Automatic recommendation of optimization methods through their worst-case complexity

François Glineur

July 2025

Abstract

Since the advent of modern computational mathematics, the literature on optimization algorithms has been ever-growing, and a wide range of methods are now available. Usually, published methods are shown to be applicable to given templates of optimization problems and are often accompanied by proofs characterizing their convergence rates. There exist many variations of such templates. Indeed, a template may describe the objective as a single function or a sum of functions with different characteristics (say, smoothness, convexity, existence of a computable proximal operator, etc.), accompanied by constraints that can also be described in various ways (e.g. linear constraints, feasible set with a computable projection operator, functional constraints, etc.). Hence, for a given optimization problem, it is not always easy to identify which templates it can match, especially if one wants to consider equivalent reformulations of the problem. This makes the task of choosing an optimization method tedious. We present a general framework for the recommendation of optimization methods for any oracle-based problem. Given a user-provided optimization problem, the framework works as follows: first, compute equivalent reformulations of the problem; second, check for each formulation whether they satisfy a known template; third, retrieve known optimization methods applicable to these templates; and fourth, order methods given their certified worst-case performance. Numerical experiments show that our theoretical ranking of the optimization methods correlates with benchmarks on corresponding instances. Finally, we release both the open source Julia package Argo.jl that implements the framework and the extensive library of optimization methods on which it is based.

Contents

1 Introduction

Optimization has been a growing field for almost a century, and there now exists a multiple of optimization methods, scattered in the literature. Optimization methods are designed and analyzed with a focus on specific problem formulations, which we call *templates*. There exist many variations of such templates. Indeed, a template may describe the objective as a single function or a sum of functions with different characteristics (say, smoothness, convexity, existence of a computable proximal operator, etc.), accompanied by constraints that can also be described in various ways (e.g. linear constraints, feasible set with a computable projection operator, functional constraints, etc.). These templates not only streamline analysis, but also maintain the capacity to represent all problems within a particular class. To solve an optimization problem, researchers and practitioners must first determine which optimization methods are applicable. This requires the often tedious task of aligning the problem with one of the existing templates. In fact, optimization problems can be represented through various equivalent formulations. Some formulations may allow for a better understanding of the problem, while others can be more computationally suitable for certain methods. Furthermore, related auxiliary problems may appear within an optimization method and can also be cast as mathematical programming formulations. Given a specific formulation of an optimization problem, finding alternative reformulations typically requires applying various transformations until (hopefully) reaching other formulations of interest (typically templates). This reformulation process is time-consuming and error-prone, and its effectiveness is contingent on the number of known transformations and target templates. However, if this reformulation process were to be automated, it would become simple to know whether an optimization problem satisfies a certain template, and thus to know which methods are applicable to a given optimization problem. Our main point of interest, beyond the enumeration of applicable methods, is to have a standard procedure to choose optimization methods among them.

1.1 Our approach

In this paper, we present an automated approach to identify applicable optimization methods and rank them based on their worst-case performance. Specifically, our approach addresses the question:

Which optimization methods offer (the best) worst-case guarantees for a given optimization problem?

At first sight, this approach seems somehow limited. Worst-case analysis, as its name suggests, establish complexity bounds which hold for any instance problem in a class, implicitly considering the most difficult instances of the class. This type of analysis ensures reliability of the methods, is essential for any kind of critical application and provides a solid theoretical foundation for comparing algorithms. However, the hard-to-solve instances taken into account in such analyses may occur rarely in practice. As a result, obtained complexity bounds might not always reflect the typical runtime behaviour of methods. We argue that our problem is nevertheless a meaningful one. First, this question bears theoretical interest. Indeed, understanding worst-case guarantees contributes to the broader theory of algorithmic complexity and helps delineate the limits of algorithmic performance. Second, given the vast number of methods available in the literature, our approach provides a principled way to preliminarily filter and rank methods before resorting to empirical benchmarking. Moreover, it is methodologically sound to first consider what theory predicts, especially in settings where benchmarking is costly or infeasible. Also, our framework is extensible and could serve as a foundation for incorporating other forms of analysis, such as average-case or empirical performance, in future work. Naturally, optimization methods with loose worst-case guarantees (i.e. overly pessimistic complexity bounds) are bound to be put at a disadvantage. Thankfully, rich new theory has been developed in recent years, largely enabled by the performance estimation problem (PEP) framework established in [citation needed]. In a few words, the PEP framework allows for finding exact worst-case convergence rates of optimization methods across various "interpolable" templates. It has been successfully applied to compute the exact convergence rates of many methods in diverse settings. Consequently, most of the worst-case guarantees considered in our framework are tight or nearly tight. In addition to making the comparison more fair (less methods are disadvantaged because they are harder to analyse tightly), this allows more refined comparison of methods. Our approach is then as follows:

1. Automatically generate multiple reformulations of a given problem,
2. Check if a reformulation fits specific template(s),
3. Recover first-order methods proven to solve such template(s) along with their complexity bounds,
4. Analyze all retrieved complexity bounds to recommend the best optimization methods for the given problem.

1.2 Contributions

Our main contributions are as follows:

1. We aggregate in an open source database a large collection of existing convergence results on which our recommendation framework relies. Convergence results are written in the form (Template, Method, Rate).

2. We provide a procedure matching a given oracle-based optimization problem to templates existing in the aforementioned database.

   This procedure relies on a variety of transformation techniques implemented in the Argo package.

3. We provide a procedure retrieving optimization methods applicable to a given optimization problem, sorted by their worst-case performance.

1.3 Related work

Benchmarking in optimization. The method selection problem is often addressed empirically through benchmarking. Benchmarking consists in solving a wide range of representative problem instances with various optimization methods and comparing their practical performance to identify the most promising methods. This approach, however, is cumbersome and only yields meaningful results when conducted with rigorous scientific methodology, see [1] for best practices in benchmarking optimization methods and [2] for a Python package making the benchmarking process reproducible, less tedious and less error-prone. While benchmarking provides valuable empirical insights, it overlooks both theoretical recommendations and, most of the time, potential transformations of the original problem.

Automatic transformations. There exists many frameworks applying successive transformations to an initial problem to enable the use of a more efficient method. In general modeling languages, the goal is to match the problem to the standard form required by a solver chosen by the user. Similar frameworks exist in continuous optimization with first-order methods. We highlight two examples. First, in convex signal processing tasks, TFOCS [3] performs transformations before applying an optimal first-order method to the modified problem. This approach is similar to ours in essence as its goal is to use the method with the best theoretical guarantees on an optimized formulation but it is tailored to specific signal processing problems. Moreover, their transformation process is systematic and always yields a standard form. Second, for additive composite minimization, the Catalyst meta-algorithm [4] can be interpreted as an inexact

accelerated proximal point method, where the proximal operator is solved approximately using an auxiliary method. We show in Section 5 that our approach can naturally design similar meta-algorithms.

Aggregation and structuration of optimization knowledge. Linnaeus [5; 6] is a computer algebra system which checks whether two iterative optimization methods are equivalent by analyzing their transfer function. It also acts as a searchable repository of optimization methods, allowing to identify whether a method is equivalent to another one already present in Linnaeus' library. However, it's about checking and observing that optimization methods are equivalent. It is in that sense more of a theoretical tool and not something that can be inserted in the optimization pipeline.

Automated worst-case complexity analysis. The framework we propose heavily relies on worst-case complexity bounds to make recommendations. Arguably, algorithms with worst-case bounds that are too loose can suffer from this criterion. Thankfully, there is a large body of literature that is dedicated to finding and computing exact worst-case convergence rates for different methods, function classes and performance metrics [citation needed]. This is the PEP framework. One can solve a small SDP and get the exact numerical value for a worst-case, via a nice MATLAB [citation needed]or Python [citation needed]API. This is done under the hood in our toolbox to retrieve some worst-case convergence rates.

Algebraic modeling languages. CVX, YALMIP, GAMS, AMPL, JuMP etc. They all represent mathematical expressions as a directed acyclic graph and transform it until it reaches a standard form for a given solver. It just so happens that they use a similar technique to ours but we definitely do not have the same goal.

1.4 Organization of the paper

This paper is organized as follows. In Section 2, we illustrate our approach through a simple example, namely X, and explain how to reproduce such results with the Argo.jl package. In Section 3, we focus on optimization modeling. We describe the types of problems that can be formulated using the `Argo.jl` package. We also explain how reformulations of these problems are computed within the framework. In Section 4, we discuss how optimization methods are described within the framework and explain how applicable methods can be retrieved and ranked. Then, in Section 5, we present three applications of the Argo.jl framework, namely X, Y and Z. Some concluding remarks are drawn in Section 6.

2 Overview of Argo.jl through a simple example

Here we present a simple example to illustrate how `Argo.jl` works. We will first derive recommendations by hand then show how they can be obtained automatically with the `Argo.jl` package.

Suppose you want to solve an elastic-net logistic regression problem. Simply put, the goal is to learn a linear classifier that is both sparse and stable by regularizing a logistic loss with both a ℓ_1 penalty (which favor sparse solutions) and a ℓ_2 penalty. This can be written as follows: given a training data set $\{(a_i, b_i)\}_{i=1}^n$ with feature vectors $a_i \in \mathbb{R}^d$ and binary labels $b_i \in \{\pm 1\}$, and regularization parameters $\lambda_1, \lambda_2 > 0$, we solve

$$\min_{x \in \mathbb{R}^d} F(x) \stackrel{\text{def}}{=} \frac{1}{n} \sum_{i=1}^{n} \log\left(1 + \exp(-b_i\, a_i^T x)\right) + \lambda_1 \|x\|_1 + \frac{\lambda_2}{2}\|x\|^2, \tag{1}$$

where the first term is the data-fitting term which uses the logistic loss.

We want to match such a problem to existing optimization templates in the literature. There are three functionals: (1) the data-fitting term is smooth, convex and we assume access to its gradient for any $x \in \mathbb{R}^d$, (2) the ℓ_1 norm is Lipschitz, convex and we can compute its subgradient and proximal operator, (3) the ℓ_2 norm is smooth, convex and we can compute its gradient and proximal operator. We can regroup the terms in multiple ways, among which:

$$F(x) = \left[\frac{1}{n} \sum_{i=1}^{n} \log\left(1 + \exp(-b_i\, a_i^T x)\right) + \lambda_1 \|x\|_1 + \frac{\lambda_2}{2}\|x\|^2\right] \tag{2}$$

$$F(x) = \left[\frac{1}{n} \sum_{i=1}^{n} \log\left(1 + \exp(-b_i\, a_i^T x)\right) + \frac{\lambda_2}{2}\|x\|^2\right] + \left[\lambda_1 \|x\|_1\right] \tag{3}$$

$$F(x) = \left[\frac{1}{n} \sum_{i=1}^{n} \log\left(1 + \exp(-b_i\, a_i^T x)\right)\right] + \left[\lambda_1 \|x\|_1 + \frac{\lambda_2}{2}\|x\|^2\right] \tag{4}$$

$$F(x) = \left[\frac{1}{n} \sum_{i=1}^{n} \log\left(1 + \exp(-b_i\, a_i^T x)\right)\right] + \left[\lambda_1 \|x\|_1\right] + \left[\frac{\lambda_2}{2}\|x\|^2\right] \tag{5}$$

The formulation Eq. (2) leads to a nonsmooth convex objective function for which we can compute a subgradient at any given test point x. In that case, the best possible convergence rate is of the order $\frac{1}{\sqrt{T}}$ where T is the number of iterations [citation needed]. Moreover, such a convergence rate can be attained by a normalized subgradient scheme. For the second formulation Eq. (3), we have the sum of a strongly convex smooth term with computable gradient and a proper

closed convex term with computable proximal operator. For such problems, the best algorithms yield linear convergence rates to decrease the suboptimality gap. For the third formulation Eq. (4), strong convexity is not guaranteed anymore for the first term but the proximal operator of the elastic-net penalization can be easily computed. We therefore have the sum of a smooth and convex term with gradient with a strongly convex proximable term. Finally, the fourth formulation Eq. (5) is the sum of three terms, a convex term with gradient, a convex term with proximal operator and a strongly convex term with proximal operator. Methods for such a template exist, see Davis-Yin splitting. It is also important to note that it is possible to transfer curvature for the last two formulations Eqs. (3) to (5). Indeed, for Eq. (3), the ℓ_2^2 regularization term can be dispatched which yields the following formulation for any $\rho \in [0, \lambda_2)$:

$$F(x) = \left[\frac{1}{n}\sum_{i=1}^{n} \log\left(1 + \exp(-b_i\, a_i^T x)\right) + \frac{\rho}{2}\|x\|^2\right] + \left[\lambda_1\|x\|_1 + \frac{\lambda_2 - \rho}{2}\|x\|^2\right] \quad (6)$$

and similarly for Eq. (5)

$$F(x) = \left[\frac{1}{n}\sum_{i=1}^{n} \log\left(1 + \exp(-b_i\, a_i^T x)\right) + \frac{\rho}{2}\|x\|^2\right] + \left[\lambda_1\|x\|_1\right] + \left[\frac{\lambda_2 - \rho}{2}\|x\|^2\right] \quad (7)$$

To find the best method theoretically it remains to compare the rates of the optimal methods for each of the templates associated to the above formulations. It is enough to look at the optimal methods for each of the formulations proposed above. The first formulation can be ditched: too much structure is lost and the optimal convergence rate is $\mathcal{O}(T^{-1})$ (strongly convex nonsmooth nonlipschitz, see Grimmer [citation needed]). For the two-terms formulations, it is straightforward to ditch all the ones that have no strong convexity in the loss term because these can't provide linear convergence. Therefore, four formulations are left to consider:

$$F(x) = \left[\frac{1}{n}\sum_{i=1}^{n} \log\left(1 + \exp(-b_i\, a_i^T x)\right) + \frac{\rho}{2}\|x\|^2\right] + \left[\lambda_1\|x\|_1 + \frac{\lambda_2 - \rho}{2}\|x\|^2\right], \quad (8)$$

$$F(x) = \left[\frac{1}{n}\sum_{i=1}^{n} \log\left(1 + \exp(-b_i\, a_i^T x)\right) + \frac{\rho}{2}\|x\|^2\right] + \left[\lambda_1\|x\|_1\right] + \left[\frac{\lambda_2 - \rho}{2}\|x\|^2\right], \quad (9)$$

$$F(x) = \left[\frac{1}{n}\sum_{i=1}^{n} \log\left(1 + \exp(-b_i\, a_i^T x)\right)\right] + \left[\lambda_1\|x\|_1\right] + \left[\frac{\lambda_2}{2}\|x\|^2\right] \quad (10)$$

and

$$F(x) = \left[\frac{1}{n}\sum_{i=1}^{n} \log\left(1 + \exp(-b_i\, a_i^T x)\right) + \frac{\lambda_2}{2}\|x\|^2\right] + \left[\lambda_1\|x\|_1\right]. \quad (11)$$

It is important to note that the cost of evaluating the proximal operator of the ℓ_1 penalty and the proximal operator of the elastic-net penalty are similar. Now it is known that the fastest methods for such templates are the strongly convex variant of FISTA (with matching lower bound, see [citation needed]) and the Davis-Yin splitting [citation needed](no matching lower bound for the template so we don't know if it's tight and if there could be a better algorithm that will outdate what I currently say). Now, the rates are here:

2.1 Code

This problem can be declared easily in `Argo.jl` as follows

```
using Argo
@variable x::R(:d)
@function f(R(:d), R())
expr = f(x) + L1norm(x, :λ1) + L2norm(x, :λ2)
```

Notice that `Argo.jl` does not support natively the logistic loss, which is here just named f. Instead, we describe the logistic loss via its properties and available oracles as follows:

```
@property f Smooth(:L)
@property f Convex()
@oracle f Derivative()
```

Regarding the ℓ_1 and ℓ_2 norms, it is not necessary to declare their properties and oracles as they are already predefined within the toolbox. The recommendation module

3 Argo.jl: Modelisation

3.1 Problem formulation

The Argo.jl package is designed to model a wide range of optimization problems.

Formulating a problem in `Argo.jl` hinges on three core modules: the `Language` module, the `Properties` module, and the `Oracle` module.

These modules work together to define the structure of the problem, the known properties of each of its components, and the oracles that can be used for each component.

Language module

The `Language` module is responsible for defining the structure of the problem. It provides a set of types that can be used to represent the different components of the problem. The core types in the `Language` module are:

- **Space**: Represents a vector space, such as $\mathbb{R}$ or $\mathbb{R}^n$.
- **Variable**: Represents a decision variable in the problem.
- **FunctionType**: Represents a function component in the problem, which may have properties and/or oracles associated with it.

These types can be combined to create more complex structures using operator types, such as:

- **FunctionCall**: Represents a function call, which can be used to apply a function to a set of variables.
- **Addition**: Represents the addition of two or more functions.
- **Subtraction**: Represents the subtraction of two or more functions.
- **Composition**: Represents the composition of two functions.
- **Maximum**: Represents the maximum of two or more functions.
- **Minimum**: Represents the minimum of two or more functions.

Any combination of the core types using the operator types can be used to define a problem in `Argo.jl`. The resulting structure is a directed acyclic graph (DAG) that represents the problem, which is of type **Expression**.

Properties module

The `Properties` module is responsible for defining the known properties of each component in the problem. It provides a set of types that can be used to represent the different properties of the components, such as:

- **Convex**: Represents a convex function.
- **MonotonicallyIncreasing**: Represents a function that is monotonically increasing.
- **StronglyConvex**: Represents a strongly convex function.
- **HypoConvex**: Represents a hypo-convex function.

- **Smooth**: Represents a smooth function.
- **Lipschitz**: Represents a Lipschitz continuous function.
- **Linear**: Represents a linear function.
- **Quadratic**: Represents a quadratic function.

These properties can be associated with the components of the problem using the `Properties` module. In practice, properties are associated with the `FunctionType` type, which can be used to represent the different components of the problem. Expressions directly inherit properties from their components, which allows infering properties of subexpressions.

Oracle module

The `Oracle` module is responsible for defining the oracles that can be used for each component in the problem. It provides a set of types that can be used to represent the different oracles:

- **Oracle**: Represents a generic oracle for a function.
- **EvaluationOracle**:
- **DerivativeOracle**: Represents an oracle that provides the (sub)gradient of a function.
- **ProximalOracle**: Represents an oracle that provides the proximal operator of a function.
- **LinearMinimizationOracle**: Represents an oracle that provides the solution to a linear minimization problem.

These oracles can be associated with the components of the problem using the `Oracle` module. The oracles can be used to evaluate the components of the problem, compute their (sub)gradients, and solve linear minimization problems. It is also possible to define inexact oracles, by specifying an absolute and/or relative tolerance for the oracle. It is also possible to specify the cost of evaluating the oracle, to accomodate for example stochastic or inexact oracles.

3.2 Reformulation strategies

The reformulation module hinges on what we call *reformulation strategies*. These are the building blocks of the reformulation module, and they are responsible for transforming a given problem into a different one. The reformulation strategies are designed to be modular and composable, allowing users to combine them in various ways to achieve the desired results.

The current reformulation strategies are listed below.

Rebalancing

This strategy is used to exploit the associative property of certain operations. It can be used to rearrange terms in an equation or to group similar terms together.

Commutativity

This strategy is used to exploit the commutative property of certain operations. It can be used to rearrange terms in an equation or to group similar terms together.

Curvature transfer

This strategy is used to transfer the curvature of a function to another one. It is particularly useful for problems where the curvature of the objective function is known or can be estimated.

Structure loss

This strategy is used to simplify the structure of a problem. It collapses structured problems into simpler ones and is a core component for the matching module.

Inexact operators

This strategy is used to generate new oracles for certain functions. It is particularly useful for problems where an oracle is not available in closed-form but can be approximated using an auxiliary method.

Monotone transform

This strategy applies a monotone transformation to a function. It is particularly useful for problems where the objective function is not convex but can be trans-

formed into a convex one using a monotone transformation. It can also improve conditioning in certain cases.

Moreau envelope

This strategy applies the Moreau envelope to a function. It is particularly useful for problems where the objective function is not smooth but can be transformed into a smooth one using the Moreau envelope. It can also improve conditioning in certain cases.

Primal-dual saddle-point reformulation

This strategy reformulates a problem into a primal-dual saddle-point problem. It is particularly useful for problems where the primal and dual problems are closely related and can be solved simultaneously.

4 Argo.jl: Computation

5 Applications

5.1 Computing the optimal reformulation parameters

5.2 Equivalence between PGM and DCA

5.3 Moreau envelope reformulation

6 Conclusion

A Optimization templates

B Optimization methods

C Implementation details

References

[1] Thomas Bartz-Beielstein, Carola Doerr, Daan van den Berg, Jakob Bossek, Sowmya Chandrasekaran, Tome Eftimov, Andreas Fischbach, Pascal Kerschke, William La Cava, Manuel

Lopez-Ibanez, et al. Benchmarking in optimization: Best practice and open issues. *arXiv preprint arXiv:2007.03488*, 2020.

[2] Thomas Moreau, Mathurin Massias, Alexandre Gramfort, Pierre Ablin, Pierre-Antoine Bannier, Benjamin Charlier, Mathieu Dagréou, Tom Dupré la Tour, Ghislain Durif, Cassio F. Dantas, Quentin Klopfenstein, Johan Larsson, En Lai, Tanguy Lefort, Benoit Malézieux, Badr Moufad, Binh T. Nguyen, Alain Rakotomamonjy, Zaccharie Ramzi, Joseph Salmon, and Samuel Vaiter. Benchopt: Reproducible, efficient and collaborative optimization benchmarks. In *NeurIPS*, 2022.

[3] Stephen R Becker, Emmanuel J Candès, and Michael C Grant. Templates for convex cone problems with applications to sparse signal recovery. *Mathematical programming computation*, 3:165–218, 2011.

[4] Hongzhou Lin, Julien Mairal, and Zaid Harchaoui. Catalyst acceleration for first-order convex optimization: from theory to practice. *Journal of Machine Learning Research*, 18 (212):1–54, 2018.

[5] Shipu Zhao, Laurent Lessard, and Madeleine Udell. An automatic system to detect equivalence between iterative algorithms. *arXiv preprint arXiv:2105.04684*, 2021.

[6] Laurent Lessard and Madeleine Udell. Algebraic characterization of equivalence between optimization algorithms. *arXiv preprint arXiv:2501.04972*, 2025.