# 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. . .

# Acronyms

| | |
|---|---|
| **FG** | Fast Gradient |

# 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:

$$\begin{aligned} &\text{minimize} \quad f(x) && (1.1a) \\ &\text{subject to} \quad x \in X && (1.1b) \end{aligned}$$

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 $\alpha$ is the step size, an adjustable parameter of the algorithm. The stepsize $\alpha$ 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 $\beta$ 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.

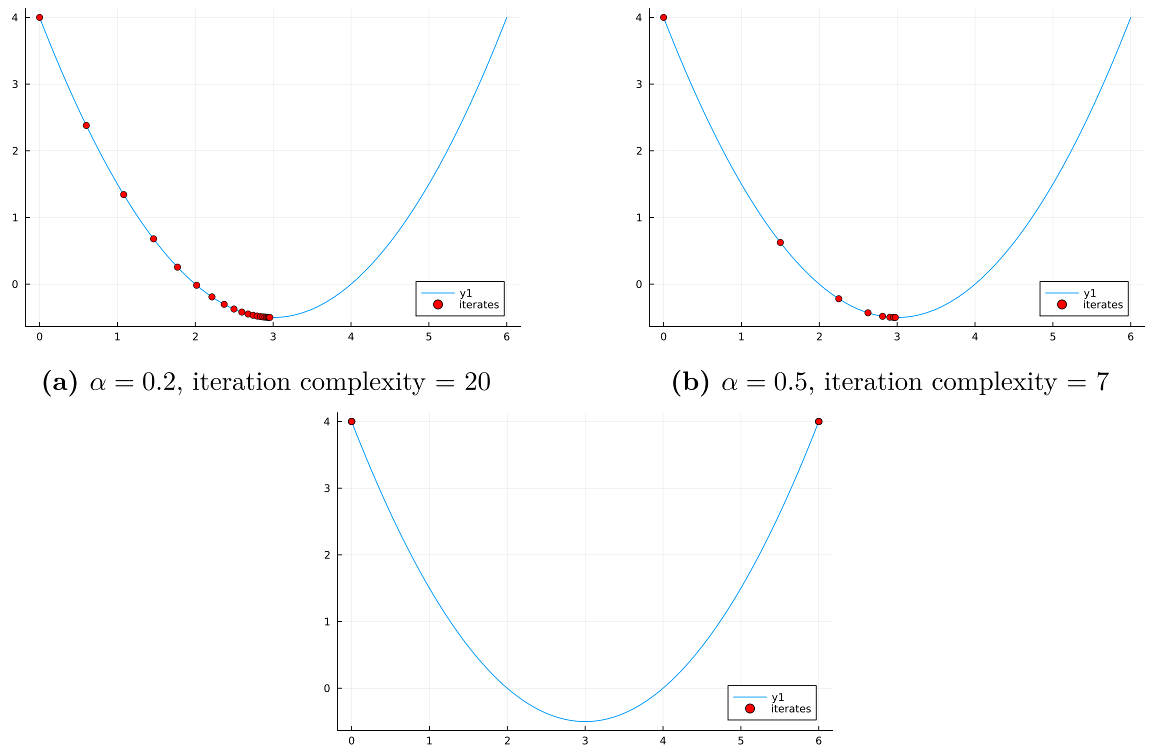
## 1.2 Algorithm analysis

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 (1.2), substituting step size $\alpha$ 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{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 1.2:** Analysis example

- Define the class of function f and its $\nabla f$, 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 $\rho$ 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.

There exist in the literature approaches to analyzing algorithms which guarantee an algorithm's minimum performance. 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 Performance Estimation Problem (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. 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]. This work culminated in a way to perform algorithm analysis to derive the performance bound in the form of a guarantee that after a fixed number of iterates, how close the last iterate is to the goal.

In [9], 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 Algorithm Analysis package implements the algorithm analysis approach presented by Van Scoy and Lessard in [10]. 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 set of smooth strongly convex functions, similar to the PEP method. The method then forms convex SDP using Lyapunov functions, the solution of which establishes whether a convergence rate can be guaranteed for the given algorithm and problem class, making the derivation of the fastest guaranteed rate a simple search problem.

Computer programs have been developed to perform algorithm analysis using the PEP methods, which are PEPit [2], a Python package, and PESTO [11], 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.

The main contribution of this thesis paper is to create a computer program named Algorithm Analysis similar PESTO and PEPit that aims to provide an accessible and fast way to analyze the performance of first-order methods for a guaranteed convergence rate. 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.

# 3

# Lyapunov-based approach

Algorithm Analysis performs algorithm analysis by using the technique layed out by Van Scoy and Lessard in [10]. 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 (1.2), (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

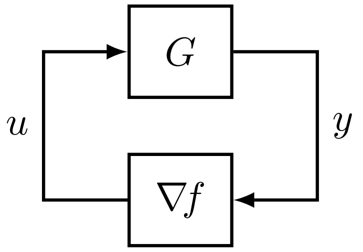


**Figure 3.1:** Block diagram representation of iterative algorithms

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 (1.2), (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-

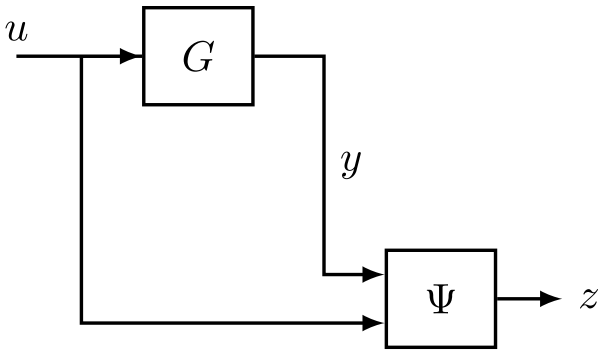


**Figure 3.2:** Block diagram of iterative algorithms with gradient replaced by interpolation condition filter

ical system $\Psi$. 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 [10], 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 formulated in [5] and reformatted in [10] as:

**Theorem 1** ( [5], Thm. 4; [10], 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:

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

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:

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

## 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 $\rho$ 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 (1.2), define $\mathbf{x_k} = x_k - x_s$, the Lyapunov function takes the form:

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

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 $\rho$ 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 $\rho^2$ after each iteration. Together, if the optimization variable $P$ can be found so that the two inequalities are satisfied, a convergence rate of $\rho$ or faster can be guaranteed. This is proven in the proof of Lemma 5 of [10], 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:

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

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

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

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 $\Lambda$ before being combined with $L1$ and $L2$. For each constraint created by the interpolation conditions, an optimization variable $\Lambda$ 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 (5.4) and $M_i$ is the linear form of each constraint scaled by a corresponding parameter $\Lambda_i$. The optimization parameters $\Lambda_i$ 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 $\Lambda_i$ enable the solver to search for $\Lambda_i$ 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 $\rho$, if some parameter $P$, $\Lambda_i^1$ and $\Lambda_i^2$ 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 $\geq 0$, it must mean that $L1 \begin{bmatrix} x & u \end{bmatrix}^\mathsf{T}$ is $\leq 0$:

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

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

(3.15c)

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

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

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 $\rho$ 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 = Rⁿ()
        y0 = Rⁿ()^2
end
@show(x0)
x0 = x0

Vector in Rⁿ()
Label: x0
Associations: Dual => x0*

@show(y0)
y0 = y0

Scalar in Rⁿ()
Label: y0
Oracles: LinearFunctional{Rⁿ}
```

**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{Rⁿ}
```

**Figure 4.2:** Example of a decomposition 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 decompositions 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 4.3:** 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. Tbe Algorithm Analysis package performs the multiplication or division of an expression by scaling the value or decomposition of an 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 4.4:** Example of multiplication operation

## Transpose of a vector

In the Algorithm Analysis program, the transpose of a vector expression can be taken to create a new expression. When a vector expression is created, a data structure that maps it to its transpose is stored in the `associations` field, allowing the program to keep track of whether an expression is the tranpose of another.

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

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

@show(x0'')
(x0')' = x0

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

**Figure 4.5:** 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 $x0^{\mathsf{T}} * y0$. The Algorithm Analysis package finds the inner product by creating a new vector variable expression with the appropriate label indicating the new expression as an inner product between two other vectors.

```
inner = x0'*y0
@show(inner)
inner = <y0,x0>

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

**Figure 4.6:** 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 4.7:** Example of norm of vector expression

## 4.3 Oracles

As mentioned in section 3.2, algorithm analysis relies on interpolation conditions — constraints on the input $y$ and output $u$ of block $\nabla f$ . Therefore this block can be treated as a blackbox, represented in this package by 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. In Figure 1.2, by calling the SectorBounded function, an oracle containing the interpolation conditions for 1 strong 10 smooth convex functions is created and labeled f'. The oracle is then sampled at points xs and x0 by defining f'(xs) and f'(x0) inside the labeling macro to create expressions ∇f(x0) and ∇f(xs)

```
@show(f'(x0))
(f')(x0) = ∇f(x0)

Vector in Rⁿ
  Label: ∇f(x0)
  Oracles: ∇f
  Associations: Dual => ∇f(x0)*
```

**Figure 4.8:** Example of an expression created by sampling an oracle

As an oracle is sampled, Algorithm Analysis uses the oracle's set of interpolation conditions to create constraints on the expression the oracle is being sampled at and the result expression, which are x0 and xs in Figure 1.2.

```
constraints(x0^2)
Set of constraints with 3 elements:
        0 <= 1.1 <∇f(x0),x0> + 2.0 <xs,x0> - 2.0 |x0|² + 1.1 <
           x0,∇f(x0)> - 2.0 |xs|² - 1.1 <xs,∇f(x0)> + 2.0 <x0,
           xs> - 0.2 |∇f(x0)|² - 1.1 <∇f(x0),xs>
        0 <= R[|x0|² <xs,x0>; <x0,xs> |xs|²]
        0 <= R[|x0|² <∇f(x0),x0> <xs,x0>; <x0,∇f(x0)> |∇f(x0)|
           ² <xs,∇f(x0)>; <x0,xs> <∇f(x0),xs> |xs|²]
```

**Figure 4.9:** Example of constraints created by sampling an oracle

Figure 4.9 is created by creating an oracle in the `SectorBounded` function class. The first constraint is the interpolation condition for the gradient to be sector bounded, given the interpolated points $x_0$ and $x_s$:

$$0 \leq 1.1\langle \nabla f(x_0), x_0\rangle + 2\langle x_s, x_0\rangle - 2\|x_0\|^2 + 1.1\langle x_0, \nabla f(x_0)\rangle - 2\|x_s\|^2 \\ - 1.1\langle x_s, \nabla f(x_0)\rangle + 2\langle x_0, x_s\rangle - 0.2\|\nabla f(x_0)\|^2 - 1.1\langle \nabla f(x_0), x_s\rangle,$$

while the third constraint applied on the Gram matrix associated with the vectors $x_0$, $\nabla f(x_0)$, and $x_s$, constraining it to be positive semidefinite:

$$0 \preceq \begin{bmatrix} \|x_0\|^2 & \langle \nabla f(x_0), x_0\rangle & \langle x_s, x_0\rangle \\ \langle x_0, \nabla f(x_0)\rangle & \|\nabla f(x_0)\|^2 & \langle x_s, \nabla f(x_0)\rangle \\ \langle x_0, x_s\rangle & \langle \nabla f(x_0), x_s\rangle & \|x_s\|^2 \end{bmatrix}.$$

The second constraint, while does not effect the mathematical derivation of a convergence rate guarantee for this class of function, is redundant and represent a part of the code that will need to be fixed before the Algorithm Analysis package can be completed.

## Transpose oracle

The transpose of a vector is coded in Julia to be an oracle, which is created when a vector expression is defined and is stored inside a wrapper data structure. During the inner product operation or norm operation, an oracle representing the transpose of a vector is sampled at another expression to create a scalar inner product expression. In this case, the oracle is not based on any class of function and create the constraint on the Gram matrix detailed in `Section 3.2`.

## 4.4 States

Algorithm Analysis represents the states of an algorithm using expressions. As the user inputs the algorithm being analyzed, an initial state is created and an updated state is defined as some algebraic combination of the initial state and the gradient. The relationship between a state and its next state can be defined using the `=>` operation inside the labeling macro, as can be seen in Figure 1.2, and the next state is stored in the `next` field of a state expression.

## 4.5 Label macro

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 macro, it is made more accessible both in terms of entering input and interpreting results.

When an expression is defined inside the labeling macro, the expression object created is given the label based on the variable name used. While x0 and y0 in Figure 1.2 were labeled with its variable name, an expression defined outside of the labeling macro, as shown in Figure 4.10 would not. In some special cases, expressions are automatically given labels that follow common math notation. For an expression created from sampling an oracle representing the gradient of a function $f$, Figure 4.11 shows the macro would assign the expression the label $\nabla f(x0)$, a math symbol typically recognized as the gradient. Similarly, as shown in Figure 4.5, the transpose of a vector such as `x'` is automatically labeled as `x*`), while Figure 4.9 shows that inner products such as `x'*y` gets labeled as $\langle$`x,y`$\rangle$.

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

**Figure 4.10:** Example of an unlabeled expression

## 4.6 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.

```
@show(f'(x0))
(f')(x0) = ∇f(x0)

Vector in R^n
Label: ∇f(x0)
Oracles: ∇f
Associations: Dual => ∇f(x0)*
```

**Figure 4.11:** Example of a labeled oracle

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:

**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 $\geq 0$ or $\leq 0$ 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 4.9 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
```

**Figure 4.12:** Example of user added constraint

## 4.7 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.

## 4.8 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 [12], 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 [13] 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 4.13 shows how JuMP can optimize the problem. However, as discussed in Section 3.3, Algorithm Analysis find the

```
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 4.13:** Example of JuMP minimizing an optimization problem

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 4.13 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 4.14.

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

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

**Figure 4.14:** Example of JuMP certifying the feasibility an optimization problem

# 5

# Analysis Process and Result

The Lyapunov function approach require 2 components which forms the semidefinite problem described in Chapter 3 to produce a worst-case performance convergence rate: the state update matrices and the constraints formed by the interpolation conditions. These components are derived from the input provided to the program which undergo transformations before they can be used to create an optimization problem in the JuMP modelling language, the process of which can be described in 3 steps:

1. The Algorithm Analysis program automatically uses the input provided to form a systematic charaterization the analysis problem using data structures described in Chapter 4, including how the algorithm being analyzed updates, the constraints created by the interpolation conditions of the class of function and the performance measure.
2. These data structures are converted to real number vectors and matrices that represent the updated state of the algorithm and each algorithm in the form of the a linear function of the initial states and inputs.
3. An optimization problem is created inside a JuMP model using these representations and solved to verify whether a certain convergence rate is feasible for a given problem. This process is repeated with different convergence rates as the program search for the lowest feasible convergence rate.

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.

## Real scalars and linear form

As specified in 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 $[x; u]$. 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>
    <xs,x0>
```

**Figure 5.1:** Initial state real scalar expressions from example 1.2

2. Define an update state vector x$^+$ consisting of the next expression of every expression in the initial state. The input vector u is then defined as every real expression that exist in the decomposition of the updated state expressions but not in the initial state expressions.

```
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

u  = collect(setdiff(variables(x⁺), variables(x)))
5-element Vector{Expression}:
    <xs,∇f(x0)>
    <x0,∇f(x0)>
    <∇f(x0),xs>
    <∇f(x0),x0>
    |∇f(x0)|²
```

**Figure 5.2:** Updated state and input real scalar expressions from example 1.2

3. The initial state and input vector `[x; u]` is the code equivalent of $\begin{bmatrix} x & u \end{bmatrix}^\mathsf{T}$ and can form every expressions required to form the linear matrix inequality. This transformation, which will be refered to as the linear form of an expression, can be derived by finding the values of each expression in the vector `[x; u]` present in the expression's decomposition dictionary.

```
linear_form = vec(linearform([x; u] => x0^2 - 3*(x0'*xs)))
linear_form'*[x; u]
Scalar in R
  Decomposition: -3 <xs,x0> + |x0|²
```

**Figure 5.3:** Example of linear form of a scalar expression 1.2

## 5.1 Performance measure

The linear form matrix of the performance measure is the first of the three components needed to form the Lyapunov function ($\|x_k - x_s\|^2$ in (3.7)). For example, the performance measure in 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 Figure 5.4.

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

**Figure 5.4:** Linear form matrix of expression $(x0 - xs)^2$

## 5.2 Algorithm, state update and Lyapunov function formulation

As shown in Figure 1.2 and in section 4.4, the algorithm to be analyzed is provided as input first by defining an initial state and how the next state is updated from the initial state. The initial state is defined to be a vector in an inner product space, and the updated state is a linear function of one or multiple initial state and the gradient of the function evaluated at some point. While the gradient descent algorithm updates using only one state and evaluate the gradient at the previous state, if an algorithm updates using multiple past states or the gradient at an interpolated point, these vectors will also have to be defined. The forming of the algorithm can then be completed by defining the relationship between states and their next states using the `=>` operation, which updates the next field of every expression in the decomposition of which there is the state on the left hand side of the operation. For example, in Figure 1.2, the state vector `x1` is defined as a function of the state vector `x0` and as the next state of `x0`, while the next state of the stationary point `xs` is itself. This not only means the `next` field of `x0` and `xs` are `x1` and `xs` respectively, but also next state of any every expression derived from the norm, inner product, or algebraic calculation of which `x0` is a part is that calculation done with `x1` instead. In this example, as shown in Figure 5.5, the next state of the inner product of $x_0$ and $x_s$ denoted as `next(x0'*xs)` is the inner product of $x_1$ and $x_s$ `x1'*xs`. This enables the operation in Figure 5.7 and allow any updated iterate to be automatically expressed a linear function of the intial states and inputs.

```
next(x0)

Vector in R^n
  Label: x1
  Decomposition: -0.18181818181818182 ∇f(x0) + x0
  Associations: Dual => x1*

next(x0'*xs)

  Scalar in R
    Decomposition: -0.18181818181818182 <xs,∇f(x0)> + <xs,x0>
```

**Figure 5.5:** `next` field of state a vector expression and a scalar formed from a state expression

As the initial state vector is defined in Figure 5.1 and the updated state vectors in Figure 5.2, their linear form matrices is the second of the three components needed to formulate the Lyapunov function and can be formed as shown in Figure 5.6 and Figure 5.7.

```
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 5.6:** Linear form state matrix x

```
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 5.7:** Linear form state matrix $x^+$

Following the steps presented in chapter 3, the Lyapunov function can begin to 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 Lyapunov functions are then created following (3.7) but with code variables as:

$$L1 = \mathcal{P} - X^{\intercal} P \tag{5.1a}$$

$$L2 = X^{+\intercal} P - \rho X^{\intercal} P \tag{5.1b}$$

## 5.3 Constraints

As presented in 4.3, the oracle created from the class of function and the transpose of each expression automatically forms the interpolation condition and Gram matrix constraints. These constraints are linearized and added to the optimization in 2 steps:

**Optimization variable multipliers** For each constraint $i \in$, two optimization variables $\lambda_i$ and $\mu_i$, which represent $\Lambda_i^1$ and $\Lambda_i^2$ 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:

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

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

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 (5.2), the linear form of the constraint in terms of $\begin{bmatrix} x \\ u \end{bmatrix}$, denoted as $M$ would be:

$$\begin{bmatrix} \lambda & \lambda \end{bmatrix} * \begin{bmatrix} (x0 - xs)^2 \\ (x0 - xs)^2 - 3 * |xs|^2 \end{bmatrix} = M * \begin{bmatrix} |x0|^2 \\ < xs, x0 > \\ |xs|^2 \end{bmatrix} \tag{5.3a}$$

$$M = \begin{bmatrix} \lambda & -\lambda & \lambda \\ \lambda & -\lambda & -2\lambda \end{bmatrix} \tag{5.3b}$$

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 (5.4) which is constrained to be positive semidefinite, its linear form would be:

$$tr(\lambda \begin{bmatrix} ||x0||^2 & \langle xs, x0\rangle & \langle \nabla f(x0), x0\rangle \\ \langle x0, xs\rangle & ||xs||^2 & \langle \nabla f(x0), xs\rangle \\ \langle x0, \nabla f(x0)\rangle & \langle xs, \nabla f(x0)\rangle & ||\nabla f(x0)||^2] \end{bmatrix}) \geq 0 \tag{5.4}$$

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

## 5.4 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 $P$, $\lambda$s and $\mu$s for which the linear matrix inequality is satisfied and a convergence rate $\rho$ can be guaranteed.

Using the definition of the convergence rate in (3.5), a $\rho$ 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 single iterate. In order to find the worst-case performance rate, the program performs bisection search, also known as binary search, for the smallest value $\rho$ between 0 and 1 that makes the optimization problem feasible, calling a function to perform the steps presented in this chapter for each value $\rho$ and checking feasibility at each iterate of the search. The smallest value $\rho$ 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 Result

In the case of the gradient descent algorithm, the interpolation conditons for $m$-$L$ sector bounded functions and $m$-strong $L$-smooth convex functions are the same. Therefore, to see if the `AlgorithmAnalysis.jl` package is able to analyze the gradient descent algorithm over the $m$-$L$ sector bounded function class, we can plot the rate bound of this case and compare it with the analysis results of the same algorithm and step size but for $m$-strong $L$-smooth convex functions presented in Figure 2 of [10]. In order to produce comparable results, we run the analysis 19 $m$-$L$ value pairs where the condition number $L/m$ varies between 1 and 100. The code used to produce this analysis result is presented in Figure 6.1. The convergence rate guarantee found for each condition number is shown in Figure 6.2.

## 6.2 Future work

Following this thesis proposal, the future work include finish coding the program. While most to all of the package have been coded, the program is only able to produce result correctly matching the analysis result produced in [10] for the gradient descent algorithm. As a result, the program need to be debugged and tested to ensure it is able to correctly implement the mathematical approach to algorithm analysis and produce accurate results.

Additionally, the program currently does not support ”lifting dimension”, an optional step of the Lyapunov method to algorithm analysis in [10]. Part of the future work of this research work is to implement this step and measure its effect on the tightness of the convergence rate guarantee.

A crucial step before the package can be completed and delivered as end-user software

```
m, condnum, = 1, [range(1,10,10); range(20,100,9)]
# condition numbers tested
GDresults = []
for L in condnum
    # Stop if the algorithm doesn't converge
    if length(GD) > 0 && last(GD) == 1
        append!(GD, 1)
    else
        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
        append!(GD, rate(performance))
    end
end
plot(condnum, [GD], title="", label="GD", xaxis=:log)
xlabel!("Condition number L/m")
ylabel!("Convergence rate bound")
```

**Figure 6.1:** Analysis results for GD and $m$-$L$ sector bounded functions over 19 $L/m$ condition numbers

and part of the future work of this thesis is the documentation of Algorithm Analysis, which includes writing operating instructions, providinig analysis examples, detailing the package's components, and the steps the program takes to perform algorithm analysis.

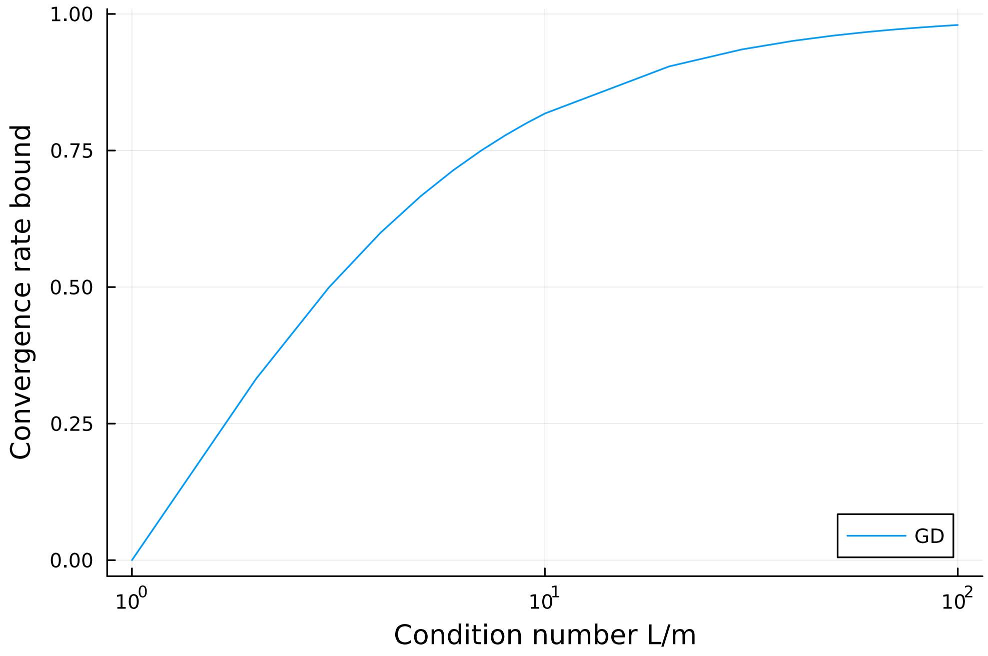


**Figure 6.2:** Convergence rate guarantee of gradient descent at optimizing $1 - 10$ sector bounded functions

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

[10] 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.

[11] 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.

[12] 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.

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