ALGORITHM ANALYSIS: JULIA™PACKAGE IMPLEMENTATION OF OPTIMIZATION ALGORITHM ANALYSIS USING LYAPUNOV FUNCTION

A Thesis

Submitted to the

Faculty of Miami University

in partial fulfillment of

the requirements for the degree of

Master of Science

by

Lam Ngoc Ha

Miami University

Oxford, Ohio

2024

Advisor: Dr. Bryan Van Scoy

Reader: Dr. Veena Chidurala

Reader: Dr. Peter Jamieson

This thesis titled

ALGORITHM ANALYSIS: JULIA™PACKAGE IMPLEMENTATION OF OPTIMIZATION ALGORITHM ANALYSIS USING LYAPUNOV FUNCTION

by

Lam Ngoc Ha

has been approved for publication by

The College of Engineering and Computing

and

The Department of Electrical and Computer Engineering

Dr. Bryan Van Scoy

Dr. Veena Chidurala

Dr. Peter Jamieson

ABSTRACT

ALGORITHM ANALYSIS: JULIA™PACKAGE IMPLEMENTATION OF OPTIMIZATION ALGORITHM ANALYSIS USING LYAPUNOV FUNCTION

by Lam Ngoc Ha

Many problems that appear in science and engineering can be considered optimization problems, and first-order iterative algorithms are used to solve large scale problems in fields such as machine learning or data science. This makes the case for the ability to compare the performance of any set of algorithms in any application to find the best performing one. Currently however, analyzing algorithms is difficult for anyone without existing knowledge of the subject, regardless of the chosen method. In this thesis paper, we demonstrate the Algorithm Analysis program, a Julia package that provides a simple and more accessible way to analyze any optimization algorithm by leveraging the Lyapunov-based analysis approach. Simultaneously, we show that by creating a language specific to the domain of optimization algorithms, the Algorithm Analysis package can serve as a platform on which different methods to perform algorithm analysis can be implemented.

Contents

List of Figures

Dedication

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

Acknowledgements

I would like to acknowledge...

1

Introduction to Algorithm Analysis

Optimization problems can be in the broadest sense described as problems where an optimal solution is obtained using a limited amount of resources. Many problems that exist in the field of engineering and natural science can be categorized as optimization problems. For example, when mapping applications are used to navigate between two points, an algorithm finds the shortest path to a destination — minimizing the distance travelled — by choosing the direction of travel while under constraints such as traffic laws or avoiding road work.

Gradient-based iterative algorithms are a prominant tool to solve large optimization problems. Their ability to efficiently optimize functions without requiring an explicit formula makes them extensively used in fields such as machine learning and data science [1]. As there exists a theoretically infinite number of these algorithms and many commonly encountered optimization problems they can be applied to with varying speed and accuracy, a strong case can be made for the ability to gauge the performance of algorithms. This ability allows the comparing algorithms to find or derive the best performing one, improving efficiency, as well as saving time and computational resources solving the problems they are meant to. As a result, substantial research has been conducted to quantify the performance of algorithms either through emperical evidence or mathematical proof, the latter of which is the method the `AlgorithmAnalysis.jl` package uses.

Algorithm analysis is a field which seeks to mathematically prove the worst-case performance of an algorithm at solving a set of optimization problems. But as different methods of analyzing algorithms are developed, anyone seeking to use one of them have had to be able to understand the underlying math and apply it to their algorithm and problem set until recently [2]. The main work of this thesis is the development of a tool that analyze gradient-based algorithms' performance characteristics accessible to both experts and non-experts: `AlgorithmAnalysis.jl` is a computer program written in the Julia programming

language that automatically and systemically finds the worst-case performance guarantee of an algorithm at solving a specified set of problems. 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 solve 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 how close a solution is to being optimal. Well-known examples of this problem are present in the training large language models (LLMs) such as ChatGPT or the machine learning models that enable self-driving features in automotives. An inegral 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 some 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 of the optimal solution) using the gradient of the function at that 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 parameter of the algorithm. 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, and stays there after it reaches the goal where the gradient is zero $\nabla f(x_s) = 0$. Two areas where an algorithm like gradient descent can improve upon are the speed at which the goal is reached, or overshooting — where

the goal is not reached within an acceptable margin due to the fixed step size. Accelerated algorithms exist that seek to solve these problems, such as Polyak's Heavy Ball (HB) method which introduces a momentum that incorporates previous iterations of x:

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

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

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

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

These are three examples of iterative gradient-based algorithms, the first of which gradient descent is used in this thesis to introduce the concept of algorithm analysis, the Lyapunov-based method and how it is implemented.

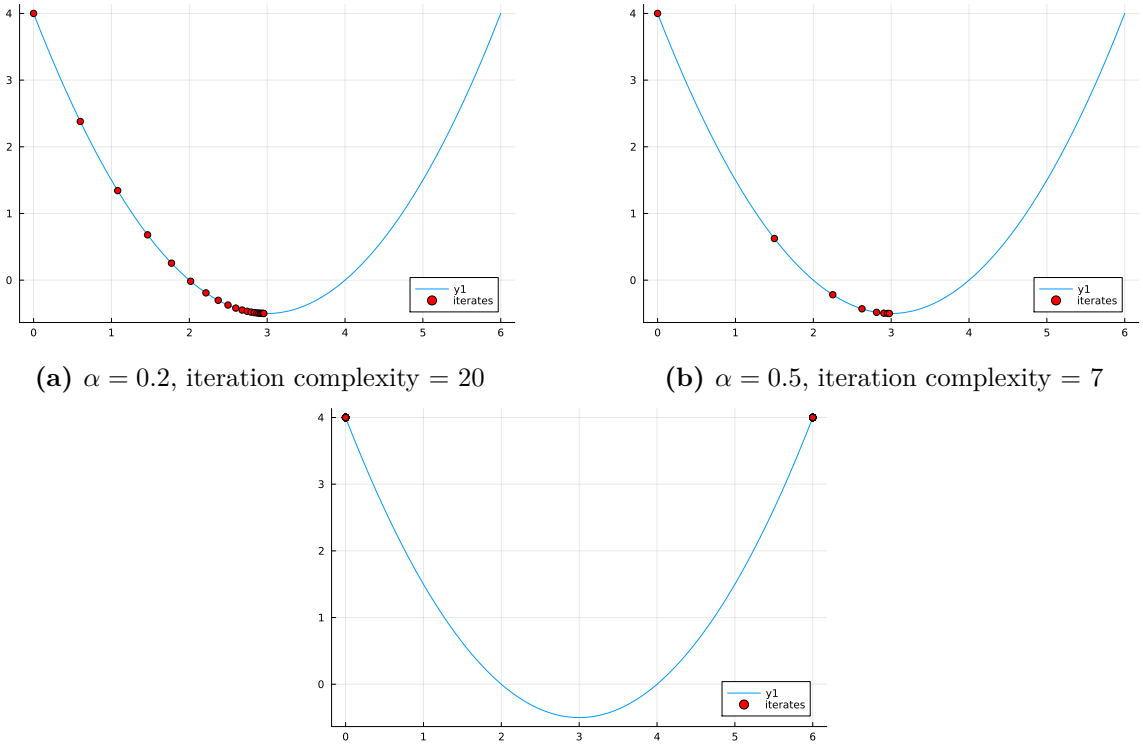
1.2 Algorithm analysis

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 (GD) equation (6.1), substituting step size α with values 0.2, 0.5, and 2, and picking a starting point of $x_0 = 0$, we can solve the 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:

It can be seen how different tunings on the same algorithm can achieve drastically different speed optimizing a function, or whether it can solve for the minimum at all. Considering there exist many other first order methods in addition to the three in 1.1, each infinitely adjustable by changing the step size or by changing the number of past iterations used, being able to predict how an algorithm will perform can speed up the process of finding the best performing algorithm which can find a more accurate solution can be found and in fewer iterations. And while this analysis 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 computationally expensive, any benefit of finding a superior algorithm at solving a problem is negated as said problem has



(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

already been solved. Additionally, any analysis result is applicable only to one function and cannot be reliably used to derive a first-order method's performance on any other problem. Another shortcoming of this simple analysis is that it is of a simple function where overshoot can easily be identified and the number of iteration needed is small. As the problem becomes larger and more complex, the benefit of choosing the most optimal algorithm available only increases. In the application of training large language and self-driving models, the training process has taken place for many years and will continue as more training data is available and the models' continous improvement is desired. 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 the energy needed.

Due to these limitations, it is more efficient to analyze algorithms' performance at solving a broader set of problems and without numerically optimizing any. As a result of their widespread application, popular iterative gradient-based algorithms have been extensively analyzed. A frequently cited attempt is the Adam algorithm [3], in which the performance of algorithms are compared using experiments and emperical evidence. There exists a different approach to quantifying the performance of algorithms, presented in [4], [5], and [6], which

aims to find a mathematically provable performance guarantee of an algorithm over a class of functions. 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 good or better at solving every function within said set.

1.3 Julia programming language

`AlgorithmAnalysis.jl` is written in 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 rivalling C, a language often used for its high efficiency [7]. Julia accomplishes this while being a high-level language with simple syntax rules that resembles existing popular languages, making it easy to develop and to understand.

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

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.

1.4 Analysis example

To perform analyze the performance of an algorithm using the `AlgorithmAnalysis.jl` package, the user needs to follow the folliowing 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.

In the Lyapunov-based method to analyzing algorithms and in `AlgorithmAnalysis.jl`, a function class is defined as every functions which share a trait. In example 1.2, the class of function is sector bounded between $m = 1$ and $L = 10$, which include any function f that

satisfy the condition:

$$(\nabla f(x) - m(x - x_*))^\mathsf{T}(\nabla f(x) - L(x - x_*)) \leq 0 \tag{1.6}$$

where x_* is the global minimum of the function f In the example code, the user:

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

        f = DifferentiableFunctional{R^n}()
        xs = first_order_stationary_point(f)
        f' ∈ SectorBounded(m, L, xs, f'(xs))

        x0 = R^n()
        x1 = x0 - α*f'(x0)

        x0 => x1

        performance = (x0-xs)^2
end

@show rate(performance)
```

Figure 1.2: Analysis example

- Define the class of function `f` and its ∇f, coded as `(f')` by calling one of the provided functions.
- Set the global minimum goal as a stationary point x_s.
- Define an initial state x_0 and the algorithm with which its next state x_1 is updated. In this example, the algorithm being analyzed is gradient descent with a step size $\alpha = 2/11$.
- Set 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 is ran automatically to return a rate of 0.6687164306640625. This is the converenge 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-oh" notation means that the performance measure 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, making it an accessible tool.

```
rate(performance) = 0.6687164306640625
```

Figure 1.3: Analysis result

1.5 Overview

Ater the introduction, in Chapter 2, the existing literature of approaches to analyzing algorithms and implementation into a program is presented. Chapter 3 discusses the Lyapunov-based mathematical method that `AlgorithmAnalysis.jl` utilizes. In Chapter 4, the main contribution of this thesis is presented: how a dommain-specific language is developed to support the package's functionality. Finally in Chapter 5, the analysis process and result are presented.

2

Literature Review

In recent years, numerous studies have been conducted comparing the performance of optimization algorithms, particularly in machine learning and deep learning contexts. These comparisons often focus on convergence speed, accuracy, and robustness across various models and datasets. In [3], Kingma and Ba presented the Adam (Adaptive Moment Estimation) algorithm. In order to prove its efficiency improvement above existing algorithms, the authors designed and conducted experiments where they are used to solve popular machine learning models and recorded the convergence rate of each. The result of these experiments provided emperical evidence that Adam performed better compared to its peers and gave a general idea of Adam's performance.

While the emperically proven performance of Adam has made it one of the most popular algorithms in machine learning, in [8], it is proved that under specific conditions, Adam is unable to converge on simple quadratic problems. This is something observational analysis did not demonstrate, highlighting its shortcomings and need for better analysis tools.

Of the approaches to automated algorithm analysis to analyzing algorithms which guarantee an algorithm's minimum performance, an influential framework is Performance Estimation Problem (PEP). In 2014, Drori and Teboulle first introduced the method of representing a class of function with constraints, reformulating the problem of analyzing an optimization method into a semidefinite program (SDP) [4]. The paper coined the term PEP and showed that by solving a convex semidefinite problem, a worst-case numerical bound on an algorithm's performance solving that class of function can be derived. This approach shares key motivations with interpolation-based Lyapunov analysis: both aim to provide tight, mathematically proven performance guarantees, and both rely on semidefinite formulations to model algorithm behavior within structured function classes.Taylor, Hendrickx and Glineur built upon this work by introducing the ideas of creating a finite representation for a class of

smooth strongly convex functions using closed-form necessary and sufficient conditions [2].

The Python library PEPit implements the PEP method into a software package as a practical way of using the framework, providing a high-level 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 that characterizes the worst-case behavior of the algorithm and solves it using an appropriate SDP solver. This automation significantly reduces the technical barrier to applying PEP, enabling researchers and practitioners 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, allowing users to test and compare different methods with minimal setup.

This work culminated in the implementation of PEP as a software package introducing a way to perform analyse an algorithm to derive its performance bound over a function class in the form of a guarantee that after a fixed number of iterates, how close the last iterate is to the goal. This implementation created PEPit [2], a Python package, and PESTO [9], a corresponding MATLAB toolbox. PESTO and PEPit follows the PEP methodology and first presented an automatic way to analyze gradient-based algorithms: Given an algorithm and problem class from a supported list, the programs can provide a guaranteed minimum convergence rate of the algorithm for every problems in the class.

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 systems in which the gradient of the objective function — a complex system — can be constrained to be in some class using IQCs, making IQCs one tool from *robust control* that can be applied to study robust stability. 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 [6], a highly influential paper in this area. While both [6] 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, making the analysis of algorithms which. In [6], authors Lessard,

Recht, and Packard demonstrate by using Lyapunov functions, a bound on an algorithm's convergence rate can be found by deriving a small and fixed size SDP.

The AlgorithmAnalysis.jl package implements the algorithm analysis approach presented by Van Scoy and Lessard in [11]. This Lyapunov-based approach to analysis 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 0 and 1 for the smallest one that can be guaranteed.

Another recent direction in the automatic analysis of optimization algorithms is the work in [12], which extends Lyapunov-based techniques to the analysis of primal-dual methods for linearly constrained convex optimization problems. Their approach uses a transformation based on a compact singular value decomposition (SVD) of the constraint matrix, allowing the algorithm's dynamics to be separated into components that are affected and unaffected by the constraints. This reformulation simplifies the structure of the problem and enables a more systematic application of Lyapunov functions to certify convergence guarantees. While the mathematical details differ from the unconstrained case, the overall methodology aligns with other interpolation-based Lyapunov frameworks in its use of convex function properties and matrix inequalities to automatically derive worst-case performance bounds. This work demonstrates the potential of automated analysis techniques beyond unconstrained settings and highlights the value of structural problem transformations in extending their applicability.

Similar to [11], another way to use Lyapunov function to perform automated algorithm analysis have introduced to analyze optimization algorithms in the form of:

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

In, [12], the authors proposed a framework for the automatic analysis of primal-dual algorithms used to solve linearly constrained convex optimization problems. Their approach uses a transformation based on a compact singular value decomposition (SVD) of the constraint matrix, allowing the algorithm's dynamics to be separated into components that are

affected and unaffected by the constraints. This reformulation simplifies the structure of the problem and enables a more systematic application of Lyapunov functions to certify convergence guarantees. While the mathematical details differ from the unconstrained case, the overall methodology is similar with the Lyapunov-based framework presented in [11] in its use of convex function properties and matrix inequalities to automatically derive worst-case performance bounds. This work proves Lyapunov function-based automated analysis can be applied beyond unconstrained settings and a special case algorithm that can be implemented into the AlgorithmAnalysis.jl package.

The main contribution of this thesis is the development of a software package `AlgorithmAnalysis.jl` that similar to PESTO and PEPit aims to provide an accessible and fast way to analyze the performance of first-order methods for a guaranteed convergence rate using the Lyapunov function-based approach instead of PEP.

By developing a domain-specific language inside the Julia programming language, Algorithm Analysis simplifies the process of defining an optimization algorithm, provides a systemic way to represent abstract concepts such as algorithm iterate or the gradient of an abstract function, and make the analysis of optimization algorithms more accessible to non-expert users.

Implemented as a domain-specific language (DSL) embedded in Julia, texttttAlgorithmAnalysis.jl enables users to describe optimization algorithms at a high level while internally handles the mathematical complexity of worst-case performance analysis. By introducing a systematic representation of algorithmic components such as iterates, gradients, and function oracles, while automating the generation of interpolation conditions and the Lyapunov function optimization problem, the package lowers the barrier to entry for non-experts and promotes experimentation, rapid prototyping, and reproducible research in algorithm design.

3

Lyapunov-based approach

Algorithm Analysis performs algorithm analysis by using the technique layed out by Van Scoy and Lessard in [11]. While Algorithm Analysis is a blackbox tool, understanding the mathematical approach on which it is based is a prerequisite to understanding the package's code and functionalities.

The technique is based on the idea that certifying whether a convergence rate of an algorithm optimizing a function can be guaranteed is itself an optimization problem. This approach, which will be discussed over the sections of this chapter, 1) represents the algorithm being analyzed in state-space form, 2) replaces the nonlinear gradient with constraints derived from interpolation conditions of a function class, 3) uses Lyapunov functions and constraints to form an optimization problem the solution to which certify whether a certain convergence rate can be guaranteed.

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 uses the gradient of the function to update an iterate or state — represented by x in equations (6.1), (1.3), and (1.4) — the gradient of which is used to define the next iterate. These algorithms can be reformulated 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. Figure 3.1 shows the block diagram of this view:

Here, G represents the LTI system, while y and u are input and output of the gradient nonlinearity. For example, (FG) equation (1.4) matches this representation if u_k is defined as $\nabla f(y_k)$. The algorithm can then be put into state-space representation with augmented

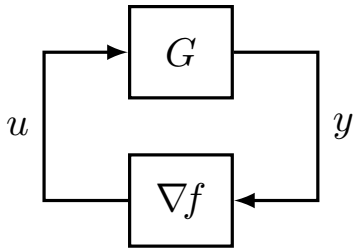


Figure 3.1: Block diagram representation of iterative algorithms

state $\xi_k = (x_k, x_{k-1})$ as:

$$\xi_{k+1} = \begin{bmatrix} (1+\beta) & -\beta \\ 1 & 0 \end{bmatrix} \xi_k + \begin{bmatrix} -\alpha \\ 0 \end{bmatrix} u_k, \tag{3.1a}$$

$$y_k = \begin{bmatrix} 1+\beta & \beta \end{bmatrix} \xi_k, , \tag{3.1b}$$

$$u_k = \nabla f(y_k) \tag{3.1c}$$

in which 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 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 (6.1), (1.3), and (1.4), these matrices are:

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

Beyond the three listed examples, any other first-order methods can be transformed into an LTI system represented by state-space matrices, enabling the formulation of Lyapunov functions in the next sections and forming the first step at transforming the algorithm analysis problem into a convex semidefinite program.

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 representing the gradient of the function cannot be efficiently solved. As a result, the Lyapunov-based approach replaces 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 Figure 3.2. Here, the nonlinear gradient block is replaced with a filter or dynam-

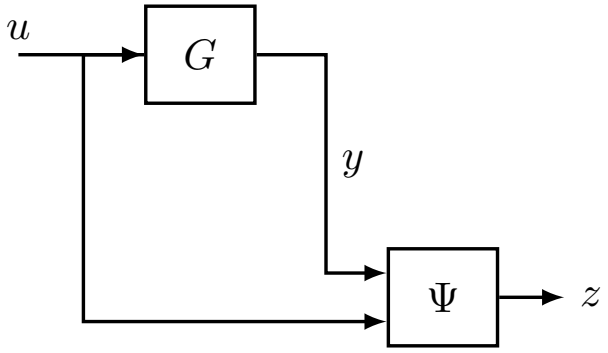


Figure 3.2: Block diagram of iterative algorithms with gradient replaced by interpolation condition filter

ical system Ψ. This dynamical system produces an output z as some quadratic function of its inputs y and u before applying some constraint to the output z. Both the filter and its corresponding constraints 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 pairs. These interpolation conditions depends on the characteristics of the function class. The analysis of any algorithm's performance at solving a function class is only possible if there exists interpolation conditions for that class.

While many function classes have interpolation conditions, this thesis and the Algorithm Analysis package focus on smooth strongly convex functions and their interpolation conditions, with the goal of analyzing the performance of algorithms at solving them, as many optmization problems such as linear regression or logistic regression in the machine learning field can be classified as convex optimization problems. This thesis uses the notation and definition of smooth strongly convex functions as [11], which uses the notation $F_{m,L}$ and define the function class as 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.

The interpolation conditions for L-smooth and m-strongly convex functions was first formu-

lated in [5] and reformatted in [11] as:

Theorem 1 ([5], Thm. 4; [11], 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 $i, j \in I$.

As an example, if we define x_0 as the initial iterate of an algorithm optimization a 1 strong 10 smooth convex function ($f \in F_{1,10}$), x_s as the minimizer, and only evaluate the gradient of f at x_k and x_s: $\nabla f(x_k)$ is the gradient of f at x_k and $\nabla f(x_s) = 0$ is the gradient of f at x_s (so $m = 1$, $L = 10$, and $I = \{s, k\}$), then the inequalities:

$$\begin{aligned} 18(f(x_k) - f(x_s)) - 10\|x_k\|^2 - 10\|x_s\|^2 + 20\langle x_k, x_s\rangle + 2\langle \nabla f(x_k), x_k\rangle \\ -2\langle \nabla f(x_k), x_s\rangle - \|\nabla f(x_k)\|^2 \geq 0 \\ 18(f(x_s) - f(x_k)) - 10\|x_k\|^2 - 10\|x_s\|^2 + 20\langle x_s, x_k\rangle + 20\langle \nabla f(x_k), x_k\rangle \\ -20\langle \nabla f(x_k), x_s\rangle - \|\nabla f(x_k)\|^2 \geq 0 \end{aligned} \tag{3.2}$$

when satisfied interpolates 1 strong 10 smooth convex functions. Using interpolation condition, the gradient of any function belonging to the associated function class is represented with an inequality, one of the two components of the convex semidefinite problem needed to analyze algorithms.

Gram matrix interpolation condition

It can be seen that other than the function values f_i and f_j, the left hand side of the interpolation condition consists of the norm and inner product of the interpolated points and gradients. These elements create a Gram matrix of real-valued elements with which the interpolation can be defined as a function of. Theorem 1 can be transformed into:

$$\sum_{i,j\in I} \operatorname{tr}\left(\begin{bmatrix} -mL & mL & m \\ mL & -mL & -m \\ m & -m & -1 \end{bmatrix} \begin{bmatrix} \|y_i\|^2 & \langle y_j, y_i\rangle & \langle u_i, y_i\rangle \\ \langle y_i, y_j\rangle & \|y_j\|^2 & \langle u_i, y_j\rangle \\ \langle y_i, u_i\rangle & \langle y_j, u_i\rangle & \|u_i\|^2 \end{bmatrix} \right) + 2(L-m)(f_i - f_j) \geq 0. \tag{3.3}$$

for all $i, j \in I$. We can then define $[x; u]$ as the Gram matrix of a set of vectors representing every interpolated points, which is also every state and gradient an algorithm uses to update its iterate

The elements of the Gram matrix are interpolable — meaning there exists vectors the norm and inner product of which are elements of a Gram matrix — if and only if the Gram matrix is positive semidefinite and the dimension n of the state vectors to be greater than or equal to the rank of the Gram matrix. Since the problem classes the algorithm being analyzed is abstract, the Lyapunov-based approach makes the assumption that each function in every class has sufficiently large dimension. The other condition on the Gram matrix is applied is similar to those created from interpolation conditions and will also be used to construct the semidefinite problem. The constraint for the Gram matrix associated with (3.2) is:

$$\begin{bmatrix} \|x_i\|^2 & \langle x_j, x_i\rangle & \langle \nabla f(x_i), x_i\rangle & \langle \nabla f(x_j), x_i\rangle \\ \langle x_i, x_j\rangle & \|x_j\|^2 & \langle \nabla f(x_i), x_j\rangle & \langle \nabla f(x_j), x_j\rangle \\ \langle x_i, \nabla f(x_i)\rangle & \langle x_j, \nabla f(x_i)\rangle & \|\nabla f(x_i)\|^2 & \langle \nabla f(x_j), \nabla f(x_i)\rangle \\ \langle x_i, \nabla f(x_j)\rangle & \langle x_j, \nabla f(x_j)\rangle & \langle \nabla f(x_i), \nabla f(x_j)\rangle & \|\nabla f(x_j)\|^2 \end{bmatrix} \succcurlyeq 0. \tag{3.4}$$

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, formed from the conditions on the Lyapunov functions and constraints 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 main goal of the program and the Lyapunov method it implement is to certify whether any given level of performance can be guaranteed, and uses *convergence rate* as a measure of performance, which is defined as the rate at which the performance measure $\|x_k - x_s\|^2$ decreases after each iteration. This definition can be expressed in equation form as, given any $k \geq 0$ as the iterate of an algorithm and define x_0 as the initial iterate of the algorithm, a proven convergence rate guarantee of ρ means:

$$\|x_k - x_s\|^2 \leq \rho^k \|x_0 - x_s\|^2. \tag{3.5}$$

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 and can be used to understand the behavior of a system. Under this definition, the dynamical system is the algorithm being analyzed, and two Lyapunov functions are used to certify whether or not a convergence rate can be guaranteed for a system. Consider the gradient descent algorithm and its representation in (6.1), define $\mathbf{x_k} = x_k - x_s$, the Lyapunov function takes the form:

$$V(\mathbf{x}) = \text{tr}(\mathbf{x_k}^T P \mathbf{x_k}) \tag{3.6}$$

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 which update its iterate using multiple past states, $\mathbf{x_k}$ would have to include every states 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 gradient descent example, if it is proven there exists some optimization variable P so that Lyapunov functions satisfy the conditions:

$$\|x_k - x_s\|^2 - V(\mathbf{x_k}) \leq 0, \tag{3.7a}$$

$$V(\mathbf{x_{k+1}}) - \rho V(\mathbf{x_k}) \leq 0 \tag{3.7b}$$

then the convergence rate of that algorithm is guaranteed to be faster than ρ 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 $\|x_k - x_s\|^2$, which will be referred to in the rest of this thesis as the performance measure, 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 after each iteration. Together, if the optimization variable P can be found so that the two inequalities are satisfied, a convergence rate of ρ or faster can be guaranteed. This is proven in the proof of Lemma 5 of [11], which can be modified to suit the gradient descent algorithm. For any $k \geq 0$:

$$\|x_k - x_s\|^2 \leq V(\mathbf{x_k}) \leq \ldots \leq \rho^k V(\mathbf{x_0}) \leq \rho^k (C_0 \|x_0 - x_s\|^2) \tag{3.8}$$

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

The Lyapunov functions conditions in their current form cannot be proven, as they are quadratic functions of abstract state vectors. In order to find a variable P that would satisfy these conditions using the Lyapunov method, we combine them with the left hand side

of the constraints detailed in Section 3.2 and transform them into functions linear in the optimization variables.

Here, it can be seen that while these Lyapunov functions that 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 $I_n \in \mathbb{R}^{n\times n}$, the identity matrix with dimension n, the same as the dimension of the state vectors, the Lyapunov function can be transformed into:

$$
\begin{aligned}
V(\mathbf{x_k}) &= \mathrm{tr}[P(x_k - x_s)(x_k - x_s)^\mathsf{T}] && \text{(3.9a)}\\
&= \mathrm{tr}[\begin{bmatrix} P & -P & 0 \\ -P & P & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \|x_k\|^2 & \langle x_s, x_k\rangle & \langle u_k, x_k\rangle \\ \langle x_k, x_s\rangle & \|x_s\|^2 & \langle u_k, x_s\rangle \\ \langle x_k, u_k\rangle & \langle x_s, u_k\rangle & \|u_k\|^2 \end{bmatrix}] \\
V(\mathbf{x_{k+1}}) &= \mathrm{tr}[P(x_{k+1} - x_s)(x_{k+1} - x_s)^\mathsf{T}] && \text{(3.9b)}\\
&= \mathrm{tr}[P(Ax_k + Bu_k - x_s)(Ax_k + Bu_k - x_s)^\mathsf{T}] \\
&= \mathrm{tr}[P \begin{bmatrix} A & B & -I_n \end{bmatrix} \begin{bmatrix} x_k \\ u_k \\ x_s \end{bmatrix} \begin{bmatrix} x_k \\ u_k \\ x_s \end{bmatrix}^\mathsf{T} \begin{bmatrix} A & B & -I_n \end{bmatrix}^\mathsf{T}] \\
&= \mathrm{tr}[\begin{bmatrix} A & B & -I_n \end{bmatrix} P \begin{bmatrix} A & B & -I_n \end{bmatrix}^\mathsf{T} \begin{bmatrix} \|x_k\|^2 & \langle x_s, x_k\rangle & \langle u_k, x_k\rangle \\ \langle x_k, x_s\rangle & \|x_s\|^2 & \langle u_k, x_s\rangle \\ \langle x_k, u_k\rangle & \langle x_s, u_k\rangle & \|u_k\|^2 \end{bmatrix}]
\end{aligned}
$$

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

$$
\|x_k - x_s\|^2 = \mathrm{tr}[\begin{bmatrix} 1 & -1 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \|x_k\|^2 & \langle x_s, x_k\rangle & \langle u_k, x_k\rangle \\ \langle x_k, x_s\rangle & \|x_s\|^2 & \langle u_k, x_s\rangle \\ \langle x_k, u_k\rangle & \langle x_s, u_k\rangle & \|u_k\|^2 \end{bmatrix}]. \qquad \text{(3.10)}
$$

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 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 simple grow in dimension. Here, it should be noted that the Gram matrices in the Lyapunov functions are the same as those in the constraints, as every states and inputs in the vector set the Gram matrix is defined from have to be interpolated. However, each constraint include 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 (3.7) with the constraints associated with the interpolation condtions, we first define the *linear form* of both.

Define $[x; u]$ as a vector containing every elements of the Gram matrix and the function values at each interpolated points: The Lyapunov functions and each constraint are linear functions of $[x; u]$. In the gradient descent example, this vector is defined as:

$$\begin{bmatrix} x \\ u \end{bmatrix} = \begin{bmatrix} \|x_k\|^2 \\ \langle x_s, x_k \rangle \\ \langle u_k, x_k \rangle \\ \langle x_k, x_s \rangle \\ \|x_s\|^2 \\ \langle u_k, x_s \rangle \\ \langle x_k, u_k \rangle \\ \langle x_s, u_k \rangle \\ \|u_k\|^2 \\ f(x_k) \\ f(x_s) \end{bmatrix}. \tag{3.11}$$

The linear form of conditions on the Lyapunov functions $L1$ and $L2$ can be defined as:

$$\|x_k - x_s\|^2 - V(\mathbf{x_k}) = L1 \begin{bmatrix} x \\ u \end{bmatrix} \tag{3.12a}$$

$$V(\mathbf{x_{k+1}}) - \rho^2 V(\mathbf{x_k}) = L2 \begin{bmatrix} x \\ u \end{bmatrix}. \tag{3.12b}$$

For the constraints associated with the interpolation condtions, each is derived and scaled by a nonnegative obtimization variable Λ before being combined with $L1$ and $L2$. For each constraint created by the interpolation conditions, an optimization variable Λ is created and scaled with. The linear form of the constraints associated with the interpolation conditions (3.3) is:

$$\sum_{i \in C} M_i(\Lambda_i) \begin{bmatrix} x \\ u \end{bmatrix} \tag{3.13}$$

where C is the set containing every constraints created in (3.3) and (5.3) 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 semidefiinite if it is a matrix. When

the constraints applied to these variables satisfied, equation (3.13) are nonnegative. At the same time, variables Λ_i enable the solver to search for Λ_i just as it does for P so that the final linear function is satisfied. 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 (3.2) in the gradient descent 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:

$$L_1 + \sum_{i \in C} M_i(\Lambda_i^1) = 0 \tag{3.14a}$$

$$L_2 + \sum_{i \in C} M_i(\Lambda_i^2) = 0. \tag{3.14b}$$

For any convergence rate ρ, if some parameter P, Λ_i^1 and Λ_i^2 can be found which would solve the linear functions in (3.14), the Lyapunov function conditions specified in (3.7) is satisfied and that performance guarantee 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 $[x; u]$, (3.14) can be multiplied with $[x; u]$. If the linear functions (3.14) equal zero, and multiplying the linear form of the constraints by $[x; u]$ give us the interpolation conditions, which are constrained to be ≥ 0, it must mean that $L1 \begin{bmatrix} x & u \end{bmatrix}^\mathsf{T}$ is ≤ 0:

$$(L1 + \sum_{i \in C} M_i(\Lambda_i^1)) \begin{bmatrix} x & u \end{bmatrix}^\mathsf{T} = 0 \tag{3.15a}$$

$$\underbrace{L1 \begin{bmatrix} x & u \end{bmatrix}^\mathsf{T}}_{\leq 0} + \underbrace{\sum_{i \in C} M_i(\Lambda_i^1) \begin{bmatrix} x & u \end{bmatrix}^\mathsf{T}}_{\geq 0} = 0 \tag{3.15b}$$

$$\tag{3.15c}$$

and similarly, $L2 \begin{bmatrix} x & u \end{bmatrix}^\mathsf{T}$ is ≤ 0.

$$
\begin{aligned}
(L1 + \sum_{i \in C} M_i(\Lambda_i^1)) \begin{bmatrix} x & u \end{bmatrix}^\mathsf{T} &= 0 \\
\underbrace{L1 \begin{bmatrix} x & u \end{bmatrix}^\mathsf{T}}_{\leq 0} + \underbrace{\sum_{i \in C} M_i(\Lambda_i^1) \begin{bmatrix} x & u \end{bmatrix}^\mathsf{T}}_{\geq 0} &= 0 \\
(L2 + \sum_{i \in C} M_i(\Lambda_i^1)) \begin{bmatrix} x & u \end{bmatrix}^\mathsf{T} &= 0 \\
\underbrace{L2 \begin{bmatrix} x & u \end{bmatrix}^\mathsf{T}}_{\leq 0} + \underbrace{\sum_{i \in C} M_i(\Lambda_i^1) \begin{bmatrix} x & u \end{bmatrix}^\mathsf{T}}_{\geq 0} &= 0
\end{aligned}
$$

.

Together, $L1 \begin{bmatrix} x & u \end{bmatrix}^\mathsf{T} \leq 0$ and $L2 \begin{bmatrix} x & u \end{bmatrix}^\mathsf{T} \leq 0$ form the conditions on the Lyapunov functions, and if (3.14) is solved, it means that the convergence rate is feasible.

Since we know how to certify whether a convergence rate guarantee is feasible, the worst-case performance of the system or fastest convergence rate that can be guaranteed for said system can be found by performing bisection search for the smallest value ρ between 0 and 1 with which the optimization problem is feasible within some margin of error.

4

Code Components

Algorithm Analysis derive a guaranteed worst-case convergence rate by following the set of instructions presented in Equation (3.7)to create and solve an optimization problem and derive a performance certification. Implenmenting a mathematical procedure as code presents a list of challenges which includes being able to understand and differentiate between variables, represent concepts such as gradients or states, and formulating and solving a convex optimization problem, while keeping the users' interaction with the program simple. This chapter goes into the code that constitutes Algorithm Analysis and enables these functionalities.

4.1 Expressions

Expressions are data structures used to represent mathematical variables and enable the implementation of the concepts presented in Chapter 3 into a computer program. Expressions can either be vectors or scalar in an inner product space, making them the smallest building block with which concepts such as gradient, constraints, or states are built.

Variable and decomposition

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 and represents either a vector or scalar. Its decomposition is itself.

Decomposition expression represents scalars and is some linear combination of scalar variable or decomposition expressions. Its decomposition is a dictionary containing

```
@algorithm begin
        x0 = R^n()
        y0 = R^n()^2
end
@show(x0)
x0 = x0

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

@show(y0)
y0 = y0

Scalar in R^n()
Label: y0
Oracles: LinearFunctional{R^n}
```

Figure 4.1: Example of a variable expression representing a vector

how many of each expression that form the decomposition expression.

```
@show(x0 + 2xs)
x0 + 2xs = 2 xs + x0

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

Figure 4.2: Example of a decomposition expression

Gram matrix

The analysis process include creating and applying constraints on Gram matrices containing expressions. To save memory and improve computational efficiency, a Gram expression struct is used to represent Gram matrices. It contains the vector of vector expressions the outer product of which forms the Gram matrix. The function `evaluate(g::Gram)` takes a Gram matrix expression as argument and return the Gram matrix.

```
@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. In Figure 4.1, expressions are defined to be in R^n, an inner product space pre-defined in the package which encompasses n-dimensional real numbers.

Algorithm Analysis supports operations that characterize inner product spaces between expressions. The examples that demonstrate these operations use vector expressions `x0` and `y0` in Figure 4.1.

Addition or subtraction between expressions

Vectors and numbers can be added together in an inner product space. In an addition operation, if both expressions of the operation posess a 'value', they are added to create the value of a new resulting expression. Otherwise, the result is a new expression whose decomposition is the merging of the decomposition dictionaries of the expressions in the operation.

```
a = x0+y0-x0-x0
@show(a)
a = a

Vector in R^n
        Label: a
        Decomposition: y0 - 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 in an inner product space. The Algorithm Analysis package performs the multiplication or division of an expression by scaling its value if is a variable expression and scale the value of its decomposition dictionary if it is a decomposition expression.

```
c = x0*-3 + y0*2
@show(c)
c = c

Vector in R^n
        Label: c
        Decomposition: 2 y0 - 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 and stored in the `associations` field, allowing the program to keep track of whether an expression is the tranpose of another. In the AlgorithmAnalysis.jl package, the transpose of a transpose expression returns the original expression.

```
@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 product operation between two vectors

In an inner product space, the inner product operation of two vectors is possible and result in a scalar. For example, the inner product of vectors y0 and x0, denoted $\langle y0, x0 \rangle$, is calculated as $x0^{\mathsf{T}} * y0$.

```
inner = x0'*y0
@show(inner)

Scalar in R
        Label: <y0,x0>
        Oracles: x0*
```

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

Squared norm

An expression of the normed vector vpace type can be squared to produce a an inner product space expression.

```
norm = x0^2

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

Figure 4.8: Example of norm of vector expression

Outer product

The outer product of two vectors of expression in the same inner product space can be found using the $\otimes$ function. It is used to construct Gram matrices from vectors of expressions, playing a crucial role in formulating the interpolation conditions and Lyapunov inequalities described in Chapter 3.

```
op = [x0, x1] ⊗ [x0, x1]

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

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

Figure 4.9: 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 condition, the package uses oracles, data structures containing the relation and constraint information between expressions. Each oracle represent a class of function and can only exist if there exist interpolation conditions for said class.

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

Its corresponding gradient oracle `f'` is also created and given the property of being the gradient of `f`. The oracle can be sampled at different expressions. For example, 4.10 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 Figure 1.2

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

Figure 4.10: Example of sampling an oracle

An oracle stores information about every expression at which it has been sampled along with their corresponding output. Figure 4.11 shows the the input and output information of the

oracle `f'` defined in 4.10.

```
inputs_outputs(f')
```

```
(R^n[xs, x0], R^n[0, ∇f(x0)])
```

Figure 4.11: Example of the inputs outputs information of an oracle

In Figure 4.10, by defining the DifferentiableFunctional oracle `f` as belonging to the 1 smooth 10 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 4.12: 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. Figure 4.13 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 3.2.

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

```
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 4.13: Example of constraints created by sampling an oracle

algorithms and other composite optimization frameworks. Each oracle can also be associated with a wide variety of mathematical properties. Oracles not just limited to representing smooth strongly convex function class, 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. This generality allows AlgorithmAnalysis.jl to support a broader class of algorithms and problem formulations beyond first-order methods for smooth convex functions.

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 have 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 4.14: Example of a transpose oracle

4.4 States

Algorithm Analysis represents the states of an algorithm using expressions. The user inputs the algorithm be defining intial 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 labeling macro, and the next state is stored in the `next` field of a state expression.

```
x0 = Rⁿ()
x1 = x0 - α*f'(x0)
x0 => x1
@show x0

Vector in Rⁿ
  Label: x0
  Next: x1
  Associations: Dual => x0*
```

Figure 4.15: Example of the `=>` operation

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

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)`.

4.5 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 4.2, Julia base functions `+`, `-`, `*`, `/`, and `^` are overloaded to support algebraic operations. Therefore, the package execute the function `x0+y0-x0-x0` in

```
@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 4.16: Example of automatic next state assignment for decomposition expressions

```
@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 4.17: Example of automatic next state assignment for oracle output expressions

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 follow math notation in that both the transpose of a function and the gradient of a function is denoted using the operation `'` in the package. With the `adjoint` function, the package allow the user to refer to the gradient of a function when using it with a supported DifferentiableFunctional or TwiceDifferentiableFunctional oracle. When used on a vector expression however, the `'` operation is used to denote the transpose of a vector expression.

- **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 samples an oracle. For example, we sample a DifferentiableFunctional oracle `f` at expression `x0` simply by calling `f(x0)`.
- **Oracle property**: The interpolation conditions on which the analysis depends is defined according to which function class we are analyzing the algorithm over. Therefore, the package uses the operation $\in$ to denotes an oracle representing the function or gradient belongs to one of the package's supported classes. In figure 4.18, 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{Rⁿ}()
f ∈ SmoothStronglyConvex(m, L)
```

Figure 4.18: Example of the $\in$ operation

4.6 Labeling expressions

Algorithm Analysis 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 define 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.

However, an expression defined outside of the labeling macro, as shown in Figure 4.20, would be given a default label.

The `@algorithm` macro is also responsible for simplying the process of assigning an expression as the 'next' of another. While in regular Julia syntax, the code `x0 => x1` simply creates

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

@show x0

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

Figure 4.19: Example of a labeled expression

```
x3 = R^n()
Vector in R^n
Label: Variable{R^n}
Associations: Dual => LinearFunctional{R^n}
```

Figure 4.20: Example of an unlabeled expression

a pair between the 2 variables, when executed inside the macro automatically assigns the expression `x1` as the next state of `x0`.

Default labels

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

- **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 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 a 2 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{R^n}()
f ∈ SmoothStronglyConvex(1, 10)
end

@show f

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

Figure 4.21: Example of an oracle's description

Hash-based ordering

The ordering of vector inner products can result in variables with different labels that represent the same objects. For example, expressions labeled `<y0, x0>` and `<x0, y0>` are both created by taking the inner product of `x0` and `y0` but is not understood by the package as the same expression. Since this affects the robustness and accuracy of the analysis, a function is used to enforce a standard ordering based on hashing: 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.

4.7 Constraints

During the analysis process, constraints are created by interpolation conditions in section 3.2, 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 which define the expression to be in a cone. The list of constraints supported include:

```
inner0 = x0'*y0
inner1 = y0'*x0
@show(inner0)

Scalar in R
        Label: <y0,x0>
        Oracles: x0*

@show(inner1)

Scalar in R
        Label: <y0,x0>
        Oracles: x0*
```

Figure 4.22: Example of consistent inner product labelling order

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. Figure 4.13 is an example showing 3 constraints. 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 is 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 4.23: Example of user added constraint

4.8 Performance measure

Part of the required inputs to perform analysis is the performance measure, the convergence rate of which the package finds the worst-case guarantee through algorithm analysis. In Figure 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.

4.9 JuMP modeling language

Analysis of an algorithm's performance optimizing a function is itself an optimization problem as defined in Section 1.1, in which the function being minimized is the convergence rate while the constraints of the problem are the constraints created by the oracle and the user. Therefore, in order to formulate and solve optimization problems, the Algorithm Analysis package uses JuMP [13], 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 elements of a problem would be 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 solver, which are packages containing algorithms to find solutions to the optmization problem being formulated. While examples of analysis shown in this thesis uses the SCS [14] 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 holds, indicating whether or not the convergence rate chosen can be guaranteed.

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. Figure 4.24 shows how JuMP can

optimize the problem.

```
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 4.24: Example of JuMP minimizing an optimization problem

However, as discussed in Section 3.3, Algorithm Analysis find the performance guarantee by finding whether there exists an optimization varible with which the constraints of the problem are satisfied. To demonstrate this functionality, we can use the model used in Figure 4.24 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 exist 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 return the termination code `INFEASABLE` to indicate that no solution can be found in Figure 4.25.

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

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

Figure 4.25: Example of JuMP certifying the feasibility an optimization problem

5

Analysis Process and Result

As descriped 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 to a minimum at a given rate, while taking into account the constraints created by the interpolation conditions or the user. This problem is turned into an optimization problem, the feasibility of which proves that a convergence rate can be guaranteed. The analysis process takes place in 4 steps:

1. The user's input necessary for to certify a convergence rate — the algorithm, the constraints and the performance measure — is collected automatically to form a systematic charaterization 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 the a linear function of the initial states and inputs. These variables represents 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 optimial with which the problem is feasible, which proves whether a certain convergence rate is feasible for a given problem.

4. The above 3 steps are repeated with different convergence rates as the program search for the smallest feasible convergence rate using bisection search.

This chapter details the analysis process of analyzing gradient descent's performance at optimizing any 1-10 sector bounded function as shown in Figure 1.2, including how these

steps presented in Chapter 3 are performed and how the optmization problem is formed and solved to derive worst-case performancce 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 variables that is part of the analysis problem using a recursive function.

This process start with the system creating a set of expressions present in the performace measure, a set of their associated oracles, and a list of constraints associated with found expressions and oracles. The original set of expression 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 are 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 1)

```
@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 2)

Pruning Gram matrix constraints

Due to how the package uses a recursive function to create the Gram matrix constraints which 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 on Gram matrices that are the outer product of vectors of expressions which are a subset of or equal to the the vector of expressions whose outer product we wish to apply a constraint on. This makes the additional constraints redundant as they are automatically satisfied as long as the constraint on the Gram matrix containing every interpolated inner product expressions is satisfied.

But while they do not affect accuracy of the result, the redundant constraints introduce more complexity to the analysis process and slowing it down. To prevent this therefore the package prune the list of constraints to remove any redundant constraints on 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 expressions that has been created during the input process, define the initial state vector `x` as every real expressions which contain another expression in its next field.

```
x  = collect(v for v ∈ vars if !ismissing(next(v)) && v isa R)
4-element Vector{R}:
    |x0|²
    |xs|²
    <x0,xs>
    f(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: Updated state and 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 number to create every expressions required to form the linear matrix inequality. The transformation of a decomposition expression of Gram matrix expression into such real number matrix will be refered to as the linear form of an expression and is crucial to creating the semidefiinite problem in JuMP.

The package now transform 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 (3.7), 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 user when defining the algorithm to be analyzed start by first defining an initial state, and it is updated using the gradient of the function taken at some point. In Figure 1.2 the initial and updated states are `x0` and `x1` respectively.

The Lyapunov functions can then be formed according to equations (3.9). This is done by the package by first separating every real scalar expressions, which mean inner product and function value expressions and exclude 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 in Section 5.2, we have already defined `x` as the initial state: every real scalar expressions 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}:
    <x0,xs> - 0.18181818181818182 <∇f(x0),xs>
    -0.18181818181818182 <∇f(x0),x0> + 0.03305785123966942
      ↪ |∇f(x0)|² - 0.18181818181818182 <x0,∇f(x0)> + |x0|²
    -0.18181818181818182 <xs,∇f(x0)> + <xs,x0>
    |xs|²
```

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

It is worth noting that while the gradient descent algorithm updates using only one state and evaluate the gradient at that state, if an algorithm updates using multiple state vectors or the gradient at an interpolated point consisting of multiple state vectors, such is the case in the Fast-gradient and Heavy-ball algorithms, these vectors will also have to be defined. The resulting additional expressions will still be collected in the step detailed in Section 5.1 and successfully separated due to the automatic next state assignment detailed in section Section 4.4.

The initial state real expressions `x` defined in Fig. 5.2 and the updated state real expressions $\mathtt{x}^+$ defined in Fig. 5.6, is 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

Following the steps presented in (3.7) and (3.9) 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 variable that will be optimized to produce a solution. The code that create the JuMP model and the formulation of the Lyapunov functions is presented in Figure 5.9.

```
X⁺ = linearform([x; u] => x⁺)
4x9 Matrix{Real}:
1   0   0   0    0         0         -0.1818    0          0
0   1   0   0   -0.1818    0          0         0.03306   -0.1818
0   0   1   0    0        -0.1818    0         0          0
0   0   0   1    0         0          0         0          0
```

Figure 5.8: Linear form state matrix x^+

```
# 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: Linear form state matrix x

5.4 Constraints

As presented in 4.7, constraints are created from sampled oracles' interpolation condition are expressions constrained to be in a cone. These constraints are linearized and added to the optimization in 2 steps:

Optimization variable multipliers For each constraint i, two optimization variables λ_i and μ_i, 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 variable will be a matrix sharing the same size with the matrix constrained. 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 problem created 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.

Linear form of constraints The linear form of each constraint scaled by the multiplier is created and added to the Lyapunov functions.

If the expression constrained is a single real scalar, the linear form of the constraint is derived similarly to the linear form of the performance measure or state space matrices but scaled by the multiplier. Suppose we have the constraint $(x0 - xs)^2 \geq 0$ and matrix $\begin{bmatrix} x \\ u \end{bmatrix} = \begin{bmatrix} |x0|^2 \\ < xs, x0 > \\ |xs|^2 \end{bmatrix}$, the linear form of the constraint in terms of $\begin{bmatrix} x \\ u \end{bmatrix}$, denoted as M would be:

$$\lambda * (x0 - xs)^2 = M * \begin{bmatrix} |x0|^2 \\ < xs, x0 > \\ |xs|^2 \end{bmatrix} \tag{5.1a}$$

$$M = \begin{bmatrix} \lambda & 2\lambda & \lambda \end{bmatrix} \tag{5.1b}$$

If the expression constrained and its corresponding multiplier are vectors of expression, the linear form of the constraint is derived as the linear form of the inner product between the multiplier vector and the constraint expression vector. Suppose we have a constraint vector $\begin{bmatrix} (x0 - xs)^2 \\ (x0 - xs)^2 - 3 * |xs|^2 \end{bmatrix} \geq 0$ and the same $\begin{bmatrix} x \\ u \end{bmatrix}$ matrix as (5.1), the linear form of the constraint in terms of $\begin{bmatrix} x \\ u \end{bmatrix}$, denoted as M would be:

$$\begin{bmatrix} \lambda & \lambda \end{bmatrix} * \begin{bmatrix} (x0 - xs)^2 \\ (x0 - xs)^2 - 3 * |xs|^2 \end{bmatrix} = M * \begin{bmatrix} |x0|^2 \\ < xs, x0 > \\ |xs|^2 \end{bmatrix} \tag{5.2a}$$

$$M = \begin{bmatrix} \lambda & -\lambda & \lambda \\ \lambda & -\lambda & -2\lambda \end{bmatrix} \tag{5.2b}$$

And if the expression constrained and its corresponding multiplier are matrices, the linear form of the constraint is the linear form of the trace of the matrix multiplication between the multiplier and the constraint expression. For the Gram matrix in (5.3) which is constrained

to be positive semidefinite, its linear form would be:

$$tr(\lambda \begin{bmatrix} ||x0||^2 & \langle xs, x0\rangle & \langle \nabla f(x0), x0\rangle \\ \langle x0, xs\rangle & ||xs||^2 & \langle \nabla f(x0), xs\rangle \\ \langle x0, \nabla f(x0)\rangle & \langle xs, \nabla f(x0)\rangle & ||\nabla f(x0)||^2 \end{bmatrix}) \geq 0 \tag{5.3}$$

Where λ is a 3x3 JuMP variable. In all three cases, for each constraints, 2 identical linear form matrices are created, one scaled by λ and added to the first Lyapunov function and the other by μ and added to the second Lyapunov function. This completes the final linear matrix inequalities as defined in (3.14).

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.

Using the definition of the convergence rate in (3.5), a ρ value of 1 means the algorithm cannot be guaranteed to converge and a convergence rate of 0 means the algorithm is guaranteed to converge after a finite number of iterations. In order to find the worst-case performance rate, the program performs 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 and Future Work

6.1 Numerical result validation

The `AlgorithmAnalysis.jl` package has been tested across a range of first-order optimization algorithms over the m-strong L-smooth convex function class to verify its ability to automatically generate accurate worst-case performance guarantees. In the test cases presented in this section, performance guarantees produced by the package matches to the mathematically produced results in [11], proving that the package successfully implement Lyapunov-based algorithm analysis.

For every plots in this section, 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. This ensures uniform coverage across orders of magnitude, capturing both small and large condition numbers with equal density in log space. The convergence rate produced by the package is plotted it on the y-axis, while the x-axis plots the condition number L/m of the tested (m, L) values.

Gradient descent

Continuing the example in Figure 1.2, we plot the package's derived worst-case convergence rate guarantee of the gradient descent algorithm with step size $\alpha = 2/(L+m)$ at optimizing 12 different m-strong L-smooth convex function classes, using the 12 sampled L values. The plot of the result produced is presented in Figure 6.1.

We have also tested the same gradient descent algorithm over 12 (m, L) sector-bounded function classes. 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 presented in Figure 6.2 and the plot of the result produced is presented in Figure 6.3. We expect the analysis result to be identical to that of gradient descent over m-strong L-smooth convex

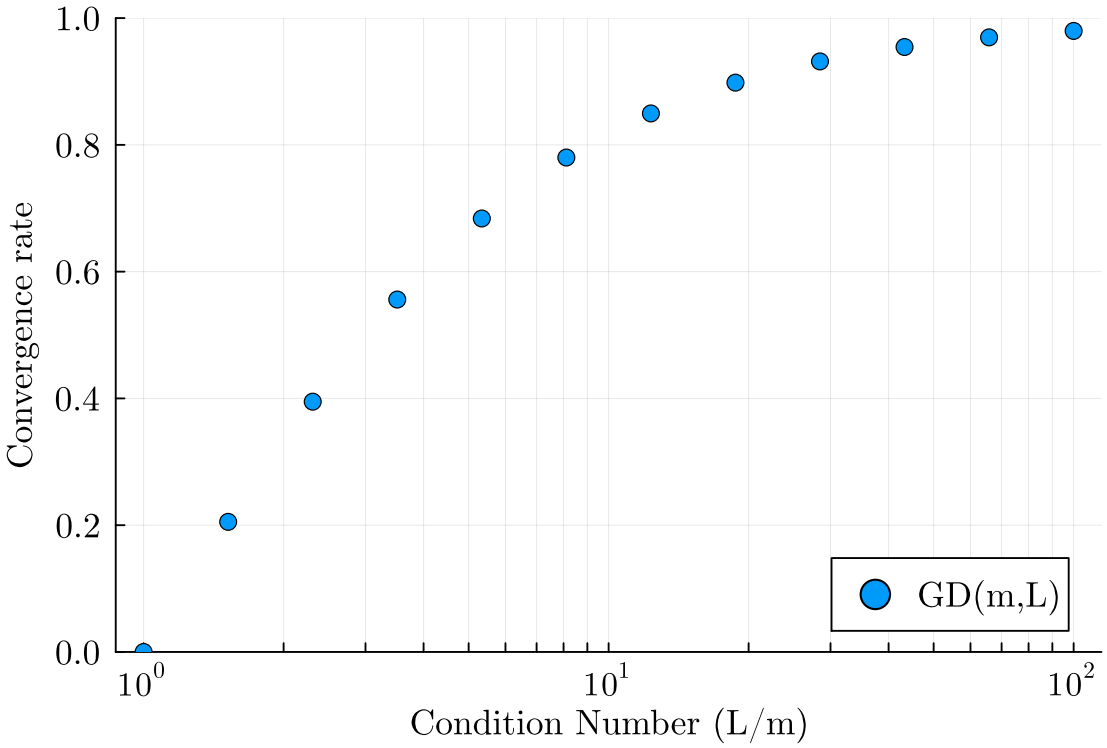


Figure 6.1: Convergence rate guarantee of gradient descent over sector smooth strongly convex classes

function classes.

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

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

Fast gradient

We tested the fast gradient algorithm with step sizes $\alpha = 4/(3L + m)$ and $\beta = (\sqrt{3L + 1} - 2)/(\sqrt{3L + 1} + 2)$. The code used to find the worst-case convergence rate guarantees of the

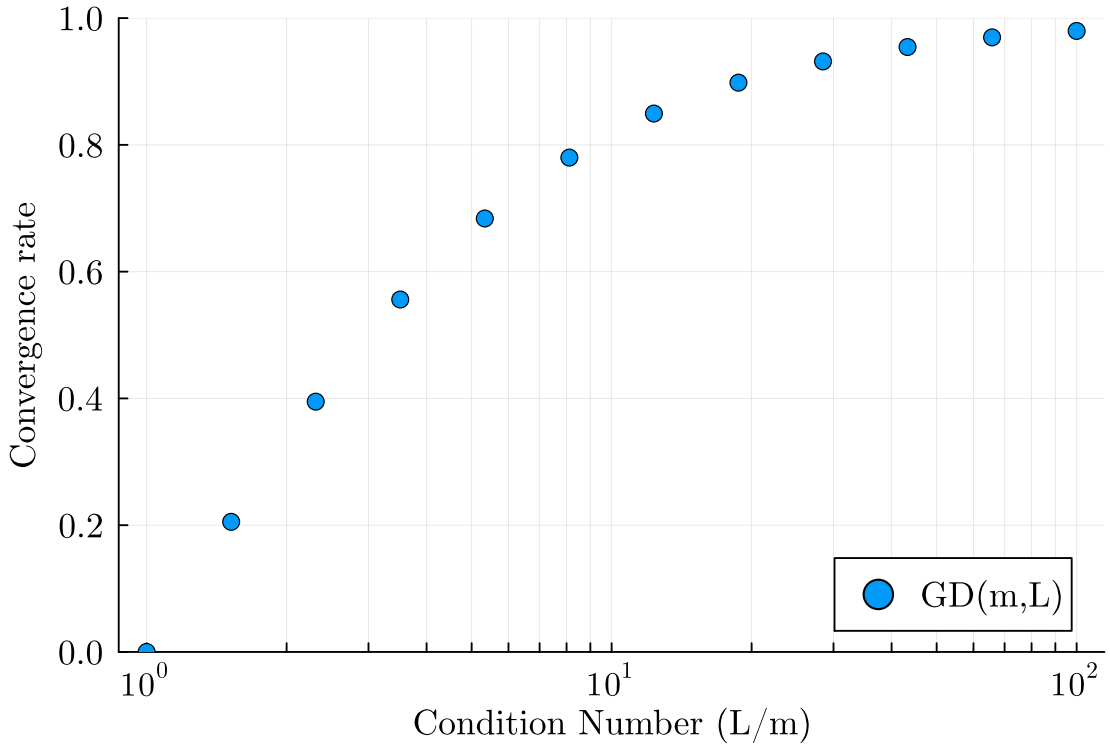


Figure 6.3: Convergence rate guarantee of gradient descent over sector bounded function classes

fast gradient algorithm at optimizing m-strong L-smooth convex function classes is presented in Figure 6.4 and the produced plot is presented in Figure 6.5.

Heavy ball

We tested the heavy ball algorithm with the following step sizes:

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

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 is presented in Figure 6.6, and the resulting plot is shown in Figure 6.8.

Triple momentum

We tested the triple momentum algorithm, which was developed by Van Scoy, Freeman, and Lynch in [15] to be the fastest known globally convergent first-order algorithm at optimizing

```
α = 4/(3*L+m); β=(sqrt(3*L+1)-2)/(sqrt(3*L+1)+2)
@algorithm begin
    f = DifferentiableFunctional{R^n}()
    xs = first_order_stationary_point(f)
    f ∈ SmoothStronglyConvex(m, L)
    x0 = R^n()
    x1 = R^n()
    y1 = x1 + β*(x1 - x0)
    x2 = y1 - α*f'(y1)
    y2 = x2 + β*(x2 - x1)
    x3 = y2 - α*f'(y2)
    x0 => x1
    x1 => x2
    x2 => x3
    performance = (y1-xs)^2
end
@show rate(performance)
```

Figure 6.4: Analysis of FG and m-smooth L-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 is determined by the condition number $k = L/m$ when optimizing m-smooth L-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 [15] that the performance guarantee matches the function $1 - \sqrt{m/L}$, which is also plotted in along side the rates produced by the package in Figure **??**. The code used to find the worst-case convergence rate guarantees of the triple momentum algorithm is presented in Figure 6.9.

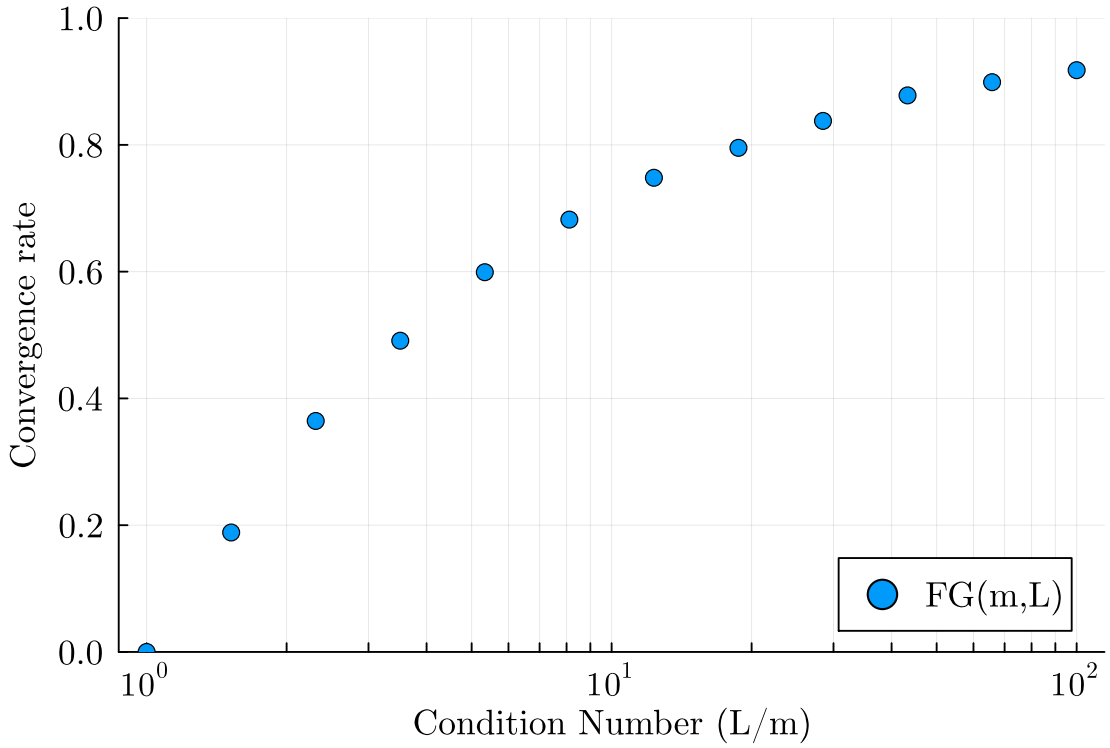


Figure 6.5: Convergence rate guarantee of fast gradient over smooth strongly convex function classes

```
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)
    x3 = x2 - α*f'(x2) + β*(x2-x1)
    x0 => x1
    x1 => x2
    x2 => x3
    performance = (x0-xs)^2
end
@show rate(performance)
```

Figure 6.6: Analysis of HB and m-smooth L-strongly convex functions

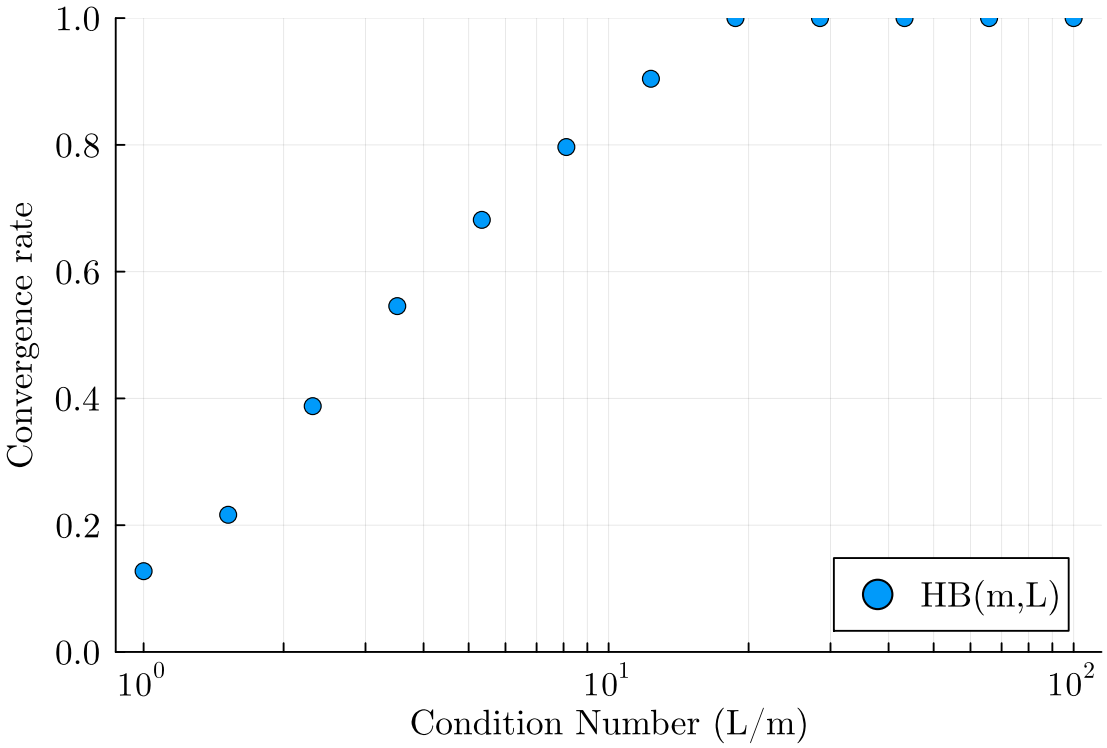


Figure 6.7: Convergence rate guarantee of heavy ball over smooth strongly convex function classes

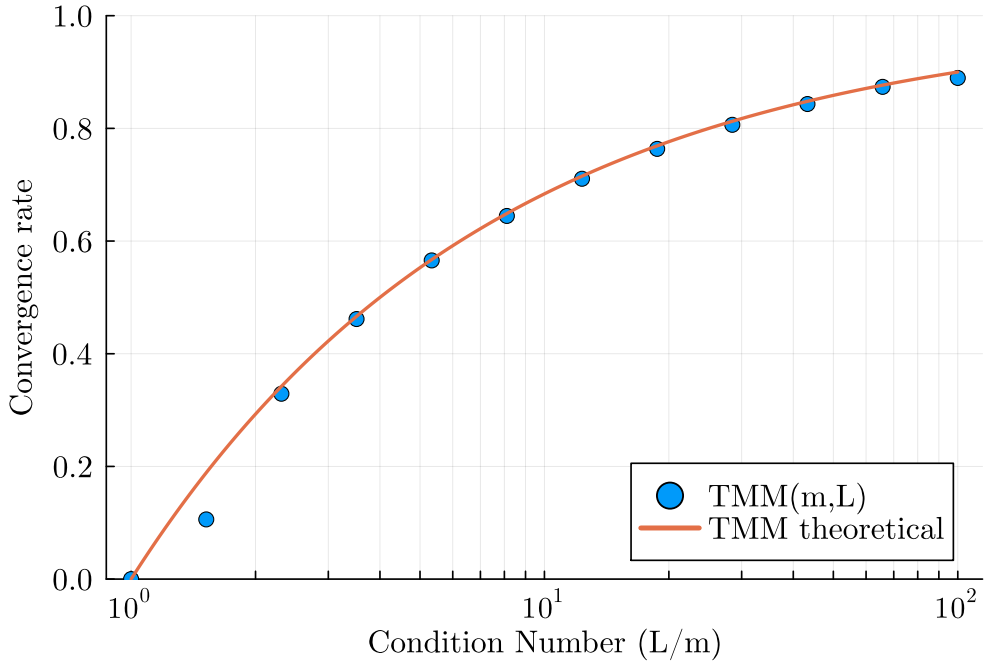


Figure 6.8: Convergence rate guarantee of triple momentum over smooth strongly convex function classes

```
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)
    y2 = (1+gamma)*x2 - gamma*x1
    x3 = (1+β)*x2 - β*x1 - α*f'(y2)
    x0 => x1
    x1 => x2
    x2 => x3
    performance = ((1+delta)*x2 - delta*x1 -xs)^2

@show rate(performance)
```

Figure 6.9: Analysis of HB and m-smooth L-strongly convex functions

6.2 Future work

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 other methods to test compatibility. Future work should focus on systematically 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 fucntion-based approach and highlight any potential limitations to be remedied and improved.

Additionally, the program currently does not support automatic "lifting dimension", a step of the Lyapunov method to algorithm analysis in [11]. While the user can manually add a lifting dimension by defining the analyze algorithm using more updates than the required 2. A function which can create these extra updated states, label them, and add them to the analysis process can allow the user to implement lifting dimension having only to enter how many extra updated states they wish to use. The lifting dimension will increase the

size of the Lyapunov functions and introduce additional interpolated points and therefore additional constraints. As a result, the optimization problem we are solving to certify a convergence rate will becore more complexto the optimization problem while this research work is to implement this step and measure its effect on the tightness of the convergence rate guarantee.

References

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

[2] 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.

[3] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[4] Yoel Drori and Marc Teboulle. Performance of first-order methods for smooth convex minimization: a novel approach, 2012.

[5] Adrien B. Taylor, Julien M. Hendrickx, and François Glineur. Smooth strongly convex interpolation and exact worst-case performance of first-order methods, 2016.

[6] 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.

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

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

[9] Adrien B. Taylor, Julien M. Hendrickx, and François Glineur. Performance estimation toolbox (PESTO): Automated worst-case analysis of first-order optimization methods. In *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, pages 1278–1283, 2017.

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

[11] Bryan Van Scoy and Laurent Lessard. A tutorial on a Lyapunov-based approach to the analysis of iterative optimization algorithms. In *IEEE Conference on Decision and Control*, 2023.

[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] 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, 2023.

[14] Brendan O'Donoghue, Eric Chu, Neal Parikh, and Stephen Boyd. Conic optimization via operator splitting and homogeneous self-dual embedding, June 2016.

[15] 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.