ALGORITHM ANALYSIS: AUTOMATIC LYAPUNOV-BASED ANALYSIS OF FIRST-ORDER METHODS IN JULIA

Lam Ngoc Ha

A THESIS

Presented to the Faculty of Miami University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Electrical and Computer Engineering

The Graduate School

Miami University

Oxford, Ohio

2025

Advisor: Dr. Bryan Van Scoy

Reader: Dr. Veena Chidurala

Reader: Dr. Peter Jamieson

ABSTRACT

First-order iterative algorithms are widely employed to solve large-scale optimization problems, many of which often appear in science or engineering. A relevant example of such a problem is in the field of machine learning, where huge amounts of computational power are spent optimizing model errors. Consequently, the ability to systematically compare algorithm performance across any given application to find the most efficient method can save time and energy. However, existing methods for algorithm analysis often have shortcomings ranging from the rigorousness of the result to user accessibility. In this thesis, we introduce the AlgorithmAnalysis.jl Julia package, a Lyapunov function-based framework for robust algorithm analysis accessible to a broad user base.

Contents

List of Figures

Dedication

I would like to dedicate this thesis to my family and close friends.

1

Introduction

Optimization problems can be described as trying to find the optimal solution while being under a set of constraints. Many problems that exist in the field of engineering and natural science can be categorized as optimization problems. A simple example might be mapping applications, which use an algorithm to find the shortest path between two points — minimizing or optimizing the distance traveled — while under constraints such as traffic laws or avoiding road work.

Gradient-based iterative algorithms are a tool to optimize large optimization problems. Their ability to efficiently optimize functions makes them extensively used in fields such as machine learning and data science [1]. There exist many different types of problems and many more methods that can be used to optimize them. Consequently, the ability to measure and compare the performance of algorithms can be helpful. It allows us to find the best-performing algorithm for any given problem category, improving efficiency and saving computation resources. As a result, substantial research has been conducted to quantify the performance of algorithms either through empirical evidence or mathematical proof.

Algorithm analysis is a field that seeks to prove a performance guarantee of an algorithm for solving a set of optimization problems. There are two approaches in the field, both of which arrive at the goal by solving an optimization problem and proving a performance rate mathematically. As a result, applying one of the methods in algorithm analysis often requires extensive knowledge of the field.

The main work of this thesis is the development of `AlgorithmAnalysis.jl`, a computer program written in the Julia programming language that applies the Lyapunov function-based approach to analyzing gradient-based algorithms. The package can automatically find the worst-case performance guarantee of an algorithm for a specified set of problems, making the analysis of iterative gradient-based methods accessible to a broad user base.

After the program is given a class of functions and the algorithm to be analyzed, it returns a guaranteed rate at which the algorithm can optimize any function in the set.

1.1 Optimization problems and algorithms

In this thesis, the optimization problem considered is in the form of finding the minimum point of a differentiable function:

$$\begin{aligned} \text{minimize} \quad & f(x) \\ \text{subject to} \quad & x \in X \end{aligned}$$

where $f(x)$ is the optimization function and X is a constraint set. Here, x is the input or decision/optimization variable, and $f(x)$ is a measure of the cost or loss associated with each candidate solution x. Well-known examples of this problem are present in the training of large language models (LLMs) such as ChatGPT or the machine learning models that enable self-driving features in automobiles. An integral part of the training process of these models, through which they are created and continuously improved, is the minimizing of loss functions. In this process, a function is used to quantify the dissimilarity between a model's output and the desired values, and the model's parameters are modified iteratively using an algorithm in order to minimize the function and improve the model's performance.

While traversing any function can give its minimum, for large-scale and complex problems, it is more efficient to optimize functions numerically using iterative gradient-based algorithms. These algorithms minimize a function by starting at an initial point x_0 and iteratively updating an estimate x_k (k representing the current iteration number), using the gradient of the function at each iteration $\nabla f(x_k)$ until it reaches a local minimum x_s. For example, the gradient descent (GD) algorithm updates x_k following this formula:

$$x_{k+1} = x_k - \alpha \nabla f(x_k) \tag{1.2}$$

where α is the step size, an adjustable algorithm parameter. The stepsize α can affect the speed at which the algorithm converges, or whether it converges at all. Following this update formula, in each iteration, x moves toward the goal x_s where the gradient is zero $\nabla f(x_s) = 0$ and stays there afterward. Two areas where an algorithm like gradient descent can be improved are the number of iterations until the goal is reached, or the problem of overshooting, where the goal is not reached within an acceptable margin due to a step

size too large. Accelerated algorithms exist that seek to overcome these problems, such as Polyak's Heavy Ball (HB) method which introduces a momentum that incorporates previous iterations of x [2]:

$$x_{k+1} = x_k - \alpha \nabla f(x_k) + \beta(x_k - x_{k-1}) \tag{1.3}$$

where β is another stepsize parameter, while Nesterov's Fast Gradient (FG) method evaluates the gradient at an interpolated point [3]:

$$x_{k+1} = x_k - \alpha \nabla f(y_k), \tag{1.4a}$$

$$y_{k+1} = x_{k+1} + \beta(x_{k+1} - x_k) \tag{1.4b}$$

In the rest of this thesis, we will use these three examples of iterative gradient-based algorithms to introduce the concept of algorithm analysis, the Lyapunov-based method, and how it is implemented.

1.2 Algorithm analysis

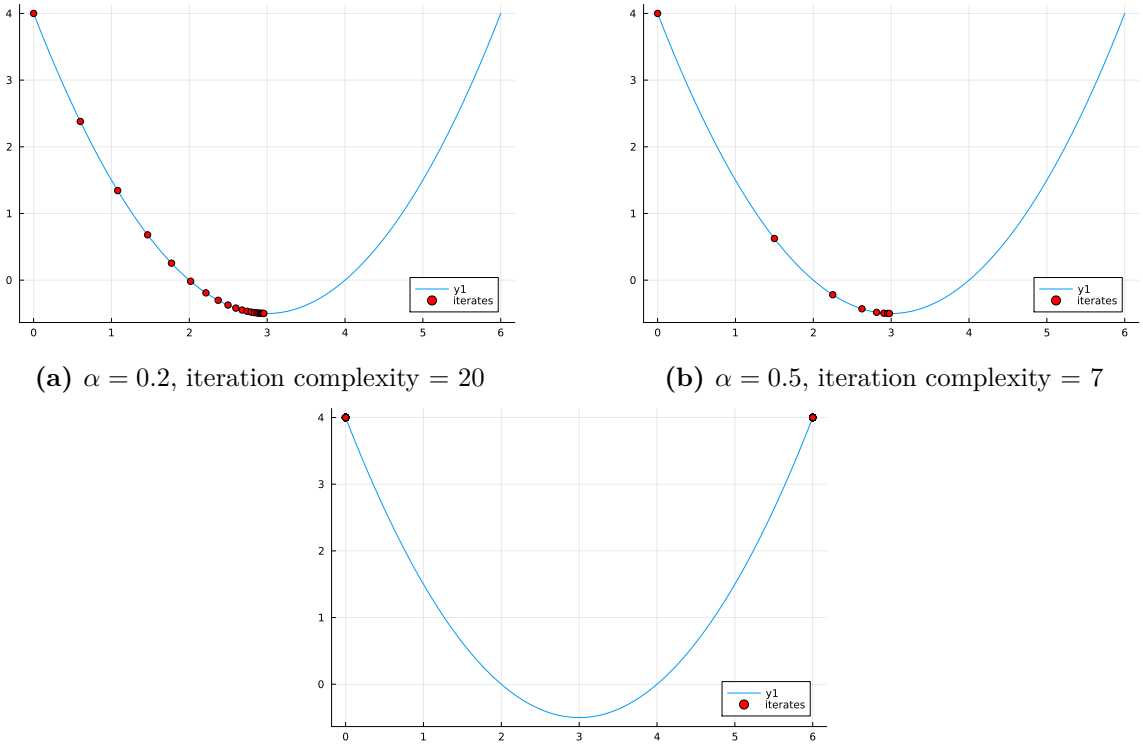
Let us consider the problem of minimizing a simple quadratic function with no constraint:

$$f(x) = x^2/2 - 3x + 4 \tag{1.5}$$

Using the (GD) equation (1.2), substituting step size α with values 0.2, 0.5, and 2, and picking a starting point of $x_0 = 0$, we will try to find the minimizer of this quadratic function. By counting the number of iterations each variation runs for before reaching 0.001 of the true minimum, we can measure the iteration complexity in Fig. 1.1.

It can be seen how different tunings on the same algorithm can achieve drastically different speeds optimizing a function, or whether it can find the minimizer at all. Considering there exist many other first-order methods in addition to the three in Section 1.1, each having infinitely adjustable parameters, finding the best algorithm for any problem set will mean it can optimized more quickly and more accurately.

While the analysis example in Fig. 1.1 yielded an analysis of the algorithms' performance, it required solving the optimization problem. Not only would solving any problem large enough to warrant being optimized numerically in the first place be computationally expensive, but any benefit of finding an algorithm superior at solving the problem is negated as it has already been solved. Additionally, any analysis result is applicable only to one function and



(a) $\alpha = 0.2$, iteration complexity = 20

(b) $\alpha = 0.5$, iteration complexity = 7

(c) $\alpha = 2$, does not converge

Figure 1.1: Performance of 3 GD variants of different step sizes solving a quadratic function

cannot be reliably used to derive a first-order method's performance on any other problem.

In the application of training large language and self-driving models, the training process is continuous as more training data becomes available. This training uses vast amounts of time and computational power, and as a result, even a small improvement in the performance of the algorithm used can speed up the training process while reducing energy usage. However, as the problem becomes larger and more complex, so does the challenge of finding a better algorithm. It is more efficient therefore to analyze algorithms' performance at solving a broad set of problems.

As a result of their widespread application, popular iterative gradient-based algorithms have been extensively analyzed. A frequently cited attempt is the Adam algorithm [4]. Its author quantified and compared the performance of algorithms using experiments and empirical evidence. There exists a different approach, which measures the performance of an algorithm by mathematically proving a performance bound. This worst-case analysis is referred to as algorithm analysis: Given a characteristic that a set of functions might share (such as being convex or quadratic), it would return the worst-case performance measure that guarantees the algorithm analyzed would perform as well as or better at solving every function within

said set.

1.3 Julia programming language

Our work uses the Julia programming language, a high-level programming language designed specifically for high-performance numerical computing. Julia's compiler performance has been benchmarked to be faster than many other languages used for numerical computing while being on par with C, a language often used for its high efficiency [5]. Julia accomplishes this while being a high-level language with simple syntax rules that resemble existing popular languages, making it easy to develop, use, and understand.

Julia is open-source and available for free on many popular platforms such as macOS, Windows, and Linux, making it a good choice for the `AlgorithmAnalysis.jl` package as it is designed with expert and novice users alike in mind.

Julia is also chosen as it is designed for numerical computing, supporting matrices as well as UTF-8 encoding, making it possible to use scientific notation: variables and functions as they exist in the code and as the user inputs them into the program can use math symbols or Greek letters. This makes Julia excel at communicating mathematical concepts, which simplifies both the process of coding the program and understanding its mathematical underpinnings. Fig. 1.2 shows sample code of how the package can be used, while Chapter 4 explains our work's core code components.

1.4 Analysis example

To analyze the performance of an algorithm using the `AlgorithmAnalysis.jl` package, the user needs to follow the following 3 steps:

1. Choose from a supported list the class of function to be optimized.
2. Define an algorithm to be analyzed .
3. Specify a performance measure.

Our package defines a function class as every function that shares a trait. In example Fig. 1.2, the class of function is m smooth—L strongly convex functions. Our work shares the notation and definition of smooth strongly convex functions with [6], which uses the notation $F_{m,L}$ and defines the function class as any continuously differentiable functions that satisfy:

1. L-Lipschitz gradients: $\|\nabla f(x) - \nabla f(x)\| \leq L\|x - y\|$ for all $x, y \in \mathbb{R}$.
2. m-strong convexity: $f(x) - \frac{m}{2}\|x\|^2$ is convex.

```
m,L = 1,10
α = 2/(L+m)
@algorithm begin

        f = DifferentiableFunctional{Rⁿ}()
        xs = first_order_stationary_point(f)
        f ∈ SmoothStronglyConvex(m, L)

        x0 = Rⁿ()
        x1 = x0 - α*f'(x0)

        x0 => x1

        performance = (x0-xs)^2
end

@show rate(performance)
```

Figure 1.2: Analysis example

In the example code, the user:

- Defines the class of function `f` and its gradient ∇f, coded as `(f')` by calling the provided functions `DifferentiableFunctional` and `SmoothStronglyConvex`.
- Defines an initial state x_0 and how the algorithm creates the next state x_1. In this example, the algorithm being analyzed is gradient descent with a step size $\alpha = 2/11$.
- Sets the performance measure as the norm distance between the initial state and the goal $(x_0 - x_s)^2$. The returned convergence rate guarantee is the rate at which the performance measure decreases after each iteration of the algorithm.
- Call the rate function to start the analysis.

With the calling of the rate function, the program runs automatically to return a rate of 0.8176803588867188. This is the convergence rate guarantee ρ such that, for some performance measure $c > 0$, it is upper bounded by $c\rho^k$ at each iteration k, for the provided algorithm and every function in the class.

This guaranteed convergence rate using "big O" notation means that the performance mea-

sure converges with a minimum rate of $O(\rho^k)$. Throughout the process, the user never has to modify the package beyond providing its input or understand how the package `AlgorithmAnalysis.jl` operates.

```
rate(performance) = 0.8176803588867188
```

Figure 1.3: Analysis result

1.5 Overview

After the introduction, in Chapter 2, the existing literature on analyzing algorithms is presented. Chapter 3 discusses the Lyapunov-based mathematical method that `Algorithm Analysis.jl` utilizes. Chapter 4 demonstrates the domain-specific language developed and how it supports the package's functionality. Chapter 5 details the analysis process. Analysis results produced by the package is presented in Chapter 6. Finally, discussion about the package and potential future work and our conclusion is presented in Chapter 7.

2

Literature Review

In recent years, numerous studies have compared the performance of optimization algorithms, particularly in machine learning and deep learning applications. These comparisons typically evaluate convergence speed, accuracy, and robustness across diverse models and datasets. In [4], the AdaM (Adaptive Moment Estimation) algorithm was introduced, combining the advantages of two popular methods: AdaGrad and RMSProp. The authors demonstrated Adam's efficiency over existing algorithms through extensive experiments on various machine learning tasks, including training deep neural networks, where it showed faster convergence and better performance.

Despite its empirical successes, however, subsequent work in [7] revealed that Adam may fail to converge on simple quadratic problems under certain conditions. These findings highlight the importance of complementing empirical evaluations with rigorous theoretical analyses to fully understand the behavior of optimization algorithms. They demonstrate the benefit of robust analytical frameworks, such as the one proposed in this thesis, to provide deeper theoretical guarantees and enhance the reliability of algorithmic performance assessments.

Among the various approaches developed for automated algorithm analysis aimed at guaranteeing minimum performance criteria, one of the most influential frameworks is the Performance Estimation Problem (PEP). Initially proposed by Drori and Teboulle in 2014, the PEP framework introduced a systematic method to evaluate the worst-case performance of optimization algorithms by translating performance analysis into an optimization problem itself, specifically a semidefinite program (SDP) [8]. By modeling a given class of optimization problems using constraints, the authors reformulated the analysis task as a convex optimization problem. This enabled the computation of exact or numerically tight worst-case performance guarantees by solving an SDP numerically, which was pioneering in providing robust theoretical performance guarantees. The shift away from manually derived analytical

proofs toward automated computational methods is also significant, as it allows the use of computers in the process of deriving algorithms's worst-case performance bounds.

Subsequent contributions significantly expanded upon this foundational approach. Taylor, Hendrickx, and Glineur further developed the PEP methodology by introducing closed-form necessary and sufficient interpolation conditions for certain function classes, such as smooth strongly convex functions, thereby providing a finite representation of these function classes in the SDP [9].

Moreover, their work culminated in the creation of PEPit, an open-source Python-based software package designed specifically as a high-level, streamlined interface for formulating and solving performance estimation problems. PEPit allows users to specify an algorithm and its target function class using a simple, declarative syntax. Internally, it automatically constructs the corresponding semidefinite program using the PEP method and solves it using an appropriate SDP solver. This automation significantly reduces the technical barrier to applying PEP, enabling researchers and developers to obtain certified performance bounds without manually deriving interpolation conditions or constraint formulations. It also comes with built-in support for numerous standard algorithms (such as gradient descent, fast gradient, and proximal methods) and common function classes, including smooth convex, strongly convex, and composite objective structures. This allows users to test and compare different methods. This ease of use and versatility or PEPit have significantly contributed to its prominence as a tool in both academic research and practical algorithm development.

Both the original formulation by Drori and Teboulle and the subsequent refinements implemented in PEPit share their motivations and techniques with Lyapunov function-based analysis. Specifically, both approaches rely on constructing structured semidefinite programming formulations to rigorously quantify worst-case algorithm performance. However, the Lyapunov approach differs in that it uses an energy-based decreasing Lyapunov function to certify whether a performance level, defined as the convergence rate of the algorithm, can be guaranteed.

In [10], Megretski and Rantzer demonstrated how integral quadratic constraints (IQCs) can be used to unify and simplify the analysis of system stability and performance. The paper introduced the idea that optimization algorithms can be interpreted as a dynamical system in which the gradient of the objective function — a complex system — can be represented with IQCs. This idea leverages tools from *robust control* similar to the Lyapunov-based approach we are implementing. Through this interpretation, the problem of analyzing algorithms' performance is transformed into a robust control problem, and analyzing the

performance of the optimization algorithm is equivalent to analyzing the stability of the corresponding dynamical system. The first paper to apply IQCs from robust control to analyze the performance of optimization algorithms is [11], a highly influential paper in this area. While both [11] and Drori and Teboulle's PEP method derive an SDP the solution of which guarantees precise bounds on the convergence rate of first-order algorithms, a major drawback of the PEP method was that the SDP scales in size with the number of iterations the algorithm is run, increasing the computational cost and time it requires so perform analysis. In [11], authors Lessard, Recht, and Packard demonstrate that by using Lyapunov functions, a bound on an algorithm's convergence rate which holds for all iterations can be found by deriving a small and fixed-size SDP.

In [6], the authors Bryan Van Scoy and Laurent Lessard present a Lyapunov-function-based approach to analysis, which transforms the analysis problem into a robust control problem, similar to the IQC method, and uses interpolation conditions to describe the complex system that is the gradient of the smooth strongly convex function class, similar to the PEP method. The method then forms a convex optimization problem of finding a Lyapunov function that proves the algorithm converges at a certain rate, the feasibility of which establishes whether a convergence rate can be guaranteed for the given algorithm and problem class. The problem of finding the fastest guaranteed rate is then to simply search over convergence rates between zero and one for the fastest one that can be guaranteed. The Lyapunov function-based approach used by the `AlgorithmAnalysis.jl` package modifies how the Lyapunov function is formed compared to the approach presented in [6] to better facilitate its implementation into a software program, and will be discussed in Chapter 3.

While [6] and this thesis focus on the optimization of unconstrained first-order methods, another work in the field of automatic analysis of optimization algorithms using Lyapunov function is the work in [12], which focuses on linearly constrained optimization problems. These problems are in the form of:

$$\begin{aligned} \text{minimize} \quad & f(x) && \text{(2.1a)} \\ \text{subject to} \quad & Ax = b && \text{(2.1b)} \end{aligned}$$

In [12], the author defines A in (2.1) as an abstract matrix whose singular values are bound by an upper and lower value. The authors proposed a framework for the automatic analysis of primal-dual algorithms [13] used to solve linearly constrained convex optimization problems. The primal dual algorithm differs from those used for unconstrained problems in

that it uses a transformation based on a compact singular value decomposition (SVD) of the constraint matrix A. This allows the algorithm's dynamics to be separated into components that are affected and unaffected by the constraints, simplifying the structure of the problem and enabling a more systematic application of Lyapunov functions to certify convergence guarantees. While the separation of the state iterate and the introduction of another nonlinearity makes the analysis of constrained problems different from the unconstrained case, the overall methodology is similar to the Lyapunov-based framework presented in [6] in its use of interpolation conditions and matrix inequalities to automatically derive worst-case performance bounds. This work demonstrates that the Lyapunov function-based approach can perform automated analysis for constrained as well as unconstrained problems.

The approaches to automated algorithm analysis in the literature presented above can be broadly categorized into two main types, the optimization-based PEP approach, which was implemented into PEPit, and the control approach, which has not been implemented into a computer program. The main contribution of this thesis is the development of `AlgorithmAnalysis.jl`, aims to be a robust framework for the analysis of iterative first-order algorithms similar to PESTO and PEPit. However, it would apply the Lyapunov function-based approach instead of the PEP method.

We will also show that by developing it as a domain-specific language inside the Julia programming language, we enable users to describe optimization algorithms at a high level, providing a more intuitive way of defining the method to be analyzed compared to PEPit. By being a domain-specific language, the package would also allow a systematic representation of algorithmic components such as iterates, gradients, and function oracles, simplifying the understanding of both the analysis result and the Lyapunov-based approach compared to PEPit. The package does this while similar to PEPit automating the generation of interpolation conditions and every other component needed to derive a performance guarantee. This means `AlgorithmAnalysis.jl` would require the user only to define the algorithm they wish to analyze and the function class, lowering the barrier to entry for non-experts and facilitating rapid experimentation in algorithm design.

3

Lyapunov-based Approach

`AlgorithmAnalysis.jl` performs analysis by modifying the technique layed out by Van Scoy and Lessard in [6]. While the package is a blackbox tool, understanding the mathematical approach on which it is based is a prerequisite to understanding its code and functionalities.

The Lyapunov-based approach builds on the key idea that verifying whether an algorithm achieves a certain convergence rate can itself be formulated as an optimization problem. It formulates and solves a problem to find a mathematical certification of a performance rate, a Lyapunov function that decreases with each step of the algorithm. This chapter introduces the approach through the following key components:

- Modeling the algorithm as a dynamical system, borrowing ideas from control theory to describe how the algorithm updates its state over time.
- Characterizing the function class using interpolation conditions, which define constraints on an algorithm iterate and its gradient.
- Formulating an optimization problem to search for a Lyapunov function that proves whether the algorithm guarantees the desired convergence rate.

3.1 Iterative algorithms as Lur'e problems

The first step in the technique is to view optimization algorithms from a control theory perspective. Iterative gradient-based algorithms use the gradient of the function to update an iterate or state — represented by x_k in equations (1.2), (1.3), and (1.4). In our approach, we reformulate these algorithms into a linear time-invariant (LTI) system (how the algorithm updates) in feedback with a static nonlinearity (the gradient of f) taken at iterate x or some linear combination of the iterates. Fig. 3.1 shows the block diagram of this view.

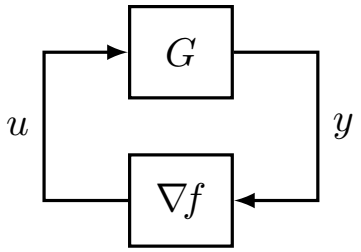


Figure 3.1: Block diagram representation of iterative algorithms

In Fig. 3.1, G represents the LTI system, while y and u are input and output of the gradient nonlinearity ∇f. For example, (FG) equation (1.4) matches this representation if u_k is defined as $\nabla f(y_k)$. can be rewritten to match this view as:

$$x_{k+1} = x_k - \alpha u_k, \tag{3.1a}$$
$$y_{k+1} = x_{k+1} + \beta(x_{k+1} - x_k), , \tag{3.1b}$$
$$u_k = \nabla f(y_k) \tag{3.1c}$$

The algorithm can then be put into state-space representation with augmented state $\xi_k = (x_k, x_{k-1})$ as:

$$\xi_{k+1} = \begin{bmatrix} (1+\beta) & -\beta \\ 1 & 0 \end{bmatrix} \xi_k + \begin{bmatrix} -\alpha \\ 0 \end{bmatrix} u_k, \tag{3.2a}$$
$$y_k = \begin{bmatrix} 1+\beta & \beta \end{bmatrix} \xi_k, , \tag{3.2b}$$
$$u_k = \nabla f(y_k) \tag{3.2c}$$

In (3.2), function f is n-multivariate, meaning $x_k \in \mathbb{R}^{1\times n}$, $\xi_k \in \mathbb{R}^{2\times n}$, $y_k \in \mathbb{R}^{2\times n}$, $u_k \in \mathbb{R}^{2\times n}$, and the gradient evaluation maps a row vector to a row vector $\nabla f : \mathbb{R}^{1\times n} \to \mathbb{R}^{1\times n}$. The Lyapunov-based approach defines every function as n-multivariate so that the analysis result is applicable to function f of any number of variables.

The LTI system G can be expressed with four matrices that change in value depending on the algorithm and size depending on the number of past states used to update x. For (GD), (FG), and (HB) as they are described in equations (1.2), (1.3), and (1.4), these matrices are:

Beyond the three listed examples, any other first-order methods for optimizing unconstrained problems can be transformed into an LTI system.

GD	HB	FG
$\left[\begin{array}{c\|c} 1 & -\alpha \\ \hline 1 & 0 \end{array}\right]$	$\left[\begin{array}{cc\|c} 1+\beta & -\beta & -\alpha \\ 1 & 0 & 0 \\ \hline 1 & 0 & 0 \end{array}\right]$	$\left[\begin{array}{cc\|c} 1+\beta & -\beta & -\alpha \\ 1 & 0 & 0 \\ \hline 1+\beta & -\beta & 0 \end{array}\right]$

3.2 Interpolation condition

In the field of robust control theory utilized by this Lyapunov-based approach, while an LTI system is relatively simple, the nonlinearity ∇f representing the gradient of the function cannot be efficiently solved. The Lyapunov function-based approach overcomes this by replacing the nonlinearity with a characterization of the class of function. Since the algorithm is being analyzed at discrete iterates, the characterization of a function class can be done using *interpolation conditions*, a set of conditions on the nonlinearity's input y and output u. Through this characterization, the block diagram representation is transformed into the block diagram depicted in Fig. 3.2.

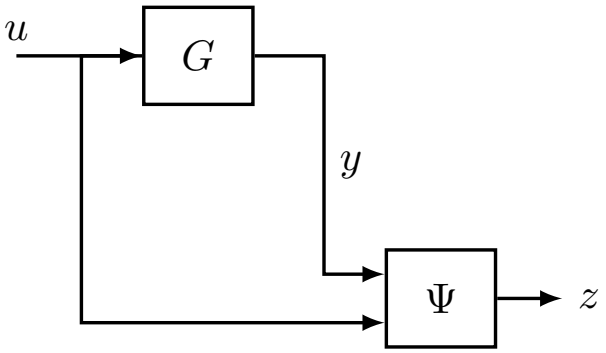


Figure 3.2: Block diagram of iterative algorithms with gradient replaced by interpolation condition filter

In Fig. 3.2, the nonlinear gradient block is replaced with a filter or dynamical system Ψ. This dynamical system produces an output z as a function quadratic in the algorithm states from its inputs y and u, before applying certain constraints to the output z.

Both the filter and the constraints applied on its output depend on the properties of the gradient of the objective function, which are characterized by its interpolation conditions. The interpolation conditions of a function class provide necessary and sufficient conditions under which there exists a function in that class that interpolates a given finite set of iterate-gradient-function value triplets. The analysis of any algorithm's performance at solving a function class using the Lyapunov-based approach is only possible if there exist interpolation

conditions for that class.

While many function classes have interpolation conditions, our work focuses on smooth strongly convex functions and their interpolation conditions, as many optimization problems such as linear regression or logistic regression in the machine learning field can be classified as convex optimization problems. The interpolation conditions for L-smooth and m-strongly convex functions was first formulated in [14] and reformatted in [6] as:

Theorem 1 ([14], Thm. 4; [6], Thm. 3)**.** Given index set I, a set of triplets $\{(y_k, u_k, f_k)\}_{k\in I}$ is $F_{m,L}$-interpolable, meaning there exists a function $f \in F_{m,L}$ satisfying $f(y_k) = f_k$ and $\nabla f(y_k) = u_k$ if and only if

$$2(L-m)(f_i - f_j) - mL\|y_i - y_j\|^2 + 2(y_i - y_j)^\mathsf{T}(mu_i - Lu_j) - \|u_i - u_j\|^2 \geq 0$$

for all $i, j \in I$.

In the analysis of the gradient descent algorithm for analyzing 1 strong 10 smooth convex function ($f \in F_{1,10}$) shown in Fig. 1.2, we define `x0` as the initial iterate of an algorithm optimization and `xs` as the minimizer. The gradient of the function `f` is evaluated at `x0` and `xs`. Using x_0 to represent `x0` and x_s to represent `xs` the set of triplets we are interpolating would be $(x_0, \nabla f(x_0), f(x_0)), (x_s, \nabla f(x_s), f(x_s))$. Apply this to Theorem 1 and consider that the gradient at the minimizer is zero $\nabla f(x_s) = 0$, then the inequalities:

$$\begin{aligned} 18(f(x_0) - f(x_s)) - 10\|x_0\|^2 - 10\|x_s\|^2 + 20\langle x_0, x_s\rangle + 2\langle \nabla f(x_0), x_0\rangle \\ -2\langle \nabla f(x_0), x_s\rangle - \|\nabla f(x_0)\|^2 \geq 0 \\ 18(f(x_s) - f(x_0)) - 10\|x_0\|^2 - 10\|x_s\|^2 + 20\langle x_s, x_0\rangle + 20\langle \nabla f(x_0), x_0\rangle \\ -20\langle \nabla f(x_0), x_s\rangle - \|\nabla f(x_0)\|^2 \geq 0 \end{aligned} \tag{3.3}$$

when satisfied interpolates 1 strong 10 smooth convex functions.

This approach of viewing an algorithm as a control system and replacing one or more of its elements with constraints is not limited to function classes. We can apply this method to any other abstract elements of an algorithm as long as it has a proven set of interpolation conditions. An example can be found in the analysis approach of the primal-dual algorithm [13], meant for optimizing constrained optimization problems. To analyze the primal-dual using Lyapunov functions, an approach presented in [12], the author uses interpolation conditions not only to characterize the gradient of smooth strongly convex functions similar to our Lyapunov approach but also the abstract matrix that forms the constraint on the problem.

Gram matrix interpolation conditions

It can be seen that other than the function value f, the left-hand side of the interpolation conditions in Theorem 1 consists of the norms and inner products of the interpolated points and gradients y and u. These elements form the Gram matrix — defined as a matrix whose elements are the inner products of a set of vectors — of the vector of interpolated points. Theorem 1 can be transformed into:

$$\sum_{i,j\in I} \operatorname{tr}\left(\begin{bmatrix} -mL & mL & m \\ mL & -mL & -m \\ m & -m & -1 \end{bmatrix} \begin{bmatrix} \|y_i\|^2 & \langle y_j, y_i\rangle & \langle u_i, y_i\rangle \\ \langle y_i, y_j\rangle & \|y_j\|^2 & \langle u_i, y_j\rangle \\ \langle y_i, u_i\rangle & \langle y_j, u_i\rangle & \|u_i\|^2 \end{bmatrix}\right) + 2(L-m)(f_i - f_j) \geq 0. \quad (3.4)$$

for all $i, j \in I$.

The elements of the Gram matrix are interpolable — or in other words they can be the inner products and norms of the interpolated points y and u — if and only if:

1. The dimension n of the state vector x_k is greater than or equal to the rank of the Gram matrix.
2. The Gram matrix is positive semidefinite.

Since the problem classes the algorithm being analyzed is abstract, the Lyapunov-based approach makes the assumption that each function and its corresponding state vectors have a sufficiently large dimension. This satisfies the condition on rank size, while the other condition on the Gram matrix will be used to construct the optimization problem.

The constraint for the Gram matrix associated with (3.3) is presented as:

$$\begin{bmatrix} \|x_k\|^2 & \langle x_s, x_k\rangle & \langle \nabla f(x_k), x_k\rangle \\ \langle x_k, x_s\rangle & \|x_s\|^2 & \langle \nabla f(x_k), x_s\rangle \\ \langle x_k, \nabla f(x_k)\rangle & \langle xs, \nabla f(x_k)\rangle & \|\nabla f(x_k)\|^2 \end{bmatrix} \succcurlyeq 0. \quad (3.5)$$

For the analysis of GD, FG, and HB, which creates the norms and inner-products of state vectors in the same vectorspace, one Gram matrix is created and constrained. As the gradient of the function is taken at additional state vectors and additional constraints are created, the size of the Gram matrix is increased.

3.3 Lyapunov function derivation

In the third and final step of the Lyapunov method, we:

1. Use Lyapunov functions to represent the energy of the system.
2. Apply conditions on the Lyapunov functions, whose satisfaction proves whether a performance rate can be guaranteed for the system.
3. Formulate an optimization problem consisting of functions linear in the optimization variables using constraints on the Lyapunov functions and those created by the interpolation conditions. For any rate of performance, whether it can be guaranteed for the system depends on whether the optimization problem can be solved.

The Lyapunov-based analyzes an algorithm's performance by certifying whether any given level of performance can be guaranteed, and uses *convergence rate* as a measure of performance.

Convergence rate is defined as the rate at which a performance measure decreases after each iteration. For example, given a performance measure which is the Euclidean norm distance between the iterate of the algorithm and the minimizer $\|x_k - x_s\|^2$ at any iterate k of the algorithm, a proven convergence rate guarantee of ρ means:

$$\|x_k - x_s\|^2 \leq \rho^k \|x_0 - x_s\|^2. \tag{3.6}$$

Following this definition, a convergence rate value of 1 means the algorithm cannot be guaranteed to converge while a convergence rate of 0 means the algorithm is guaranteed to converge after a finite number of iterations. The smaller the convergence rate, the better the algorithm performs.

In the field of control, the Lyapunov function is a fundamental tool, defined as a nonnegative function that decreases in time along the orbit of a dynamical system. It can be used to understand the behavior of a system and represent its energy. Under this definition, the dynamical system is the algorithm being analyzed, and two constraints on a Lyapunov function are used to certify whether or not a convergence rate can be guaranteed for a system. Consider the GD algorithm and its representation in (1.2), define $\mathbf{x_k} = x_k - x_s$, the Lyapunov function takes the form:

$$V(\mathbf{x}) = \mathrm{tr}(\mathbf{x_k}^T P \mathbf{x_k}) \tag{3.7}$$

where P is a symmetric matrix optimization variable. Note that the Lyapunov function represents a state of a system, in this case, an algorithm, and for algorithms that update its iterate using multiple past states, $\mathbf{x_k}$ would have to include every state used to iterate. For example, for the Fast gradient algorithm as it is represented in (1.4), $\mathbf{x_k} = \begin{bmatrix} x_k - x_s \\ x_{k-1} - x_s \end{bmatrix}$. Continuing the GD example, if it is proven there exists some optimization variable P so that Lyapunov functions satisfy the conditions:

$$\|x_k - x_s\|^2 - V(\mathbf{x_k}) \leq 0, \tag{3.8a}$$

$$V(\mathbf{x_{k+1}}) - \rho^2 V(\mathbf{x_k}) \leq 0 \tag{3.8b}$$

then the convergence rate of that algorithm is guaranteed to be faster than ρ^2 for every function in the function class.

The first Lyapunov function inequality if satisfied guarantees for each iteration of an algorithm, the distance from the iterate to the minimizer $\|x_k - x_s\|^2$ is smaller or equal to the Lyapunov function. The second Lyapunov function inequality if satisfied guarantees after each iteration, the Lyapunov function decreases at a rate faster or equal to ρ^2. By satisfying these conditions we can certify a convergence rate of ρ^2. This was proven in Lemma 5 of [6], which can be modified to suit the GD algorithm. For any $k \geq 0$:

$$\|x_k - x_s\|^2 \leq V(\mathbf{x_k}) \leq \ldots \leq \rho^k V(\mathbf{x_0}) \leq \rho^k (C_0 \|x_0 - x_s\|^2) \tag{3.9}$$

where C_0 is a constant that depends on the intialization of the algorithm and the optimization parameter P.

Linear transformation

The `AlgorithmAnalysis` package certifies the inequalities in (3.8) by representing each variable in it and the constraints created by the interpolation conditions with an optimization variable. These constraints along with those created by the interpolation conditions form an optimization problem: Find the variables with which every constraint of the problem is satisfied.

The expressions constrained by interpolation conditions and the Lyapunov function inequalities have to be linearized to be parameterized with optimization variables. While the interpolation conditions in Theorem 1 are linear in the function values f and in the inner products and norms of state vectors in (3.4), the Lyapunov functions are quadratic in the

state vectors. Therefore, we deviate from the approach in [6] by transforming the Lyapunov functions to also be linear in the inner products and norms of state vectors. Chapter 5 will then show how these inner products, norms, and function values are represented by optimization variables in the optimization problem.

It can be seen from (3.7) that while these Lyapunov functions are quadratic in $\mathbf{x_k}$, they are linear in the Gram matrix of the state and input vectors $[x_k, x_s, u_k]$. Define $I_n \in \mathbb{R}^{n\times n}$ as the identity matrix with dimension n, the same as the dimension of the state vectors, the Lyapunov function can be transformed into:

$$\begin{aligned} V(\mathbf{x_k}) &= \mathrm{tr}[P(x_k - x_s)(x_k - x_s)^\mathsf{T}] \qquad (3.10a)\\ &= \mathrm{tr}[\begin{bmatrix} P & -P & 0\\ -P & P & 0\\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \|x_k\|^2 & \langle x_s, x_k\rangle & \langle u_k, x_k\rangle\\ \langle x_k, x_s\rangle & \|x_s\|^2 & \langle u_k, x_s\rangle\\ \langle x_k, u_k\rangle & \langle x_s, u_k\rangle & \|u_k\|^2 \end{bmatrix}]\\ V(\mathbf{x_{k+1}}) &= \mathrm{tr}[P(x_{k+1} - x_s)(x_{k+1} - x_s)^\mathsf{T}] \qquad (3.10b)\\ &= \mathrm{tr}[P(Ax_k + Bu_k - x_s)(Ax_k + Bu_k - x_s)^\mathsf{T}] \end{aligned}$$

After applying the state space representation of our algorithm, we can take advantage of the cyclic property of the trace operator, which allows us to rearrange the order of multiplied matrices inside the trace, to perform the following transformations:

$$\begin{aligned} &= \mathrm{tr}[P\begin{bmatrix} A & B & -I_n \end{bmatrix} \begin{bmatrix} x_k\\ u_k\\ x_s \end{bmatrix} \begin{bmatrix} x_k\\ u_k\\ x_s \end{bmatrix}^\mathsf{T} \begin{bmatrix} A & B & -I_n \end{bmatrix}^\mathsf{T}]\\ &= \mathrm{tr}[\begin{bmatrix} A & B & -I_n \end{bmatrix} P \begin{bmatrix} A & B & -I_n \end{bmatrix}^\mathsf{T} \begin{bmatrix} \|x_k\|^2 & \langle x_s, x_k\rangle & \langle u_k, x_k\rangle\\ \langle x_k, x_s\rangle & \|x_s\|^2 & \langle u_k, x_s\rangle\\ \langle x_k, u_k\rangle & \langle x_s, u_k\rangle & \|u_k\|^2 \end{bmatrix}] \end{aligned}$$

while the performance measure $\|x_k - x_s\|^2$ can be transformed into:

$$\|x_k - x_s\|^2 = \mathrm{tr}[\begin{bmatrix} 1 & -1 & 0\\ -1 & 1 & 0\\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \|x_k\|^2 & \langle x_s, x_k\rangle & \langle u_k, x_k\rangle\\ \langle x_k, x_s\rangle & \|x_s\|^2 & \langle u_k, x_s\rangle\\ \langle x_k, u_k\rangle & \langle x_s, u_k\rangle & \|u_k\|^2 \end{bmatrix}]. \qquad (3.11)$$

The Lyapunov functions $V(\mathbf{x_k})$ and $V(\mathbf{x_{k+1}})$, as well as the performance measure $\|x_k - x_s\|^2$ are now linear functions of the Gram matrix and optimization variable P. This holds for every iterative gradient-based algorithm as they update using a linear function of past states

x and inputs u. If a different algorithm uses multiple iterates or multiple gradients to update, the corresponding Gram matrix of the Lyapunov functions will simply grow to include the additional states and inputs.

Here, it should be noted that the Gram matrices in the Lyapunov functions are the same as those in the constraints, as every state and input vector whose norm and inner product form the Gram matrix has to be interpolated. However, each constraint includes the function values f_i, f_j on top of the elements of the Gram matrix present in the Lyapunov conditions. In order to combine the conditions in (3.8) with the constraints associated with the interpolation conditions, we first define the *linear form* of both.

Define $[x; u]$ as a vector containing every real element of the Gram matrix and the function values at each interpolated point. It can be seen from (3.3) and (3.10) that the Lyapunov function and each constraint are linear functions of $[x; u]$. In the GD example, this vector is defined as:

$$\begin{bmatrix} x \\ u \end{bmatrix} = \begin{bmatrix} \|x_k\|^2 \\ \langle x_s, x_k \rangle \\ \langle u_k, x_k \rangle \\ \langle x_k, x_s \rangle \\ \|x_s\|^2 \\ \langle u_k, x_s \rangle \\ \langle x_k, u_k \rangle \\ \langle x_s, u_k \rangle \\ \|u_k\|^2 \\ f(x_k) \\ f(x_s) \end{bmatrix}. \tag{3.12}$$

The linear form of conditions on the Lyapunov functions L_1 and L_2 can be defined as:

$$\|x_k - x_s\|^2 - V(\mathbf{x_k}) = L_1 \begin{bmatrix} x \\ u \end{bmatrix} \tag{3.13a}$$

$$V(\mathbf{x_{k+1}}) - \rho^2 V(\mathbf{x_k}) = L_2 \begin{bmatrix} x \\ u \end{bmatrix}. \tag{3.13b}$$

where L_1 and L_2 are matrices of real numbers. For the constraints associated with the interpolation conditions, each is derived and scaled by a nonnegative optimization variable Λ before being combined with L_1 and L_2. The linear form of the constraints associated with

the interpolation conditions (3.4) is:

$$\sum_{i \in C} M_i(\Lambda_i) \begin{bmatrix} x \\ u \end{bmatrix} \tag{3.14}$$

where C is the set containing every constraint created in (3.4) and M_i is the linear form of each constraint scaled by a corresponding parameter Λ_i. The optimization parameters Λ_i are constrained in the optimization problem to be nonnegative if it is a scalar, nonnegative element-wise if it is a vector, or positive semidefinite if it is a matrix. When the constraints applied to these variables are satisfied, constraints (3.14) are nonnegative. When solving the optimization problem of finding the Lyapunov function that certifies a performance rate, the solver searches for Λ_i just as it does for P for a set of values that satisfies the final inequality function.

If the constraint is applied to a scalar variable in the case of those created by interpolation conditions associated with the function class, M_i is the linear form of the scalar variable or the left-hand side of (3.3) in the GD example. On the other hand, if the constraint is applied to the Gram matrix, M_i is the linear form of its trace.

Lastly, as L_1, L_2 and $M_i(\Lambda_i)$ are all functions consisting of a vector or matrix of real-valued scalars and an optimization variable, we can combine them to create functions linear in the optimization variables, forming our optimization problem:

$$L_1 + \sum_{i \in C} M_i(\Lambda_i^1) = 0 \tag{3.15a}$$

$$L_2 + \sum_{i \in C} M_i(\Lambda_i^2) = 0. \tag{3.15b}$$

If for a convergence rate ρ^2, a combination of parameters P, Λ_i^1, and Λ_i^2 can be found which would satisfy the inequalities in (3.15), the Lyapunov function conditions specified in (3.8) would be satisfied and a performance of guarantee of ρ^2 is feasible. However, if no such parameter can be found, the Lyapunov function conditions are not satisfied and the convergence rate cannot be guaranteed to be feasible. This can be proven as, for any $[x; u]$, (3.15) can be multiplied with $[x; u]$ to derive the Lyapunov non-positive constraints in (3.8) added with the non-negative constraints in Theorem 1. If the linear functions (3.15) equal zero, and multiplying the linear form of the constraints by $[x; u]$ gives us the interpolation

conditions, which are constrained to be ≥ 0, it must mean that $L_1 \begin{bmatrix} x & u \end{bmatrix}^\mathsf{T}$ is ≤ 0:

$$(L_1 + \sum_{i \in C} M_i(\Lambda_i^1)) \begin{bmatrix} x & u \end{bmatrix}^\mathsf{T} = 0 \tag{3.16a}$$

$$\underbrace{L_1 \begin{bmatrix} x & u \end{bmatrix}^\mathsf{T}}_{\leq 0} + \underbrace{\sum_{i \in C} M_i(\Lambda_i^1) \begin{bmatrix} x & u \end{bmatrix}^\mathsf{T}}_{\geq 0} = 0 \tag{3.16b}$$

$$\tag{3.16c}$$

and similarly, $L_2 \begin{bmatrix} x & u \end{bmatrix}^\mathsf{T}$ is ≤ 0.

$$(L_2 + \sum_{i \in C} M_i(\Lambda_i^1)) \begin{bmatrix} x & u \end{bmatrix}^\mathsf{T} = 0$$

$$\underbrace{L_2 \begin{bmatrix} x & u \end{bmatrix}^\mathsf{T}}_{\leq 0} + \underbrace{\sum_{i \in C} M_i(\Lambda_i^1) \begin{bmatrix} x & u \end{bmatrix}^\mathsf{T}}_{\geq 0} = 0$$

.

Together, $L_1 \begin{bmatrix} x & u \end{bmatrix}^\mathsf{T} \leq 0$ and $L_2 \begin{bmatrix} x & u \end{bmatrix}^\mathsf{T} \leq 0$ form the conditions on the Lyapunov functions, and if (3.15) can be optimized, it means that the convergence rate is feasible.

Since we know how to certify whether a convergence rate guarantee is feasible, the fastest convergence rate that can be guaranteed for that system can be found by performing a bisection search for the smallest value ρ between 0 and 1 with which the optimization problem is feasible.

Lifted algorithm dynamics

In Section 3.2, it is shown that as additional triplets of state vectors, gradients, and function values are created, additional constraints are created. When we define $\mathbf{x_k} = \begin{bmatrix} x_k - x_s \\ x_{k-1} - x_s \end{bmatrix}$ for the analysis of FG instead of $\mathbf{x_k} = x_k - x_s$ for GD, an additional state iterate vector is created, whose norm and inner product with other state vectors increases the size of the Gram matrix. As a result, the number of variables that the Lyapunov function is parameterized by increases. In both cases, while the additional variables and constraints will make the problem more complex and slow down the analysis process, they can make the analysis result more precise. This is because when certifying a given convergence rate, a Lyapunov function might be found in a system with more optimization variables and constraints compared to

an unlifted system.

The Lyapunov function-based approach therefore introduces a lifting dynamic to the analysis. Here, lifting simply means to define an algorithm with more states than the required minimum, and the lifting dimension refers to how many total algorithm states there are. As additional state vectors are created, constraints based on interpolation conditions are created and applied in the optimization problem. The Lyapunov function would also be defined to include the additional states by changing the definition of $\mathbf{x_k}$. This implementation of lifting is also different from that presented in [6] to match the modification in the linearization of the Lyapunov function.

In a lifted system where a total of $k+1$ iterates of an algorithm is created and where x_0 is the first iterate and x_k is the last, $\mathbf{x_k}$ is defined as

$$\mathbf{x_k} = \begin{bmatrix} x_{k-1} - x_s \\ x_{k-1} - x_s \\ \vdots \\ x_0 - x_s \end{bmatrix} \tag{3.18}$$

Meanwhile, $\mathbf{x_{k+1}}$ is defined as

$$\mathbf{x_k} = \begin{bmatrix} x_k - x_s \\ x_{k-1} - x_s \\ \vdots \\ x_1 - x_s \end{bmatrix} \tag{3.19}$$

In (3.18), and (3.19), if an algorithm can be defined with a minimmum of n iterates, the lifting dimension is the total number of defined iterates minus n.

4

Code Components

The package derives a guaranteed worst-case convergence rate by following the set of instructions presented in Chapter 3, creating and solving an optimization problem to derive a performance certification.

Implementing a mathematical procedure into a program presents a list of challenges, such as enabling the program to understand and differentiate between variables representing concepts such as gradients or states. Another challenge is the formulation and solving of a convex optimization problem. Meanwhile, the goal of creating a simple user experience has to be fulfilled. This chapter goes into the code that constitutes the package and enables these functionalities.

4.1 Expressions

Expressions are data structures used to represent variables created in the analysis, such as the state of an algorithm, or the inner product of two state vectors. Expression enables the implementation of the elements presented in Chapter 3 into a computer program.

Expression types

The `AlgorithmAnalysis,jl` package defines the field `R` as a concrete expression type representing real scalar expressions. For vector expressions, they can exist in either one of two vector spaces $\mathtt{R}^n$ and $\mathtt{R}^m$ defined in the package. The analysis of GD, HB, and FG creates scalar expressions in `R` and vector expressions in the same vector space $\mathtt{R}^n$.

State vectors and the gradient of a function at a state vector are represented by vector expressions in the $\mathtt{R}^n$ vector space by the package, while the value of a function evaluated at

a vector are scalar expressions in R. Inner products of two vectors in the same vector space or the square norm of a vector, which appear in (3.3) are also scalar expressions in R.

A vector expression can be defined with the operator $R^n()$ and a scalar expression with R()

```
@algorithm begin
x0 = R^n()
y0 = R()
end
@show(x0)

Vector in R^n()
Label: x0
Associations: Dual => x0*

@show(y0)

Scalar in R
Label: y0
```

Figure 4.1: Example of a real and vector expressions

Variable and decomposition expressions

Expressions can either be a variable expression or a decomposition expression representing some combination of variable expressions. An expression's decomposition is stored in its value field.

Variable expression is defined to be in a field or vector space and represents either a vector or scalar. A variable expression's decomposition is itself. The expressions x0 and y0 created in Fig. 4.1 are examples of a variable expression.

Decomposition expression is a combination of variables or other decomposition expressions in the same field or vector space. Its decomposition is a dictionary containing how many of each variable expression form the decomposition expression.

Gram matrix

A Gram expression data structure is used to represent Gram matrices. It contains the vector of vector expressions whose inner products make up the element of the Gram matrix. Fig. 4.3 shows the expression representing matrix [|x0|² ⟨x0,x1⟩; ⟨x0,x1⟩ |x1|²].

```
@algorithm begin
        x0 = R^n()
        x1 = R^n()
@show x0 + 2*x1
end

Vector in R^n
  Decomposition: x0 + 2 x1
  Associations: Dual => LinearFunctional{R^n}
```

Figure 4.2: Example of a decomposition expression

```
@show Gram([x0, x1])
Gram([x0, x1]) = Gram matrix of vector R^n[x0, x1]

Gram matrix in Gram
        Value: R^n[x0, x1] ⊗ R^n[x0, x1]
```

Figure 4.3: Example of a Gram matrix expression

4.2 Algebra

Expressions in the package belong in an inner product space, which is a set of elements that can be vectors or scalar and which supports certain operations such as norm and inner product in addition to algebraic operations.

`AlgorithmAnalysis.jl` supports operations that characterize inner product spaces between expressions. The examples that demonstrate these operations use vector expressions `x0` and `x1` in Fig. 4.2.

Addition or subtraction between expressions

Vectors and numbers can be added together in an inner product space if they are both vector expressions or scalar expressions. In an addition operation, the result is a new expression whose decomposition is the merging of the decomposition dictionaries of the expressions being added. Subtraction works the same as addition, except the values in the decomposition dictionary of the term being subtracted have their signs flipped.

```
a = x0+x1-x0-x0
@show(a)

Vector in R^n
        Label: a
        Decomposition: x1 - x0
        Associations: Dual => a*
```

Figure 4.4: Example of addition and subtraction operation

Multiplication or division between an expression and a scalar

Vectors and scalars can be scaled by a real number (not a scalar expression) in the package. The package performs the multiplication or division of an expression by scaling its value when scaling a variable expression and scaling the value of its decomposition dictionary if it is a decomposition expression.

```
c = x0*-3 + x1*2
@show(c)

Vector in R^n
        Label: c
        Decomposition: 2 x1 - 3 x0
        Associations: Dual => c*
```

Figure 4.5: Example of multiplication operation

Transpose of a vector

When a vector expression is created, its transpose is also created in the `associations` field, allowing the program to keep track of whether an expression is the transpose of another. In the AlgorithmAnalysis.jl package, the transpose of a transpose expression returns the original expression.

Inner product operation between two vectors

In an inner product space, the inner product operation of two vectors is possible and results in a scalar. For example, the inner product of vectors x1 and x0, denoted $\langle x1, x0 \rangle$, is calculated as $x0^{\mathsf{T}} * x1$.

```
@show(x0')

Oracle
        Description: Linear functional on R^n
        Label: x0*
        Properties: Linear()

@show(x0'')

Vector in R^n
        Label: x0
        Associations: Dual => x0*
```

Figure 4.6: Example of transpose operation

```
inner = x0'*x1
@show(inner)

Scalar in R
        Label: <x0,x1>
        Oracles: x1*
```

Figure 4.7: Example of an inner product between two vector expressions

The ordering of vector inner products can result in two variables with different labels and perceived by the program as two different objects even though they represent the same mathematical inner product. For example, expressions labeled `<x1, x0>` and `<x0, x1>` are both created by taking the inner product of `x0` and `x1` but is not understood by the package as the same expression. This was shown during the testing of the package to affect the robustness and accuracy of the analysis through testing during the development of the package. A function is used to enforce a standard ordering based on hash: it determines the order of the inner product operation by comparing each expression's hash, ensuring the inner products between any two vector expressions are consistently ordered.

Squared norm

A vector expression can be squared to create its squared norm, a scalar expression.

```
@show (x0'*x1, x1'*x0) # Create two identical expressions

(<x0,x1>, <x0,x1>)

@show (x0'*x1 + x1'*x0)

Scalar in R
  Decomposition: 2 <x0,x1>
```

Figure 4.8: Example of consistent inner product labelling order

```
norm = x0^2

@show(norm)
norm = |x0|²
Scalar in R
        Label: |x0|²
        Oracles: x0*
```

Figure 4.9: Example of norm of vector expression

Outer product

The outer product of a vector of vector expressions in the same inner product space can be found using the $\otimes$ function. It is used to construct a Gram matrix from two vectors of vector expressions.

```
op = [x0, x1] ⊗ [x0, x1]

@show op
op = R[|x0|² <x0, x1>; <x0, x1> |x1|²]

2x2 Matrix{R}:
 |x0|²    <x0, x1>
 <x0, x1>  |x1|²
```

Figure 4.10: Example of outer product

4.3 Oracles

As mentioned in Section 3.2, algorithm analysis of first-order systems relies on interpolation conditions — for every state at which the gradient of the system is taken, constraints are applied to the state vector, the function value at said vector, and the gradient at that vector. To implement these interpolation conditions, the package uses oracles, data structures containing the relation and constraint information between expressions. Each oracle represents a class of function and can only exist if there exist interpolation conditions for said class.

Oracles can be sampled at one expression to return another, establishing the relation information between the two expressions.

When the oracle `f` representing a differentiable function is created, its corresponding gradient oracle `f'` is automatically created and given the property of being the gradient of `f`.

The oracle can be sampled at different expressions. For example, Fig. 4.11 shows how the oracle `f'` is sampled by defining `f'(x0)` and `f'(xs)` inside the labeling macro to create expressions ∇`f(x0)` and ∇`f(xs)`. This sampling can also be seen in Fig. 1.2.

```
@algorithm begin
        f = DifferentiableFunctional{Rⁿ}()
        f ∈ SmoothStronglyConvex(1, 10)
        x0 = Rⁿ()
        xs = first_order_stationary_point(f)
        f'(x0)
        f'(xs)
end
```

Figure 4.11: Example of sampling an oracle

An oracle stores information about every expression at which it has been sampled along with their corresponding output. Fig. 4.12 shows the the input and output information of the oracle `f'` defined in Fig. 4.11.

```
inputs_outputs(f')

(Rⁿ[xs, x0], Rⁿ[0, ∇f(x0)])
```

Figure 4.12: Example of the inputs outputs information of an oracle

In Fig. 4.11, by defining the DifferentiableFunctional oracle `f` as belonging to the 10 smooth

1 strongly convex class of functions, the convex property is assigned to `f`. This information is also stored in the oracle.

```
f

Oracle
        Description: Differentiable functional on R^n
        Label: f
        Properties: 10-smooth, 1-strongly convex
        Associations: Gradient => ∇f

f'

Oracle
        Description: Map from R^n to R^n
        Label: ∇f
        Properties: Empty set of properties
        Associations: GradientOf => f
```

Figure 4.13: Example of the properties of an oracle

During the analysis process, the package can derive the necessary information to create interpolation conditions from the oracles: the property of the oracles and the interpolated points, or at which expression the oracle has been sampled. Fig. 4.14 shows constraints created by sampling an oracle representing the 1 smooth 10 strongly convex function class at the interpolated points `x0` and `xs`. This matches the interpolation condition equation in Eq. (3.3).

```
constraints(f')
Set of constraints with 2 elements:
  0 ≤ 0.9 f(x0) + 0.1 <x0,∇f(x0)> - 0.5 |xs|^2 - 0.1
     ↪ <xs,∇f(x0)> - 0.9 f(xs) - 0.05 |∇f(x0)|^2 + <xs,x0> -
     ↪ 0.5 |x0|^2
  0 ≤ -0.9 f(x0) - 0.5 |xs|^2 + <x0,∇f(x0)> - <xs,∇f(x0)> +
     ↪ 0.9 f(xs) - 0.05 |∇f(x0)|^2 + <xs,x0> - 0.5 |x0|^2
```

Figure 4.14: Example of constraints created by sampling an oracle

While this section focuses primarily on `DifferentiableFunctional` oracles and their associated gradients, which are most relevant to analyzing smooth strongly convex functions, the package also supports other types of oracles. These include general `Map`, `LinearMap`, and `SymmetricLinearMap` types, which are especially useful in the analysis of primal-dual

algorithms and other composite optimization frameworks. Each oracle can also be associated with a wide variety of mathematical properties. Oracles representing function classes are not just limited to representing smooth strongly convex function classes, and can also support properties such as being sector-bounded or slope-restricted, as commonly encountered in robust control or nonlinear system analysis. The framework is extensible: any function property can be encoded into an oracle as long as its interpolation conditions are formally defined and provable.

Transpose oracle

The transpose of an expression is coded in Julia to be an oracle and is primarily used to support the inner product and square norm operations. In this case, an oracle representing the transpose of a vector is sampled at another expression to create a scalar inner product expression. The oracle representing the transposed expression has the `Linear` property and is used to create constraints on the Gram matrix detailed in Section 3.2.

```
x0 = Rⁿ()

@show x0'

Oracle
  Description: Linear functional on Rⁿ
  Label: x0*
  Properties: Linear()
```

Figure 4.15: Example of a transpose oracle

4.4 States

The `AlgorithmAnalysis.jl` package represents the states of an algorithm using vector expressions. The user inputs the algorithm by defining the initial states, and how the algorithm updates is defined by creating an updated state as an algebraic combination of the initial state and the gradient. The algorithm can only be defined if the relationship between a state and its next state is established. This is done in the `AlgorithmAnalysis.jl` package using the `=>` operation inside the `@algorithm` labeling macro, and the next state is stored in the `next` field of a state expression.

When using the package, the user only has to define the next state relationship for the state expressions of the algorithm. For other expressions in the analysis, the package automatically

```
@algorithm begin
        x0 = R^n()
        x1 = x0 - α*f'(x0)
        x0 => x1
end

@show x0

Vector in R^n
  Label: x0
  Next: x1
  Associations: Dual => x0*
```

Figure 4.16: Example of the `=>` operation

assigns the relationship if it exists. For example, if a decomposition expression `y1` is defined as the sum of expressions `x0` and `x1`, and the next state of `x0` is `x1` and the next state of `x1` is `x2` while `y1`'s next state is not defined by the user, the package automatically define that next state it as an expression with decomposition `x1+x2`. This is shown in Fig. 4.17.

```
@algorithm begin
        x0 = R^n()
        x1 = R^n()
        x2 = R^n()
        y0 = x0 + x1
        x0 => x1
        x1 => x2
end

next(y0)

Vector in R^n
  Decomposition: x1 + x2
  Associations: Dual => LinearFunctional{R^n}
```

Figure 4.17: Example of automatic next state assignment for decomposition expressions

Automatic next state assignment is also performed on the output of sampled oracles. For example, if an oracle `f` is sampled at `x0` and its next state `x1`, the output expression `f(x1)` is automatically assigned as the next state of `f(x0)`.

```
@algorithm begin
        x0 = R^n()
        x1 = R^n()
        f'(x0)
        f'(x1)
        x0 => x1
end

next(f'(x0))

Label: ∇f(x1)
  Oracles: ∇f
  Associations: Dual => ∇f(x1)
```

Figure 4.18: Example of automatic next state assignment for oracle output expressions

First order stationary point

To represent the minimizer of a function, the user can use the function `first_order_stationary_point()` to create the minimizer vector expression. This function is a convenient way to create a vector whose next state is itself, and where the gradient is zero.

```
@algorithm begin
f = DifferentiableFunctional{R^n}()
xs = first_order_stationary_point(f)

@show next(xs), f'(xs)
(xs, 0)
```

Figure 4.19: Example of a minimizer expression

4.5 Automatic lifting

When using the `AlgorithmAnalysis.jl` package, the user can call the `lift` function to automatically create additional state iterates which will be a part of the analysis. Once the algorithm being analyzed has been defined, the latest iterate and the desired number of additional states or `lifting_dimension` can be provided as arguments to the `lift` function, which create the additional iterates in three steps:

1. State and input extraction: Given the latest iterate, its decomposition is split between inputs — expressions which are the output of sampling an oracle, and states — any

remaining expression in the decomposition. The states present in the decomposition of the latest iterates are sorted by their next value similar to a singly-linked list to create a list called `initial_state`: The first element of the list is the head, which is the first iterate that no other state expression points to as the next state. Each successive state in `initial_state` is the next state of the last element in the list, excluding the latest state.

2. Formula extraction: A state formula is created, consisting of scalars to multiply elements in `initial_state` and the inputs in order to create the latest iterate. Similarly, an input formula is created to express how each input is created using the `initial_state`.

3. Creating new state iterate: To create the next iterate, `initial_state`'s first element is removed and the lastest state appended. The input formulas can be applied to this updated state list to create the latest inputs. The updated state list and latest inputs are then used to create the newest state iterate, following the state formula.

This process is placed in a loop to ensure the number of additional states created matches the `lifting_dimension` value given by the user. Every new state created by the `lift` function is given the appropriate next state connection. Every new states and inputs are given labels `"(i)th_lift_state"` and `"(i)th_lift_(j)th_input"` respectively, where `i` is the total number of existing state plus one and `j` minus one is the number of input that has already been created to form the latest state.

4.6 Special syntax operations

The `AlgorithmAnalysis.jl` package contains functions that are named using math notation symbols and overloads Julia base functions to create a simplified user experience. These functions include:

- **Algebraic**: As shown in Section 4.2, Julia base functions `+`, `-`, `*`, `/`, and `^` are overloaded to support algebraic operations. Therefore, the package executes the function `x0+y0-x0-x0` in Fig. 4.4 using its addition and subtraction functions meant for expression variables instead of Julia's base functions.

- **Transpose and Gradient**: The AlgorithmAnalysis.jl package follows math notation in that both the transpose of a function and the gradient of a function are denoted using the operation `'` in the package. And since in Julia, the operator `'` uses the `adjoint` function, we overload the function in the package with functions that allow the user

to refer to the transpose of a vector expression when the operator `'` is placed after a vector expression. On the other hand, when used on one of the package's supported differentiable function oracles, the operator denotes the gradient of a function.

- **Sampling functions and gradients**: By sampling an oracle at input an expression, we get an output expression that together with the input expression is constrained according to the interpolation condition. The package overloads the `()` operation to sample an oracle. For example, we sample a `DifferentiableFunctional` oracle `f` at expression `x0` simply by calling `f(x0)`.
- **Sampling transpose oracles**: As the transpose of a vector expression is considered an oracle, inner product or norm expressions are created by sampling an oracle. Therefore, the package overloads the `*` operation when used between a transpose oracle and a vector expression to denote sampling that oracle, facilitating the inner product operation.
- **Oracle property**: The function class is a necessary requirement to identify the property of an oracle and therefore its associated interpolation conditions. Therefore, the package uses the operation $\in$ to denote an oracle belonging to one of the package's supported classes. In Fig. 4.20, the function `f` is defined to be in the class of m smooth L strongly convex functions which informs the analysis process on which set of interpolation conditions to apply.

```
f = DifferentiableFunctional{R^n}()
f ∈ SmoothStronglyConvex(m, L)
```

Figure 4.20: Example of the $\in$ operation

4.7 Labeling expressions

The package uses a macro to keep the process of providing inputs to the program simple as some of the rules of programming might be difficult for novice users to navigate. While the package still works without the label functionalities, it is made more accessible both in terms of entering inputs and interpreting results produced by the package.

@algorithm label macro

When using the AlgorithmAnalysis.jl package, the user defines the algorithm, function class, and the performance measure inside of the `@algorithm` macro. It is the main way through

which the process of providing input for the analysis is simplified.

When an expression is defined inside the labeling macro `@algorithm`, the expression object created is given the label based on the variable name used.

```
@algorithm x0 = R^n()

@show x0

Vector in R^n
  Label: x0
  Associations: Dual => x0*
```

Figure 4.21: Example of a labeled expression

However, an expression defined outside of the labeling macro, as shown in Fig. 4.22, would be given a default label.

```
x3 = R^n()

@show x3

Vector in R^n
Label: Variable{R^n}
Associations: Dual => LinearFunctional{R^n}
```

Figure 4.22: Example of an unlabeled expression

The `@algorithm` macro is also responsible for simplifying the process of assigning an expression as the 'next' of another. While in regular Julia syntax, the code `x0 => x1` simply creates a pair between the 2 variables, when executed inside the macro automatically assigns the expression `x1` as the next state of `x0`. This could be seen in Fig. 4.16.

Default labels

In special cases, expressions are automatically given default labels that follow common math notation. This list includes:

- **Transpose**: A transposed variable is labeled by appending an asterisk. For example, given a vector expression `x`, its transpose `x'` is labeled x^*.
- **Gradient**: A gradient is labeled using the nabla symbol. For example, given a DifferentiableFunctional Oracle `f`, its gradient `f'` is labeled ∇ `f`

- **Abstract Operator Application**: An abstract operator applied to an expression is labeled with parenthesis similar to the mathematical notation of a function. For example, if a DifferentiableFunctional Oracle `f` is sampled at vector expression `x0`, the resulting expression is labeled `f(x0)`
- **Inner Product**: An inner product between two expressions is displayed using angled brackets. For example, the inner product of vector expressions `x0` and `x1` is labeled `<x0, x1>`
- **Squared norm**: An inner product between an expression and itself is displayed following squared norm math notation. For example, the square norm of vector expression `x0` is labeled $|x0|^2$
- **Oracle description**: Any created oracle is given a description based on its type, which is displayed when the oracle is accessed in the terminal.

```
f = DifferentiableFunctional{Rⁿ}()
f ∈ SmoothStronglyConvex(1, 10)
end

@show f

Oracle
        Description: Differentiable functional on Rⁿ
        Label: f
        Properties: 10-smooth, 1-strongly convex
        Associations: Gradient => ∇f
```

Figure 4.23: Example of an oracle's description

4.8 Constraints

During the analysis process, constraints are created by interpolation conditions, as well as when the user defines constraints on the iterative algorithm being analyzed. In order to keep track of these constraints while keeping the process of defining them simple, the program uses data structures that include the scalar expression being constrained and the constraint itself. The package defines three constraints, which place the expression under it into one of three cones, which include:

Equal to zero Scalar expressions can be constrained to be equal to zero with the $== 0$ operation, in which case the expression is constrained to exist in the zero set cone.

Non-positivity or non-negativity Scalar expressions can be constrained to be larger or equal to zero or less or equal to zero with the ≥ 0 or ≤ 0 operation, in which case the expression is constrained to exist in the positive orthant cone.

Positive or negative semidefinite Symmetric matrices consisting of scalar expressions can be constrained to be positive semidefinite with the $\succeq 0$ or $\preceq 0$ operation, in which case the expression is constrained to exist in the positive semidefinite cone.

Constraints upon being defined are added to the `constraints` field of each variable expression that form the decomposition of the expression being constrained. Fig. 4.14 is an example showing three constraints automatically created by an oracle's interpolation conditions. In addition to being created by sampling oracles, constraints can also be defined by users. When analyzing any algorithm, constraints can be added to the initial condition of the algorithm. Any constraints added by the user are included in the formation of the optimization problem used to derive the performance bound.

```
@algorithm (x0-xs)^2 <= 1
0 ≤ 1 - |xs|² + 2 <x0,xs> - |x0|²
```

Figure 4.24: Example of user added constraint

4.9 Performance measure

Part of the required inputs to perform the analysis is the performance measure. The worst-case convergence rate found by the analysis belongs to this performance measure. In Fig. 1.2, the performance measure is set as $(x_0 - x_s)^2$, which is the norm or distance between the initial point and the goal. This means the convergence rate guarantee returned is that of the distance between the point updated using gradient descent after each iteration x_k and the goal x_s. Depending on which criteria the user wishes to analyze the algorithm by, the performance measure can be modified so long as it is a scalar expression.

4.10 JuMP modeling language

The search for a Lyapunov function that certifies a given convergence rate is an optimization problem, one where the parameters are the optimization variables P, λ, and μ. Therefore, in order to formulate and solve optimization problems, the `AlgorithmAnalysis.jl` package uses JuMP [15], a modeling language specialized in mathematical optimization embedded in

Julia as part of the analysis process. The tools and functionalities offered by JuMP enable and simplify the steps of creating and solving an optimization problem. These tools are:

Modeling An optimization problem created by JuMP would include variables and their constraints along with the problem to be optimized, all of which need to be passed on to the solver. JuMP works by creating a model in which every element of a problem is defined and categorized. Variables and their constraints can then be defined in the model to form the optimization problem.

Solver JuMP supports a large list of open-source and commercial solvers, which are packages containing algorithms to find solutions to the optimization problem being formulated. While examples of analysis shown in this thesis and the results in Chapter 6 use the Mosek [16] solver, any JuMP-supported solver capable of solving semidefinite problems can be used instead.

Solution After an optimization problem has been formed and solved, the solver returns whether the constraints on the Lyapunov function hold, indicating whether or not the convergence rate chosen can be guaranteed.

The package has been tested with the SCS [17] and Mosek solvers. Mosek was chosen over SCS as the SCS solver produced inaccurate performance guarantees in select scenario, which is discussed in Chapter 7. It should be noted however that while the SCS can be installed for free in JuMP and Julia, Mosek requires a liscense, although a free academic license is available.

Suppose we have a trivial optimization problem: minimize $x+y$, subject to $x \geq 3$ and $y \geq 4$. It can easily be found the minimum value of $x + y$ is 7. Fig. 4.25 shows how JuMP can optimize the problem.

Lyapunov-based algorithm analysis finds the performance guarantee by certifying whether there exist optimization variables with which the optimization problem together with its constraints are satisfied. To demonstrate this functionality, we can use the model used in Fig. 4.25 with the same constraints on x and y, but instead of setting an objective to minimize $x+y$, set a constraint that $x+y = 6$. As there exists no x and y given the constraints applied on them which would satisfy the constraint $x + y = 6$, the optimization problem cannot be solved. Given this problem, JuMP optimizes the problem and returns the termination code `INFEASABLE` to indicate that no solution can be found in Fig. 4.26.

```
model = JuMP.Model(SCS.Optimizer)
JuMP.set_silent(model)
JuMP.@variable(model, x)
JuMP.@variable(model, y)
JuMP.@constraint(model, x ≥ 3)
JuMP.@constraint(model, y ≥ 4)
JuMP.@objective(model, Min, x + y)
JuMP.optimize!(model)

JuMP.objective_value(model)
7.000038313789448
```

Figure 4.25: Example of JuMP minimizing an optimization problem

```
JuMP.@constraint(model, x + y == 6)
JuMP.optimize!(model)

JuMP.termination_status(model)
INFEASIBLE::TerminationStatusCode = 2
```

Figure 4.26: Example of JuMP certifying the feasibility an optimization problem

5

Analysis Process

As described in Chapter 3 The Lyapunov function approach certifies whether a given convergence rate can hold given the input by finding a Lyapunov function that both bounds the performance measure and converging at the given rate. The analysis process takes place in four steps:

1. The components necessary to certify a convergence rate — the algorithm, the constraints, and the performance measure — are collected automatically to form a systematic characterization of the analysis problem, using the code structures described in Chapter 4. This characterization includes using the state information to determine how the algorithm being analyzed updates, creating constraints on interpolated points based on the interpolation conditions of the oracles' properties.

2. These data structures are converted to real number vectors and matrices in the form of a linear function of the initial states and inputs. These variables represent the algorithm's initial and updated states as well as the constraints created.

3. An optimization problem is created using the representations created in the last step inside a JuMP model. The problem of finding a Lyapunov function is transformed into finding the optimal with which the problem is feasible, which proves whether a certain convergence rate is feasible for a given problem.

4. The above three steps are repeated with different convergence rates as the program searches for the smallest feasible convergence rate using bisection search.

This chapter details the analysis process of analyzing gradient descent's performance at optimizing any 10 smooth 1 strongly convex function as shown in Fig. 1.2, including how the steps presented in Chapter 3 are performed and how the optimization problem is formed and solved to derive worst-case performance convergence rate.

5.1 Analysis problem formulation

Once the user has given the package the necessary input to perform analysis, they can call the `rate` function on the performance measure to begin analysis. The package begins the analysis automatically finding every variable that is part of the analysis problem using a recursive function.

This process starts with the system creating a set of expressions present in the performance measure, a set of their associated oracles, and a list of constraints associated with found expressions and oracles. The original set of expressions is then appended with expressions present in the constraints or sampled with the oracles, and the process is repeated until no new variable is found. Figure 5.1 shows how every expression, constraint, and oracle is collected.

```
vars, cons, orcs = variables_constraints_oracles(performance)

@show vars
Set{Expression} with 11 elements:
  <∇f(x0),xs>
  f(x0)
  |∇f(x0)|²
  |xs|²
  <x0,xs>
  |x0|²
  f(xs)
  ∇f(x0)
  <∇f(x0),x0>
  xs
  x0
```

Figure 5.1: Collected expressions, oracles, and constraints (part one)

```
@show cons
Set of constraints with 3 elements:
  0 ⪯ Gram matrix of vector R^n[∇f(x0), xs, x0]
  0 ≤ -0.1 <∇f(x0),xs> - 0.9 f(xs) + 0.9 f(x0) - 0.05
     ↪ |∇f(x0)|² + <x0,xs> + 0.1 <∇f(x0),x0> - 0.5 |xs|² -
     ↪ 0.5 |x0|²
  0 ≤ -<∇f(x0),xs> + 0.9 f(xs) - 0.9 f(x0) - 0.05 |∇f(x0)|² -
     ↪ 0.5 |xs|² + <x0,xs> + <∇f(x0),x0> - 0.5 |x0|²

@show orcs
Set{Oracle} with 5 elements:
  x0*
  ∇f(x0)*
  ∇f
  f
  xs*
```

Figure 5.1: Collected expressions, oracles, and constraints (part two)

Pruning Gram matrix constraints

Due to the fact the package uses a recursive function to create the Gram matrix constraints that interpolate the inner product expressions present in the analysis problem, it in some cases creates more Gram matrix constraints than intended. These constraints are applied to Gram matrices that are the principal submatrix of the Gram matrix we wish to constrain. Since any principal submatrix of a matrix is positive semidefinite as long as the matrix is positive semidefinite, any constraint applied on the principal submatrix is redundant.

While redundant constraints do not affect the accuracy of the certification of a convergence rate, they can significantly increase computational cost. As will be discussed in the next section, each constraint applied to a Gram matrix introduces a number of additional optimization variables equal to the size of the matrix. Therefore, including any unnecessary Gram matrix constraints leads to an increased problem size and can slow down the analysis process. To prevent this therefore the package prunes the list of constraints to remove any redundant constraints on the Gram matrix.

5.2 Linear form transformation

As specified in Section 3.3, the linear matrix inequalities are constructed from the linear form of the Lyapunov functions and the constraints as a function of the vector $[x; u]$ that represents initial states and inputs. This process is done in three steps, which are:

1. Of every expression that has been created during the input process, define the initial state vector `x` as every real expression which contains another expression in its next field.

```
x  = collect(v for v ∈ vars if !ismissing(next(v)) && v isa R)
4-element Vector{R}:
    f(xs)
    |xs|²
    |x0|²
    <x0,xs>
```

Figure 5.2: Initial state real scalar expressions

2. The input vector u is then defined as every real expression that does not have a next state.

```
u  = collect(v for v ∈ vars if ismissing(next(v)) && v isa R)
4-element Vector{Expression}:
    <∇f(x0),xs>
    <∇f(x0),x0>
    |∇f(x0)|²
    f(x0)
```

Figure 5.3: Input real scalar expressions

3. The initial state and input vector `[x; u]` is the code equivalent of $\begin{bmatrix} x & u \end{bmatrix}^\mathsf{T}$ and can be multiplied with a matrix of real numbers to create every expression required to form the linear matrix inequality. The transformation of a decomposition expression of Gram matrix expression into such real number matrix will be referred to as the linear form of an expression and is crucial to creating the semidefinite problem in JuMP.

The package now transforms the necessary input into its linear form in preparation for the final step of formulating an optimization problem in JuMP.

```
linear_form = (linearform([x; u] => x0^2 - 3*(x0'*xs)))

@show linear_form

1x8 Matrix{Int64}:
 0   -3   1   0   0   0   0   0

@show linear_form*[x; u]

Scalar in R
  Decomposition: -3 <x0,xs> + |x0|²
```

Figure 5.4: Example of linear form of a scalar expression

5.3 Formulating optimization problem

Performance measure

The linear form matrix of the performance measure is the first of the three components needed to form the Lyapunov function in (3.8), we used $\|x_k - x_s\|^2$. For example, the performance measure in Fig. 1.2, which is defined as `(x0-xs)²` and which evaluates into `|x0|² - <xs, x0> - <x0, xs> + |xs|²`, has the linear form presented in Fig. 5.5.

```
𝒫 = vec(linearform( [x; u] => performance ))
print(𝒫)
[-1, -2, 1, 0, 0, 0, 0, 0, 0]
```

Figure 5.5: Linear form matrix of expression `(x0-xs)²`

Lyapunov function formulation

The Lyapunov functions can be formed according to equations (3.10). This is done by the package by first separating every real scalar expression, which are inner product, norms, and function value expressions, and excluding state vector expressions, into an initial state and an updated state by making the distinction between real expressions that have a next state and those that do not.

In order to perform linear form transformations, we have already defined `x` as the initial state in Section 5.2 as every real scalar expression with a next expression. We can then define the updated state $\mathtt{x}^+$ consisting of the next expression of every expression in the initial state.

```
x⁺ = next(x)

4-element Vector{R}:
    f(xs)
    |xs|²
    |x0|² - 0.36363636363636365 <∇f(x0),x0> +
       ↪ 0.03305785123966942 |∇f(x0)|²
    -0.18181818181818182 <∇f(x0),xs> + <xs,x0>
```

Figure 5.6: Updated state $\mathtt{x}^+$ real scalar expression

The initial state expressions x defined in Fig. 5.2 and the updated state real expressions $\mathtt{x}^+$ defined in Fig. 5.6, are then transformed into their linear form matrices. This is the second of the three components needed to formulate the Lyapunov function and is shown in Fig. 5.7 and Fig. 5.8.

```
X  = linearform([x; u] => x)
    4x9 Matrix{Int64}:
    1  0  0  0  0  0  0  0  0
    0  1  0  0  0  0  0  0  0
    0  0  1  0  0  0  0  0  0
    0  0  0  1  0  0  0  0  0
```

Figure 5.7: Linear form state matrix x

```
X⁺ = linearform([x; u] => x⁺)
4x8 Matrix{Real}:
1  0  0  0   0          0          0          0          0
0  1  0  0   0          0          0          0          0
0  0  1  0   -0.363636  0.0330579  0          0          0
0  0  0  1   0          0          0          0         -0.1818
```

Figure 5.8: Linear form state matrix $\mathtt{x}^+$

Following the steps presented in (3.8) and (3.10) of Chapter 3, the Lyapunov function can be formed by first defining an optimization variable P in the JuMP model as a JuMP variable. Once JuMP and the solver start optimizing the problem, P is one of the variables that will be optimized to certify a convergence rate. The code that creates the JuMP model and the formulation of the Lyapunov functions is presented in Figure 5.9.

```
# Create the JuMP model
model = JuMP.Model(SCS.Optimizer)
JuMP.set_silent(model)
# Create Optimization variables
JuMP.@variable(model, P[1:length(x)])
# Create Lyapunov functions
V = X'*P
V⁺ = X⁺'*P
L1 = 𝒫 - V
L2 = V⁺ - ρ²*V
```

Figure 5.9: Formation of optimization problem in JuMP

5.4 Constraints

The last of the three components needed to perform analysis, the constraints are also transformed into their linear form and added to the optimization in three steps:

Optimization variable multipliers For each constraint i, two optimization variables λ and μ, which represent Λ_i^1 and Λ_i^2 in Chapter 3 are defined as JuMP variables. If the constraint is applied to the Gram matrix, the optimization variables will be two matrices of the same size. Otherwise, if the constraint is created from the interpolation conditions of the class of function and is applied to a single real scalar expression, the optimization variables will have a size of 1.

Constraint on multiplier The JuMP variables multipliers are constrained in the JuMP model depending on the constraint expression they were created for: The multiplier is not constrained if the expression is constrained to be zero, constrained to be non-negative if the expression is constrained to be non-negative, and constrained to be symmetrical and in the JuMP supported positive semidefinite cone if the expression is constrained to be positive semidefinite.

Transformation into the linear form of constraints For each constraint, the expression being constrained is scaled by the two optimization variable multipliers, before being transformed into their linear form. The two resulting matrices of multipliers are added to `L1` and `L2` respectively.

5.5 Derived feasibility and bisection search

Upon the completion of the linear matrix inequalities, the solver of the JuMP model is called to optimize the problem and find the variables P, λ-s and μ-s for which the linear matrix inequality is satisfied and a convergence rate ρ can be guaranteed.

In order to find the worst-case performance rate, the program performs a bisection search, also known as binary search, for the smallest value ρ between 0 and 1 that makes the optimization problem feasible, calling a function to perform the steps presented in this chapter for each value ρ and checking feasibility at each iterate of the search. The smallest value ρ found within a tolerance of 10^{-5} is returned as the guaranteed convergence rate, and the analysis process is complete.

6

Results

The `AlgorithmAnalysis.jl` package's ability to perform automatic algorithm analysis have been tested on GD, FG, HB. We have also tested on Triple-momentum (TMM), which was proven to be the fastest known globally convergent algorithm for minimizing smooth strongly convex functions in [18]. In [6], these four algorithms were tested with m-strong L-smooth convex function class over a range of m/L condition numbers between one and 100. To show the package's ability to accurately analyze algorithms for worst-case performance guarantees, we seek to replicate the analysis results presented in Fig. 2 of [6] with results derived from running `AlgorithmAnalysis.jl` over the same range of condition numbers. The analysis results are presented in Fig. 6.1.

To create Fig. 6.1, we set the value of m to 1 and sample 12 values of L that are logarithmically spaced between 1 and 100, using a base-10 scale. The convergence rate produced by the package is plotted on the y-axis, while the x-axis plots the condition number L/m of the tested (m, L) values. Also included in Fig. 6.1 is the theoretical performance limit of the TMM algorithm at optimizing smooth strongly convex function presented in [18]. Every algorithm analyzed were lifted using a lifting dimension of one.

Comparing Fig. 6.1 with Fig. 2 of [6] and Fig. 1 of [18], we find the analysis result produced by `AlgorithmAnalysis.jl` matches that in [6] for all four algorithms GD, FG, HB, and TMM. This proves the package's ability to accurately analyze these algorithms for smooth strongly convex functions.

Gradient descent

To analyze GD, we use stepsize $\alpha = 2/(L + m)$ similar to Fig. 1.2 and that used in [6]. The code used to generate GD(m, L) in Fig. 6.1 is presented in Fig. 6.2 with `m` set to one, `L` set to the sampled L values, and `lifting_dimension` set to one.

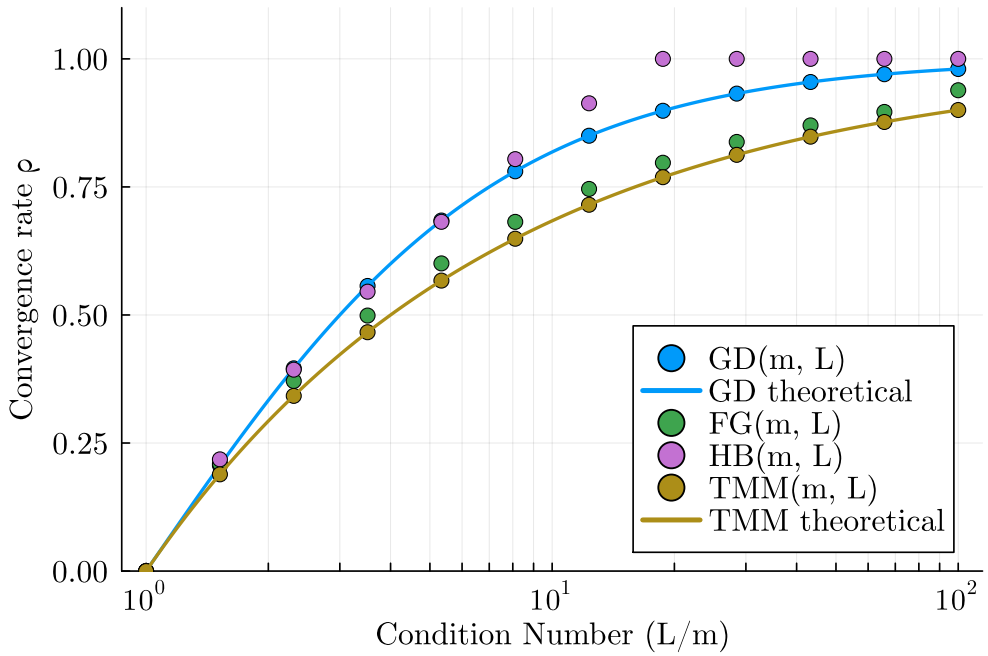


Figure 6.1: Convergence rate guarantee of four algorithms over smooth strongly convex function class

```
α = 2/(L+m)
@algorithm begin
    f = DifferentiableFunctional{R^n}()
    f ∈ SmoothStronglyConvex(m, L)
    xs = first_order_stationary_point(f)
    x0 = R^n()
    x1 = x0 - α*f'(x0)
    x0 => x1
    lift(x1, lifting_dimension)
    performance = (x0-xs)^2
end
@show rate(performance, prev_rate)
```

Figure 6.2: Analysis of gradient descent and m-L sector bounded functions

We have also tested the same gradient descent algorithm over a function class where the gradient is (m, L) sector-bounded. The code used to find the worst-case convergence rate guarantees of the fast gradient algorithm at optimizing (m, L) sector-bounded function classes is identical to the code used for GD on smooth strongly convex function, but with command

```
f' ∈ SectorBounded(m, L, xs, f'(xs))
```

replacing the command `f ∈ SmoothStronglyConvex(m, L)`. The plot of the result produced

is presented in Figure 6.3.

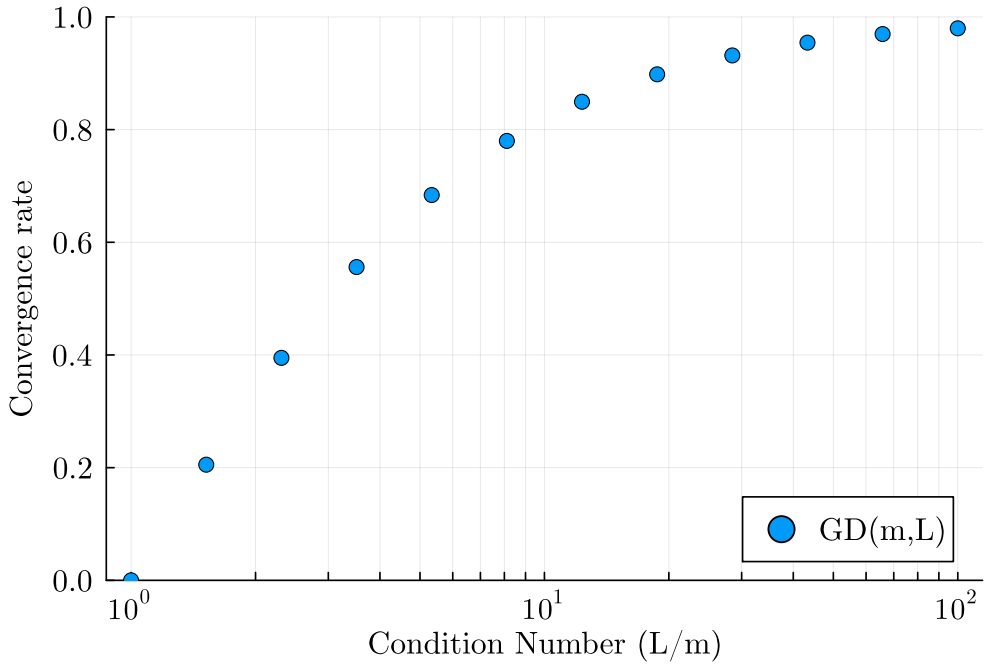


Figure 6.3: Convergence rate guarantee of gradient descent over sector bounded function class

In order to measure the effect of creating additional state iterates on the convergence rate guarantee found, we plot the results of analyzing GD with the same parameters as those used to create Fig. 6.1 but with `lifting_dimension` values zero and one. The analysis result is presented in Fig. 6.4, from which we can see that additional lifting does not affect the performance guarantee found for GD.

Fast gradient

We tested the fast gradient algorithm with step sizes similar to that used in [6]:

$$\alpha = \frac{4}{3L+m}, \quad \beta = \frac{\sqrt{3L+1}-2}{\sqrt{3L+1}+2}$$

The code used to find the worst-case convergence rate guarantees of the FG algorithm in Fig. 6.1 is presented in Fig. 6.5, with `m` set to one, `L` set to the sampled L values, and `lifting_dimension` set to one.

In order to see the effect of lifting on the convergence rate guarantee found, we plot the results of analyzing FG with `lifting_dimension` values of zero, one, and two. The analysis

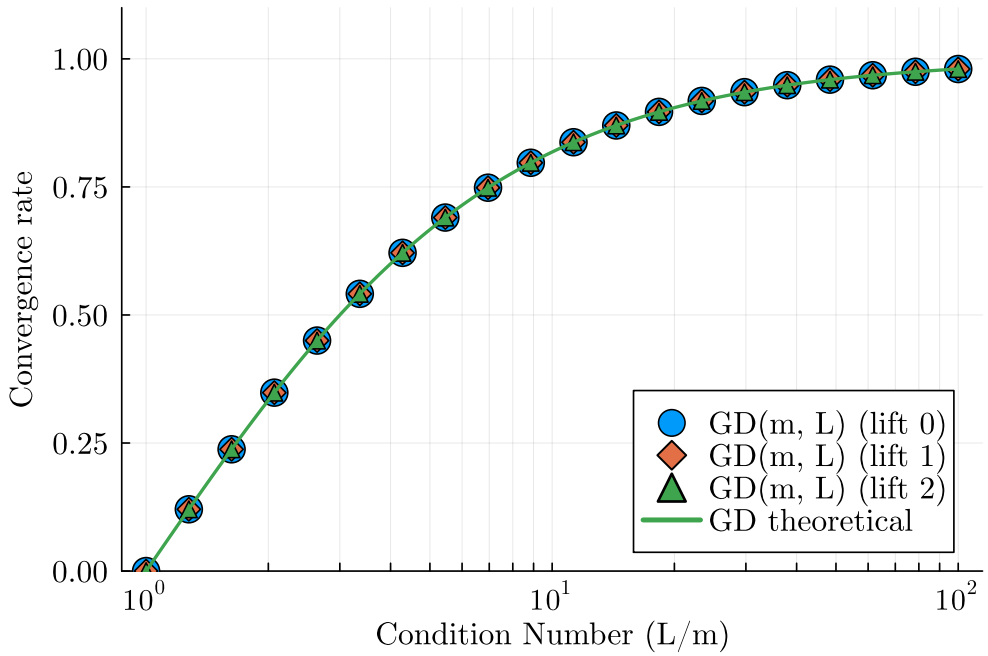


Figure 6.4: Convergence rate guarantee of gradient descent lifting dimensions zero and one, over smooth strongly convex function class

```
α = 4/(3*L+m); β=(sqrt(3*L+1)-2)/(sqrt(3*L+1)+2)
@algorithm begin
    f = DifferentiableFunctional{Rⁿ}()
    xs = first_order_stationary_point(f)
    f ∈ SmoothStronglyConvex(m, L)
    x0 = Rⁿ()
    x1 = Rⁿ()
    y1 = x1 + β*(x1 - x0)
    x2 = y1 - α*f'(y1)
    x0 => x1
    x1 => x2
    lift(x2, lifting_dimension)
    performance = (y1-xs)^2
end
@show rate(performance)
```

Figure 6.5: Analysis of fast gradient and L-smooth m-strongly convex functions

result is presented in Fig. 6.6. We can see that while there is no difference in the analysis result between a creating one or two more iterates than the minimum, a lifting dimension of zero gives the incorrect result compared to Fig.2 of [6].

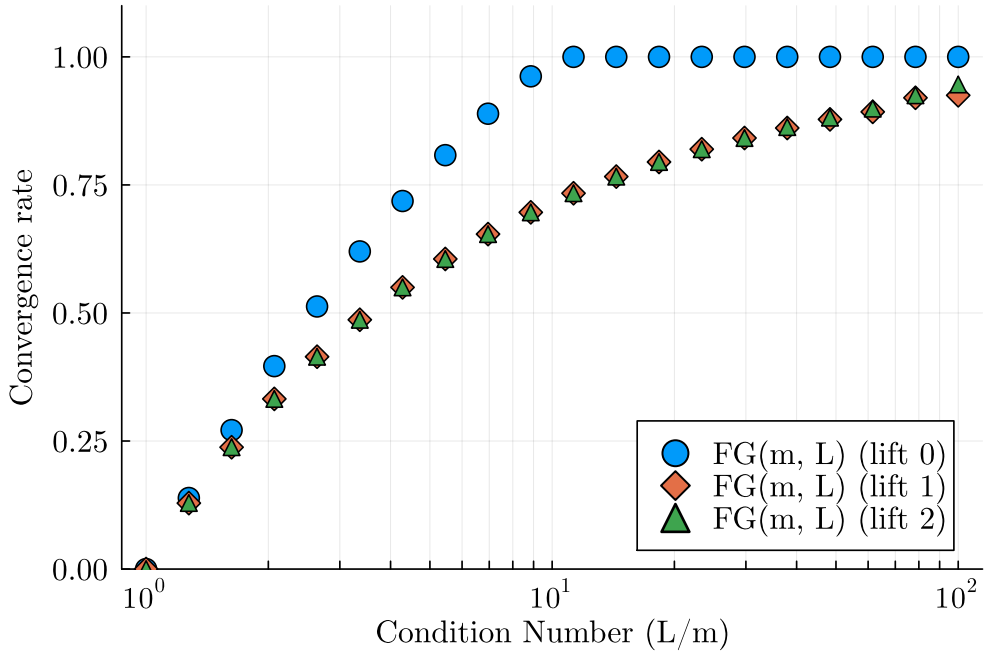


Figure 6.6: Convergence rate guarantee of fast gradient with lifting dimensions zero, one, and two, over smooth strongly convex function class

Heavy ball

We tested the heavy ball algorithm with step sizes similar to that used in [6]:

$$\alpha = \frac{4}{\left(\sqrt{L} + \sqrt{m}\right)^2}, \quad \beta = \left(\frac{\sqrt{L/m} - 1}{\sqrt{L/m} + 1}\right)^2$$

The code used to find the worst-case convergence rate guarantees of the heavy ball algorithm for optimizing m-strongly convex and L-smooth function classes are presented in Fig. 6.7, with `m` set to one, `L` set to the sampled L values, and `lifting_dimension` set to one.

We also plot the results of analyzing HB with `lifting_dimension` values zero, one, and two. The analysis result is presented in Fig. 6.8, which shows similar characteristic with Fig. 6.6 where a lift of zero produces inaccurate results while lifting values of one and two result in accurate and similar convergence rate guarantees.

Triple momentum

We tested the triple momentum algorithm, which was developed by Van Scoy, Freeman, and Lynch in [18] to be the fastest known globally convergent first-order algorithm at optimizing

```
α = 4/((sqrt(L)+sqrt(m))^2); β=((sqrt(L/m)-1)/(sqrt(L/m)+1))^2
@algorithm begin
    f = DifferentiableFunctional{R^n}()
    xs = first_order_stationary_point(f)
    f ∈ SmoothStronglyConvex(m, L)
    x0 = R^n()
    x1 = R^n()
    x2 = x1 - α*f'(x1) + β*(x1-x0)
    x0 => x1
    x1 => x2
    lift(x2, lifting_dimension)
    performance = (x1-xs)^2
end
@show rate(performance)
```

Figure 6.7: Analysis of heavy ball and L-smooth m-strongly convex functions

strongly convex functions. The algorithm is defined as:

$$x_{k+1} = (1+\beta)x_k - \beta x_{k-1} - \alpha\nabla f((1+\gamma)x_k - \gamma x_{k-1}) \tag{6.1}$$

The triple momentum algorithm's optimal parameters are determined by the condition number $k = L/m$ when optimizing L-smooth m-strongly convex functions. The algorithm's parameters are defined as:

$$\begin{aligned}
\rho &= 1 - \frac{1}{\sqrt{k}}, \\
\alpha &= \frac{1+\rho}{L}, \\
\beta &= \frac{\rho^2}{2-\rho}, \\
\gamma &= \frac{\rho^2}{(1+\rho)(2-\rho)},
\end{aligned}$$

Under these parameters, it was proven in [18] that the performance guarantee that can be guaranteed the function $1 - \sqrt{m/L}$, which is plotted alongside the rates produced by the package in Fig. 6.1 as TMM theoretical. The code used to find the worst-case convergence rate guarantees of the triple momentum algorithm is presented in Fig. 6.9, with `m` set to one, `L` set to the sampled L values, and `lifting_dimension` set to one. Note that in Fig. 6.9, the performance measure is set to `((1+delta)*x2 - delta*x1 - xs)^2` in order to get the

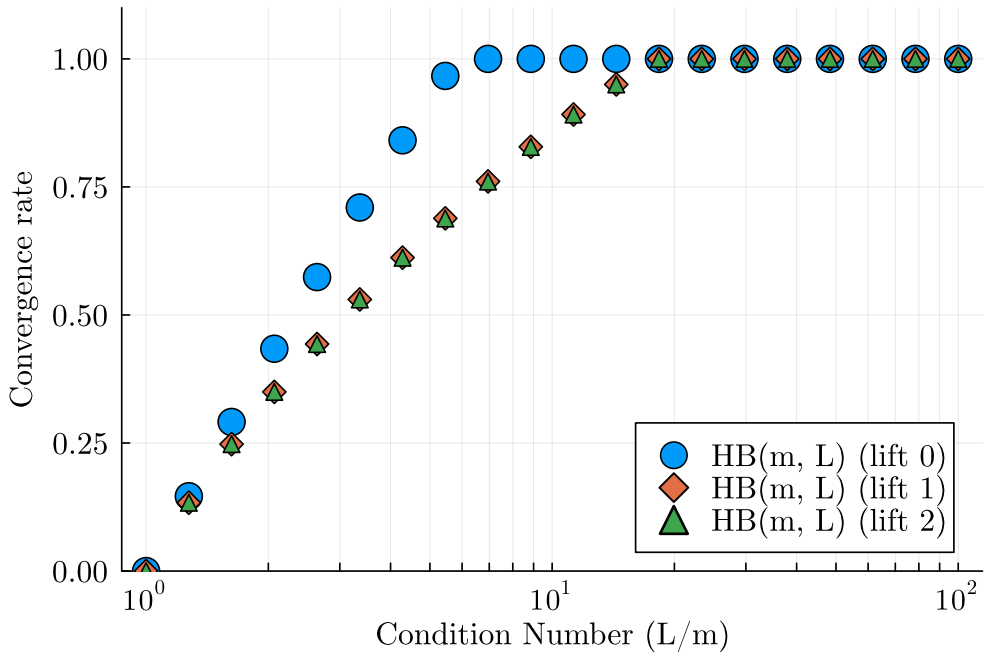


Figure 6.8: Convergence rate guarantee of heavy ball with lifting dimensions zero, one, and two, over smooth strongly convex function class

result in Fig. 6.1. This value is taken from Theorem 1 of [18] and adapted to fit the automated analysis in `AlgorithmAnalysis.jl`, and selecting a different performance measure might not result in analysis figures that match that in Fig.1 of [18].

When we plot the result of analyzing TMM with `lifting_dimension` values zero, one, and two, we see the same impact lifting has on the convergence rate guarantee as did FG and HB: not lifting the algorithm produces inaccurate results while creating one or two additional iterates does not affect the result. This can be seen in Fig. 6.10.

```
k = L/m; rho = 1 - 1/(sqrt(k))
α = (1 + rho)/L
β = (rho^2)/(2-rho)
gamma = (rho^2)/((1+rho)*(2-rho))
delta = (rho^2)/(1-rho^2)
@algorithm begin
    f = DifferentiableFunctional{R^n}()
    xs = first_order_stationary_point(f)
    f ∈ SmoothStronglyConvex(m, L)
    x0 = R^n()
    x1 = R^n()
    y1 = (1+gamma)*x1 - gamma*x0
    x2 = (1+β)*x1 - β*x0 - α*f'(y1)
    x0 => x1
    x1 => x2
    lift(x2, lifting_dimension)
    performance = ((1+delta)*x2 - delta*x1 -xs)^2
end
@show rate(performance)
```

Figure 6.9: Analysis of triple-momentum and L-smooth m-strongly convex function class

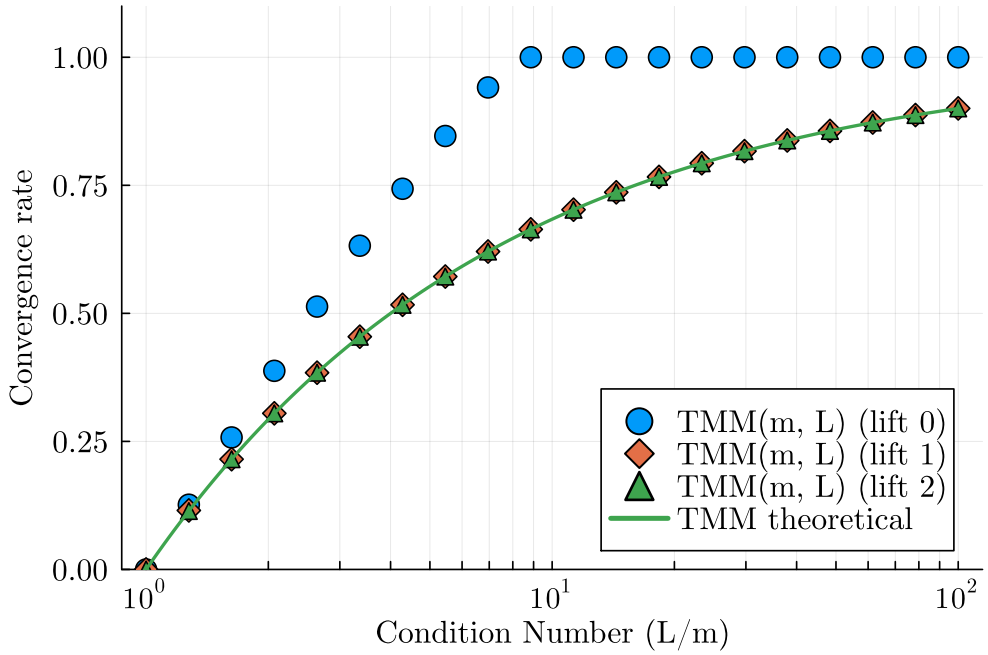


Figure 6.10: Convergence rate guarantee of triple momentum with lifting dimensions zero, one, and two,over smooth strongly convex function classes

7

Discussion And Conclusion

Discussion and Future work

In a development version of the `AlgorithmAnalysis.jl` package which uses the SCS [17] solver instead of the Mosek solver, we observed that the program produced inaccurate result in the analysis of TMM over L-smooth m-strongly convex functions for condition numbers L/M between one and two. In these cases, the program returned a performance guarantee faster than the algorithm's theoretical limit, which is set at $1-\sqrt{m/L}$. This behavior ceased when we switch to the Mosek solver.

While the current experimental validations demonstrate that the `AlgorithmAnalysis.jl` package accurately generates worst-case performance guarantees for several prominent first-order optimization methods, we can explore other methods to test compatibility. Future work should can include systematic testing and verifying the package against a wider range of first-order unconstrained optimization algorithms. This extended analysis will further confirm the robustness of the package's implementation of the Lyapunov approach and highlight any potential limitations.

An area of the package that can be expanded in the future is the analysis of the primal-dual algorithm used to optimize constrained problems. These problems are different from the unconstrained type we have looked at so far in that the minimizer x_s has to satisfy a constraint $Ax = b$ where A is an abstract matrix. In the Lyapunov-based approach in [12], this constraint would be characterized by bounds on the value of A, which introduces the values of the bounds on A as the third input in our analysis, on top of the algorithm and the function class. Similar to a function class, the matrix A would be represented in the analysis by interpolation conditions.

The SVD-based transformation used by the approach in [12] also separates each state vector

into two vectors in different vector spaces. These two vectors would each have its own corresponding gradient while together samples the function to create one function value. In order to automate the creation of these elements and their constraints, it requires oracles different from the one input one output ones we have used for the analysis of GD, FG, HB, and TMM — The function oracle for PD would have two vector expressions as input and one scalar expression as output, while the corresponding gradient oracle would have the same input and two vector expression outputs. The necessary interpolation conditions and oracle types to analyze PD have been implemented, however the package is unable to derive an performance guarantee accurate to those presented in [12]. Possible future work can include debugging the code that implements analysis of PD.

Concluson

This thesis has presented the implementation of algorithm analysis using Lyapunov functions in the Julia programming language. `AlgorithmAnalysis.jl` provides a robust and accessible platform for analyzing first-order optimization algorithms to derive a worst-case performance guarantee. The package adapts the Lyapunov approach to better suit software implementation, establishes a domain-specific language that substantially simplifies the user experience, and created a systemic and automated way to apply interpolation conditions and Lyapunov function-based optimization problems. Ultimately, the package bridges the gap between theoretical performance analysis and practical optimization tasks, making rigorous algorithm analysis accessible to a broader audience of researchers and developers.

The package can be found in the AlgorithmAnalysis.jl GitHub repository at https://github.com/vanscoy/AlgorithmAnalysis.jl.

References

[1] Stephen J. Wright and Benjamin Recht. *Optimization for Data Analysis*. Cambridge University Press, Cambridge, 2022.

[2] Boris T. Polyak. *Introduction to Optimization*. Optimization Software, Inc., New York, 1987. Chapter 3, Section 3.2: The Heavy Ball Method.

[3] Yurii Nesterov. *Smooth Convex Optimization*, pages 59–137. Springer International Publishing, Cham, 2018.

[4] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[5] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. 2012.

[6] Bryan Van Scoy and Laurent Lessard. A tutorial on a Lyapunov-based approach to the analysis of iterative optimization algorithms, 2023.

[7] Sebastian Bock and Martin Weiß. *Non-convergence and Limit Cycles in the Adam Optimizer*, page 232–243. Springer International Publishing, 2019.

[8] Yoel Drori and Marc Teboulle. Performance of first-order methods for smooth convex minimization: A novel approach. *Mathematical Programming*, 145, 06 2012.

[9] Baptiste Goujaud, Céline Moucer, François Glineur, Julien Hendrickx, Adrien Taylor, and Aymeric Dieuleveut. PEPit: computer-assisted worst-case analyses of first-order optimization methods in python, 2024.

[10] A. Megretski and A. Rantzer. System analysis via integral quadratic constraints. *IEEE Transactions on Automatic Control*, 42(6):819–830, 1997.

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

[12] Bryan Van Scoy, John W. Simpson-Porco, and Laurent Lessard. Automated Lyapunov analysis of primal-dual optimization algorithms: An interpolation approach. In *IEEE Conference on Decision and Control*, 2023.

[13] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. 3(1):1–122, January 2011.

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

[15] Miles Lubin, Oscar Dowson, Joaquim Dias Garcia, Joey Huchette, Benoît Legat, and Juan Pablo Vielma. Jump 1.0: recent improvements to a modeling language for mathematical optimization. *Mathematical Programming Computation*, 15(3):581–589, 2023.

[16] MOSEK ApS. *MOSEK Optimizer API for Julia 11.0.17*, 2025.

[17] Brendan O'Donoghue, Eric Chu, Neal Parikh, and Stephen Boyd. Conic optimization via operator splitting and homogeneous self-dual embedding. *Journal of Optimization Theory and Applications*, 169(3):1042–1068, June 2016.

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