\star,d} \quad & \frac{\|x_1 - x_\star\|^2}{\|x_0 - x_\star\|^2} \\ \text{subject to} \quad & f \in \mathcal{F}_{\mu,L}, \\ & x_1 = x_0 - \alpha \nabla f(x_0), \\ & \nabla f(x_\star) = 0. \end{aligned}$$

Variables: $f, x_0, x_1, x_\star, d$. Parameters: μ, L, α.

Sampled version

- **Abstract problem.** The original PEP has an infinite-dimensional function as a variable:

$$\begin{aligned} \text{maximize}_{f,x_0,x_1,x_\star,d} \quad & \frac{\|x_1 - x_\star\|^2}{\|x_0 - x_\star\|^2} \\ \text{subject to} \quad & f \text{ is } L\text{-smooth and } \mu\text{-strongly convex}, \\ & x_1 = x_0 - \alpha \nabla f(x_0), \\ & \nabla f(x_\star) = 0. \end{aligned}$$

- **Sampled problem.** The sampled version replaces f by finitely many unknown samples at x_0 and $x_\star$:

$$\begin{aligned} \text{maximize}_{\substack{x_0,x_1,x_\star \\ g_0,g_\star \\ f_0,f_\star,d}} \quad & \frac{\|x_1 - x_\star\|^2}{\|x_0 - x_\star\|^2} \\ \text{subject to} \quad & \exists f \in \mathcal{F}_{\mu,L} \text{ such that } \begin{cases} f_i = f(x_i), & i = 0, \star, \\ g_i = \nabla f(x_i), & i = 0, \star, \end{cases} \\ & x_1 = x_0 - \alpha g_0, \\ & g_\star = 0. \end{aligned}$$

New variables: $x_0, x_1, x_\star, g_0, g_\star, f_0, f_\star, d$.

Concept of interpolation

- **Setup.** Let I be an index set and suppose we are given samples

$$\{(x_i, g_i, f_i)\}_{i \in I},$$

where x_i are points, g_i are gradients, and f_i are function values.

- **Interpolation question.**

$$\text{does there exist } f \in \mathcal{F}_{\mu,L} \text{ such that } f(x_i) = f_i, \quad \nabla f(x_i) = g_i \quad \forall i \in I?$$

- **Interpolation condition.** A necessary and sufficient condition is that, for all $i, j \in I$,

$$f_i \geq f_j + \langle g_j, x_i - x_j \rangle + \frac{1}{2L}\|g_i - g_j\|^2 + \frac{\mu}{2(1-\mu/L)}\left\|x_i - x_j - \frac{1}{L}(g_i - g_j)\right\|^2.$$

Existence constraint $\leftrightarrow$ interpolation constraint

- The sampled PEP contains the existence constraint

$$\exists f \in \mathcal{F}_{\mu,L} \text{ such that } \begin{cases} f_i = f(x_i), & i = 0, \star, \\ g_i = \nabla f(x_i), & i = 0, \star. \end{cases}$$

- Interpolation allows us to replace the existence constraint by explicit inequalities between the two sampled points:

$$f_\star \geq f_0 + \langle g_0, x_\star - x_0 \rangle + \frac{1}{2L}\|g_\star - g_0\|^2 + \frac{\mu}{2(1-\mu/L)}\left\|x_\star - x_0 - \frac{1}{L}(g_\star - g_0)\right\|^2,$$
$$f_0 \geq f_\star + \langle g_\star, x_0 - x_\star \rangle + \frac{1}{2L}\|g_0 - g_\star\|^2 + \frac{\mu}{2(1-\mu/L)}\left\|x_0 - x_\star - \frac{1}{L}(g_0 - g_\star)\right\|^2.$$

- With $x_1 = x_0 - \alpha g_0$ and $g_\star = 0$, this has the same optimal value as the sampled PEP: no relaxation has happened yet.
- What changed is the form of the problem: it is now a finite but *nonconvex* QCQP.

Semidefinite lifting

- **Key observation.** The problem now contains only scalar function values and inner products/ norm squares involving the iterates and gradients in $\mathbb{R}^d$.
- **Gram matrix.** Define

$$P \triangleq [x_0 - x_\star,\ g_0] \in \mathbb{R}^{d\times 2}, \qquad F \triangleq f_0 - f_\star.$$

- The Gram matrix

$$G \triangleq P^T P = \begin{bmatrix} \|x_0 - x_\star\|^2 & \langle g_0, x_0 - x_\star \rangle \\ \langle g_0, x_0 - x_\star \rangle & \|g_0\|^2 \end{bmatrix} \succeq 0, \quad \operatorname{rank} G \leq d$$

linearizes all squared norms and inner products!

- **Large-scale assumption.** For $d \geq 2$, $\operatorname{rank} G \leq 2$, so the rank constraint is satisfied automatically.
- **Final SDP.** Substituting x_1 and $g_\star$, and normalizing $G_{1,1} = 1$ gives the 2×2 SDP.

$$\begin{aligned} \text{maximize}_{G,F} \quad & G_{1,1} + \alpha^2 G_{2,2} - 2\alpha G_{1,2} \\ \text{subject to} \quad & F + \frac{L\mu}{2(L-\mu)} G_{1,1} + \frac{1}{2(L-\mu)} G_{2,2} - \frac{L}{L-\mu} G_{1,2} \leq 0, \\ & -F + \frac{L\mu}{2(L-\mu)} G_{1,1} + \frac{1}{2(L-\mu)} G_{2,2} - \frac{\mu}{L-\mu} G_{1,2} \leq 0, \\ & G_{1,1} = 1, \qquad G \succeq 0. \end{aligned}$$

PEP reformulation recap

- The workflow is consistent across other setups as well.
- Computation of the worst-case convergence bound is represented by:
  - vector F containing function values f_i,
  - a Gram matrix G encoding inner products between points and gradients,
  - linear constraints encoding interpolation and initialization,
  - a linear objective encoding the performance metric.
- The formulated SDP will be of the form:

$$\begin{aligned} \text{maximize}_{G,F} \quad & \text{linear function of } (F, G) \\ \text{subject to} \quad & G \succeq 0, \\ & \text{linear interpolation constraints in } (F, G), \\ & \text{linear initial conditions in } (F, G). \end{aligned}$$

- `PEPit.jl` lets us write the PEP in its most natural mathematical description and does the conversion to SDP automatically.

Solving the example problem in `PEPit.jl`

Initialize PEP. Let us start with defining an empty PEP, which we will construct step by step.

```
problem = PEP([], [], [], [], [], nothing)
```

```
1 problem = PEP()
```

Define function parameters. Define the function parameters μ, L, α.

```
1.0
```

```
begin
    mu=0.1
    L=1.0
    alpha=1.0
end
```

Define function class Now we define the function class $\mathcal{F}_{\mu,L}$ itself. Here `reuse_gradient=true` means that our function is differentiable.

```
func =
 SmoothStronglyConvexFunction(0.1, 1.0, PEPFunction(1, true, OrderedDict(circular reference ⇒ 1
```

```
func = declare_function!(problem, SmoothStronglyConvexFunction, OrderedDict("mu" =>
mu, "L" => L); reuse_gradient=true)
```

Define initial+optimal points. Define the initial point x_0 as `x0` and the optimal point $x_\star$ as `xs`.

```
xs = Point(2, true, OrderedDict(circular reference ⇒ 1.0), 0, nothing)
```

```
xs = stationary_point!(func)
```

```
x0 = Point(5, true, OrderedDict(circular reference ⇒ 1.0), 1, nothing)
```

```
x0 = set_initial_point!(problem)
```

Define algorithm. Now define the gradient descent step:

$$x_1 = x_0 - \alpha \nabla f(x_0)$$

```
x1 =
 Point(10, false, OrderedDict(Point(5, true, OrderedDict( more), 1, nothing) ⇒ 1.0, Point(7,
```

```
x1 = x0 - alpha * gradient!(func, x0)
```

Initial condition. Next, define the *initial condition*: $\|x_0 - x_\star\|^2 \leq 1$.

```
[Constraint(Expression(14, false, OrderedDict( more), nothing, nothing), "inequality", 0, no
```

```
set_initial_condition!(problem, (x0 - xs)^2 <= 1)
```

Performance metric. Finally, define the *performance metric*, which measures how far we are from the optimal point $x_\star$. Smaller values are better; zero means the iterate is optimal.

```
[Expression(17, false, OrderedDict((Point(5, true, OrderedDict( more), 1, nothing), Point(5,
```

```
set_performance_metric!(problem, (x1 - xs)^2)
```

Solving the PEP

We now have everything needed to build the PEP. It is time to solve it.

```
primal_result =
```

```
(wc_value = 0.81, model = A JuMP Model
                          └ solver: Clarabel
```

```
primal_result = solve!(problem; verbose = true, return_full_model = true)
```

```
 PEPit:  Setting up the problem: size of the main PSD matrix: 3x3
 PEPit:  Setting up the problem: performance measure is minimum of 1 element
(s)
 PEPit:  Setting up the problem: Adding initial conditions and general constr
aints ...
 PEPit:  Setting up the problem: initial conditions and general constraints
(1 constraint(s) added)
 PEPit:  Setting up the problem: interpolation conditions for 1 function(s)
         function 1 : 2 scalar constraint(s) added
 PEPit:  Compiling SDP
 PEPit:  Calling SDP solver
-----------------------------------------------------------------
           Clarabel.jl v0.11.1  -  Clever Acronym

                   (c) Paul Goulart
                University of Oxford, 2022
-----------------------------------------------------------------

problem:
  variables     = 9
  constraints   = 10
  nnz(P)        = 0
  nnz(A)        = 32
  cones (total) = 2
    : Nonnegative = 1,  numel = 4
    : PSDTriangle = 1,  numel = 6

settings:
```

```
tau_star = 0.8100000002841585
```

```
tau_star = primal_result.wc_value
```

```
G_star = 3×3 Matrix{Float64}:
          0.528314   0.0283115  -0.0500003
          0.0283115  0.528309    0.0499997
         -0.0500003  0.0499997   0.01
```

```
G_star = JuMP.value.(primal_result.variables.G)
```

```
F_star = [-0.025, 0.025]
```

```
F_star = JuMP.value.(primal_result.variables.F)
```

Parameter sweeps for the gradient-step PEP

We wrap the one-step gradient descent PEP in a function that takes μ and α as inputs.

Throughout these sweeps, we fix $L = 1$ and compute the worst-case contraction factor $\tau^\star(\mu, \alpha)$.

```
gd_contraction_pep_problem (generic function with 1 method)
```

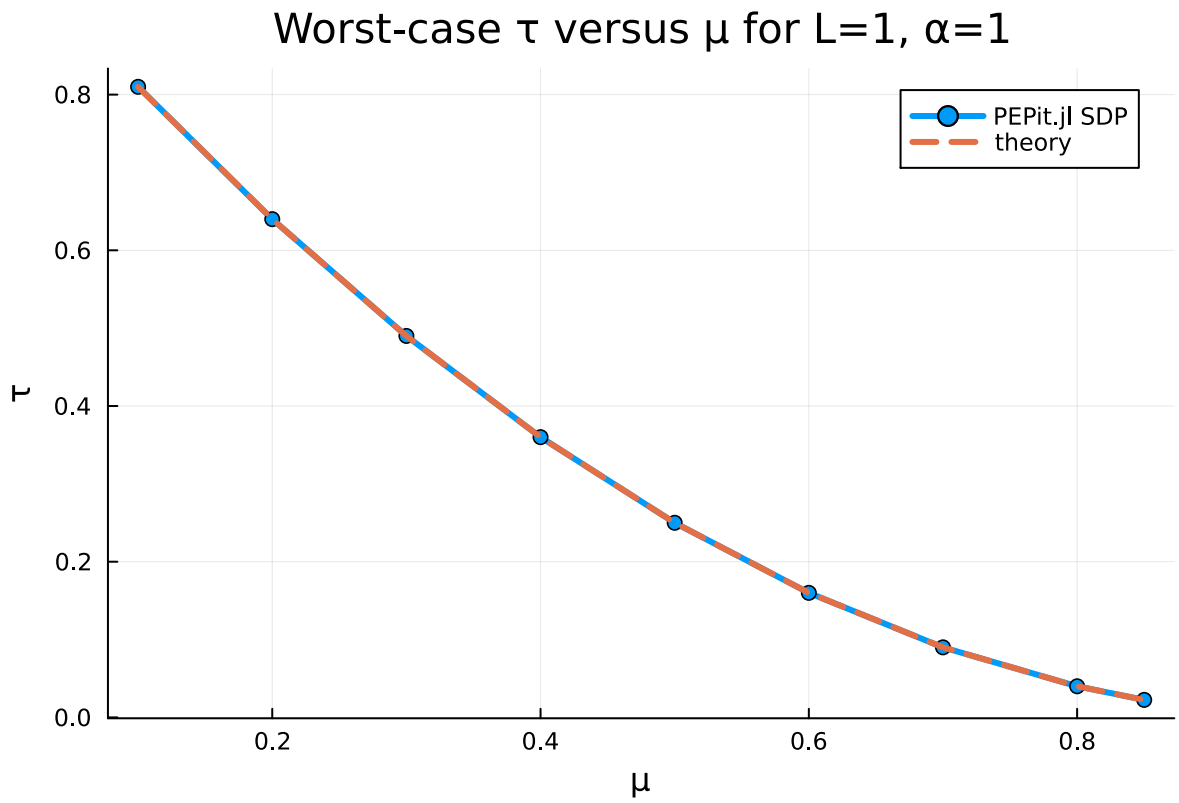
```
gd_worst_case_tau (generic function with 1 method)
```

```
gd_theory_tau (generic function with 1 method)
```

Plot of τ vs μ

Fix $L = 1$ and $\alpha = 1$. Vary μ from 0.1 to 0.85.

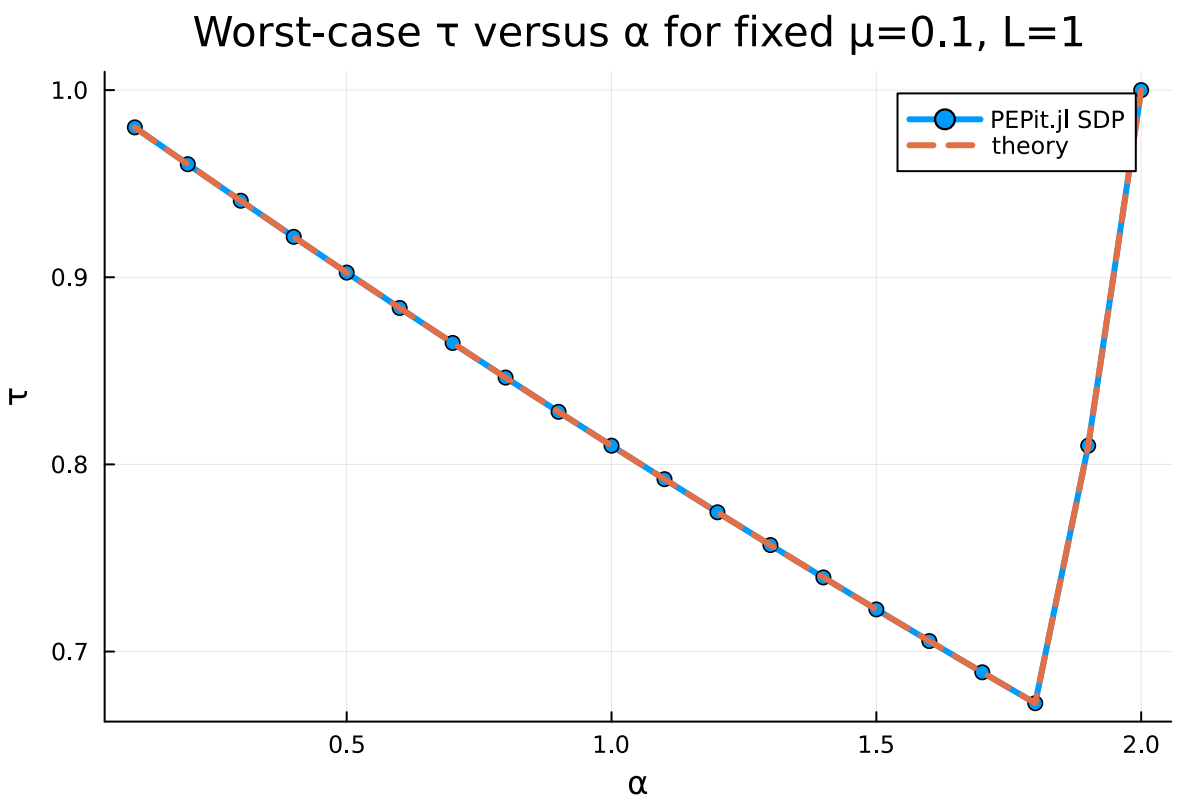
```
[0.81, 0.64, 0.49, 0.36, 0.25, 0.16, 0.09, 0.04, 0.0225]
```




```
[0.9801, 0.9604, 0.9409, 0.9216, 0.9025, 0.8836, 0.8649, 0.8464, 0.8281, 0.81, 0.7921, 0.774
```

Plot of τ vs α

Fix $L = 1$ and $\mu = 0.1$. Vary α from 0.1 to 2.0 with granularity 0.1.



Internal architecture of `PEPit.jl`

- At a high level, it follows the architecture of Python `PEPit`.
- The Julia package uses `structs`, `methods`, and `multiple dispatch` instead of Python-style classes.

The package is built around a small set of mathematical objects:

What?	Why?
`Point`	`vector-like quantity: iterate, gradient, residual`
`Expression`	`scalar quantity: function value, norm square, inner product`
`Constraint`	`scalar equality or inequality`
`PSDMatrix`	`linear matrix inequality`
`PEPFunction`	`sampled function with interpolation constraints`
`PEP`	`container for assumptions, initial conditions, metrics, and SDP assembly`

Step 1: user input becomes a `PEP`

The user describes a worst-case analysis problem through four ingredients:

- **Problem class:** assumptions such as $f \in \mathcal{F}_{\mu,L}$.

- **Algorithm:** a first-order recurrence generating $x_1, \dots, x_N$.
- **Initial condition:** a normalization such as $\mathcal{I}(x_0, \dots) \leq 1$.
- **Performance measure:** the quantity $\mathcal{P}(x_N, \dots)$ to maximize.

In `PEPit.jl`, these ingredients are stored in a `PEP` object.

Mathematically, the `PEP` represents

$$\tau^\star = \sup \{\mathcal{P}(x_N, \dots, x_0, f) : f \in \mathcal{F},\ \mathcal{I}(x_0, \dots, f) \leq 1,\ \text{algorithmic recurrence holds}\}.$$

So `PEP` is the container that accumulates:

- sampled functions,
- symbolic points and expressions,
- initial conditions,
- performance metrics,
- constraints needed to assemble the final SDP.

Step 2: symbolic execution creates `Point` and `Expression` objects

`PEPit.jl` first runs the algorithm symbolically.

- Vector-like quantities are represented by `Point` objects:

$$x_0,\ x_1, \dots, x_N, \qquad g_i = f'(x_i), \qquad r_i, \dots$$

- Scalar quantities are represented by `Expression` objects:

$$f_i = f(x_i), \qquad \|x_i - x_j\|^2, \qquad \langle g_i, x_j - x_k \rangle.$$

- For example, a gradient step

$$x_{i+1} = x_i - \alpha_i f'(x_i)$$

is stored as an algebraic relation between `Point` objects.

- The performance measure and initial condition are `Expression` inequalities:

$$\mathcal{P}(x_N, \dots) \quad \text{and} \quad \mathcal{I}(x_0, \dots) \leq 1.$$

Step 3: `PEPFunction` generates interpolation

`Constraint`s

- The problem class is represented by a `PEPFunction`.
- For each sampled point x_i, the `PEPFunction` introduces symbolic data

$$(x_i, g_i, f_i), \qquad g_i = f'(x_i), \qquad f_i = f(x_i).$$

- The infinite-dimensional statement

$$f \in \mathcal{F}$$

is replaced by finitely many interpolation `Constraint`s:

$$\{(x_i, g_i, f_i)\}_{i \in I} \text{ is extendable to some } f \in \mathcal{F}.$$

- For $\mathcal{F}_{\mu,L}$, these are exactly the smooth strongly convex interpolation inequalities seen earlier.

Thus:

- `PEPFunction` knows the function class,
- it samples gradients and function values,
- it contributes the interpolation `Constraint`s to the `PEP`.

Step 4: `Constraint`s are linearized through the Gram matrix and `PSDMatrix`

- All vector quantities appearing in the PEP are collected into a list

$$p_1, \ldots, p_m.$$

- These may include iterates, gradients, residuals, or differences of such objects.

`PEPit.jl` introduces the Gram matrix

$$G = \begin{bmatrix} \langle p_1, p_1 \rangle & \cdots & \langle p_1, p_m \rangle \\ \vdots & \ddots & \vdots \\ \langle p_m, p_1 \rangle & \cdots & \langle p_m, p_m \rangle \end{bmatrix} \succeq 0.$$

- Every norm square and inner product becomes linear in G:

$$\|p_i\|^2 = G_{i,i}, \qquad \langle p_i, p_j \rangle = G_{i,j}.$$

- The condition $G \succeq 0$ is represented internally as a `PSDMatrix`.

So the conversion is:

$$\text{Point geometry} \quad \longrightarrow \quad \text{Gram matrix } G \quad \longrightarrow \quad \texttt{PSDMatrix}.$$

Step 5: the `PEP` assembles the final SDP

After the Gram lifting, the `PEP` has everything needed to build the SDP.

- `Point` objects determine the Gram matrix entries.
- `Expression` objects become linear functions of G and scalar values F.
- `Constraint` objects become linear equalities or inequalities.
- `PSDMatrix` objects become semidefinite constraints.
- `PEPFunction` objects contribute interpolation constraints.
- The `PEP` object assembles all pieces into a JuMP model.

The final SDP has the form

$$\begin{aligned} \text{maximize}_{G,F} \quad & \langle C, G \rangle + c^\top F \\ \text{subject to} \quad & \langle A_\ell, G \rangle + a_\ell^\top F \leq b_\ell, \qquad \ell = 1, \dots, m, \\ & G \succeq 0. \end{aligned}$$

The primal SDP returns the worst-case value $\tau^\star$. The dual SDP gives a proof certificate:

$$\mathcal{P}(x_N, \dots, x_0, f) \leq \tau^\star \mathcal{I}(x_0, \dots, f).$$

In short:

$$\texttt{PEP} \Rightarrow \texttt{Point} + \texttt{Expression} + \texttt{PEPFunction} \Rightarrow \texttt{Constraint} + \texttt{PSDMatrix} \Rightarrow \text{JuMP}.$$

Function classes and primitive steps

The package includes core function classes such as:

- convex functions,
- smooth functions,
- strongly convex functions,
- smooth strongly convex functions,
- convex Lipschitz functions,
- convex indicators.

It also includes primitive algorithmic steps:

- gradient and inexact gradient steps,
- proximal and inexact proximal steps,
- Bregman gradient/proximal steps,
- exact line search,
- linear and shifted optimization steps.

These are small compositional blocks for building many PEPs without rewriting interpolation logic.

Operator splitting

Beyond smooth optimization, `PEPit.jl` can model operator-splitting problems, such as

- monotone operators,
- nonexpansive operators,
- Lipschitz operators,
- linear operators.

This supports examples such as:

- accelerated proximal point,
- Douglas-Rachford splitting,
- three-operator splitting,
- fixed-point iterations.

The same workflow applies: define objects, run the recurrence, set the metric, and solve the PEP.

Different optimization setups: stochastic, online, potential, and worst-case examples

The example suite also includes:

- stochastic gradient descent,
- online gradient descent,
- nonconvex gradient descent,
- optimized gradient methods,
- potential-function examples,
- low-dimensional worst-case scenario searches.

For a JuMP audience, this is the important extensibility point:

$$\text{new modeling ingredient} \quad \Rightarrow \quad \text{new interpolation/constraint block} \quad \Rightarrow \quad \text{same JuMP s}$$

- Note that there is a dedicated package called `AutoLyap` with both Python and Julia implementations for potential function analysis [Upadhyaya et al. 2026].

Future plans

We are working toward incorporating step-size optimization in `PEPit.jl` and Python `PEPit`:

> inner problem: certify a worst-case bound for fixed algorithm parameters,
> outer problem: optimize over step sizes or algorithm coefficients.

Here:

- the inner dual provides a certificate structure (`PEPit.jl` can already solve it separately),
- the outer problem searches over algorithm parameters,
- branch-and-bound [Das Gupta–Van Parys–Ryu 2024] or other nonlinear methods [Kamri–Hendrickx–Glineur 2025] can handle the nonconvexity,
- in some cases [Drori–Taylor 2020; Taylor–Drori 2023], step-size optimization can also be solved as a convex SDP,
- the output should be both a method and a proof.

This is the bridge from analyzing a method to discovering new, more efficient methods!

Thank you!

References

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

[Taylor–Hendrickx–Glineur 2017a] A. B. Taylor, J. M. Hendrickx, and François Glineur. *Smooth strongly convex interpolation and exact worst-case performance of first-order methods.* Mathematical Programming, 2017.

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

[Goujaud et al. 2024] B. Goujaud, C. Moucer, François Glineur, J. M. Hendrickx, A. B. Taylor, and A. Dieuleveut. *PEPit: computer-assisted worst-case analyses of first-order optimization methods in Python*. Mathematical Programming Computation, 2024.

[Taylor 2024] A. B. Taylor. *Towards principled and systematic approaches to the analysis and design of optimization algorithms*. 2024.

[Das Gupta–Van Parys–Ryu 2024] 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, 2024.

[Kamri–Hendrickx–Glineur 2025] Y. Kamri, J. M. Hendrickx, and François Glineur. *Numerical design of optimized first-order algorithms*. arXiv:2507.20773, 2025.

[Upadhyaya et al. 2026] M. Upadhyaya, S. Das Gupta, A. B. Taylor, S. Banert, and P. Giselsson. *The AutoLyap software suite for computer-assisted Lyapunov analyses of first-order methods*. arXiv:2506.24076, 2026.

[Drori–Taylor 2020] Y. Drori and A. B. Taylor. *Efficient first-order methods for convex minimization: a constructive approach*. Mathematical Programming, 2020.

[Taylor–Drori 2023] A. B. Taylor and Y. Drori. *An optimal gradient method for smooth strongly convex minimization*. Mathematical Programming, 2023.