JUPE: 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

JUPE: 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

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

by Lam Ngoc Ha

JuPE is a computer program written in the Julia language that seeks to provide an accessible way of finding the worst-case performance guarantee of any gradient-based optmization algorithm in a process called *algorithm analysis*. As first-order optimization methods are used to optimize large-scale problems in fields such as machine learning or data science, savings in computational capacity can be made if algorithms can be compared on their performance. JuPE by implementing the Lyapunov-based approach to algorithm analysis and creating a domain-specific compiler simplifies and automate the process of analyzing algorithms.

Instead of starting with the solution (JuPE), start with the goal and the problem: optimization problems appear throughout science and engineering, analyzing their performance is difficult, so you design a program to make the analysis accessible. Also, the last sentence is a little awkward (you at least need comma's after "JuPE" and "compiler", but it would be better to rearrange it).

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 JuPE

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 - (this should be an em-dash, which is typed in latex as —) 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 means they are extensively used in fields such as machine learning and data science. 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 opens the potential of comparing algorithms to find or derive the best performing one, improving efficiency, (this is known as the Oxford comma) and saving time and computational resources. As a result, substantial research have (has) been conducted to quantify the performance of algorithms either through emperical evidence or mathematical proof, the latter of which is the method this paper (refer to this as a thesis, not paper) uses.

Algorithm analysis is a field which seeks to mathematically prove the worst-case performance of an algorithm at solving a function class (haven't talked about an objective function yet, so this is a bit vague. I would instead say a set of optimization problems). But as different methods of analyzing algorithms are developed, anyone seeking to apply have had to understand the underlying math until recently [1]. The main work of this thesis presents a tool that automatically analyze gradient-based algorithms' performance characteristics accessible

to both experts and non-experts: JuPE (Julia Performance Estimation) 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, the algorithm to be analyzed and the performance measure, it returns a guaranteed rate at which the algorithm can solve any function in the set.

1.1 Optimization problems and algorithms

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

$$\text{minimize} \quad f(x) \tag{1.1a}$$

$$\text{subject to} \quad x \in X \tag{1.1b}$$

Where (do not capitalize unless starting a new sentence) $f(x)$ is the optimization function and X is a constraint set. Here, x is the input (or decision variable) and $f(x)$ is a measure of how close a solution is to being optimal. Well-known examples of this problem are large language models (LLMs) such as ChatGPT and machine learning models that enable self-driving features in automotives. These models are created and continuously improves in a training process, an integral part of which is the minimizing of loss functions, a process where 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 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 ~~be~~ optimize 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 the last iteration $\nabla f(x_k)$ until it reaches a local minimum x_*. For example, the gradient descent (GD) algorithm updates x_k following this formula:

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

(again, do not capitalize here unless starting a new sentence) Where α is the step size, an adjustable parameter of the algorithm. The stepsize (avoid starting sentences with symbols) 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_*. Accelerated algorithms exist that seek to solve the problem of overshooting, 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 which are used in this paper to introduce the concept of algorithm analysis, the Lyapunov-based method and how it is coded into and performed by JuPE. In the last chapter, the analysis result of these algorithms done by JuPE will be presented.

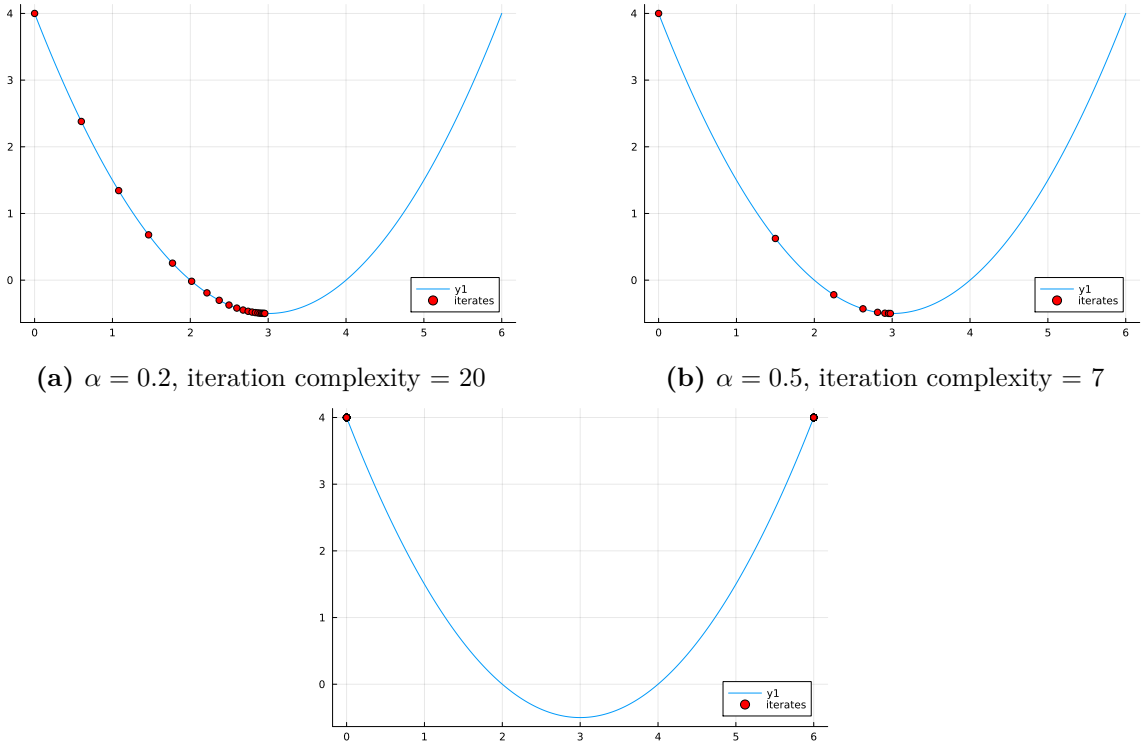
1.2 Algorithm analysis

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 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 the example is of a simple function where overshoot can easily be identified and the number of iteration needed is small, the benefit of using an optimal algorithm only increases as the problem gets larger and more complex. In the application



(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

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, 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.

Considering the quadratic function example, while it 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 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.

Due to these limitations, it is more efficient to analyze algorithms' performance at solving a broader set of problem. As a result of their widespread application, popular iterative gradient-based algorithms have been extensively analyzed. A frequently cited attempt is the Adam algorithm [2], 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 [3], [4], and [5], 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 that a characteristic that a set of functions might share (such as being convex or quadratic), algorithm analysis would return the worst-case performance measure that guarantees the algorithm analyzed would perform as good or better solving every function within said set.

1.3 Julia programming language

JuPE 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 [6]. Julia accomplishes this while being a high-level language with simple syntax rules that resembles existing popular languages, making it easy to code with and to understand.

Julia was 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.

Julia was also chosen as it is open-source and available for free. As JuPE is a package designed for expert and novice users alike to install and use, it made sense to choose Julia as it available on many of the popular platforms such as macOS, Windows, and Linux.

1.4 Overview

JuPE performs worst-case algorithm analysis when three main inputs are provided: The class of functions in question, the algorithm being analyzed, and a performance measure. The package then performs the algorithm analysis and returns the fastest guaranteed convergence rate.

To use JuPE, an iterative first-order algorithm need to be defined and inputted to the program by specifying how it is updated. The class of function is provided by detailing the characteristic of the set, and a performance measure is defined. Throughout the process, the

user never has to change the code of the package or understand how JuPE works, making it an easy to use black box tool.

In the next chapters, we will 1) detail the existing literature of approaches to analyzing algorithms and implementation into a program, 2) discuss the Lyapunov-based mathematical method that JuPE utilizes, 3) shows how a dommain-specific compiler is created to enable Julia to work, and 4) demonstrate the analysis process and include the analysis result that the package has done.

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 [2], 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.

On the other hand, there exist in the literature approaches to performing algorithm analysis. 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) whose size is proportionate with the number of iterations the algorithm is run. [3]. The paper coined the term Performance Estimation Problem (PEP) and showed that by solving a convex semidefinite problem (SDP), 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. This work culminated in a way to perform algorithm analysis to derive the performance bound in the form of a guarantee that after a finxed number of iterates, how close the last iterate is to the goal.

In [7], 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 shows how a complex system can be described using certain IQCs and presents a stability

theorem for systems described by IQCs.

Computer programs have been developed to perform algorithm analysis using the PEP methods, which are PESTO [8], a MATLAB toolbox, and PEPit [1], a Python package. The program is able to perform algorithm analysis and generate a worst-case performance guarantee for algorithms and function classes from a supported list. PESTO and PEPit follows the PEP methodology and first presented an automatic way to analyze gradient-based algorithms.

JuPE implements the approach presented in [9], which uses IQCs to represent the gradient of the algorithm and transforms the problem of deriving a performance guarantee to a convex SDP similar to the PEP approach. However, this approach instead uses Lyapunov functions to create a small and fixed size SDP, unlike that created by the PEP approach which grows in size for each iteration an algorithm is run, and proves that the performance measure inputted by the users decreases at a guaranteed rate throught out the optimizing process.

The main contribution of this thesis paper is to create a computer program named JuPE 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 compiler inside the Julia programming language, JuPE 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 accessible the analysis of optimization algorithms.

3

Lyapunov-based approach

JuPE performs algorithm analysis by using the technique layed out by Van Scoy and Lessard in [9]. While JuPE is a blackbox tool, understanding the mathematical approach on which it is based is prequisite 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, 2) replaces the nonlinear gradient with constraints derived from interpolation conditions of a function class, 3) form an optimization problem using Lyapunov functions and constraints and solve it to 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: As iterative gradient-based algorithms uses the gradient of the function to update x, they can be reformulated into a linear time-invariant (LTI) system (how the algorithm update) in feedback with a static nonlinearity (the gradient of f) at point x. Using a block diagram, the algorithm can be seen as:

Here, G represents the LTI system, while y and u are input and output of the gradient nonlinearity. For example, (FG) equation 1.4 can be rewritten to match this view as:

$$x_{k+1} = x_k - \alpha u_k, \tag{3.1a}$$

$$y_{k+1} = x_{k+1} + \beta(x_{k+1} - x_k),, \tag{3.1b}$$

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

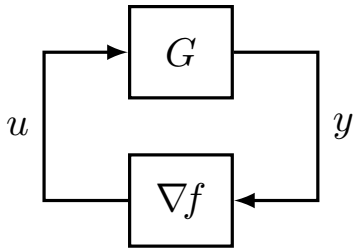


Figure 3.1: Block diagram representation of iterative algorithms

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

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

$$y_k = \begin{bmatrix} 1+\beta & \beta \end{bmatrix} \xi_k, , \tag{3.2b}$$

$$u_k = \nabla f(y_k) \tag{3.2c}$$

Due to the fact that function f is n-multivariate, each iterate x_k is n-dimensional $x_k \in \mathbb{R}^n$ and is a 1 by n vector. ξ in this case therefore is a 2 by n matrix.

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 1.1, these matrices are:

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 SDP.

3.2 Interpolation condition

Function class interpolation conditions

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 linearity's input and output y and u. Through this characterization, the block diagram representation is transformed into: Here, the nonlinear

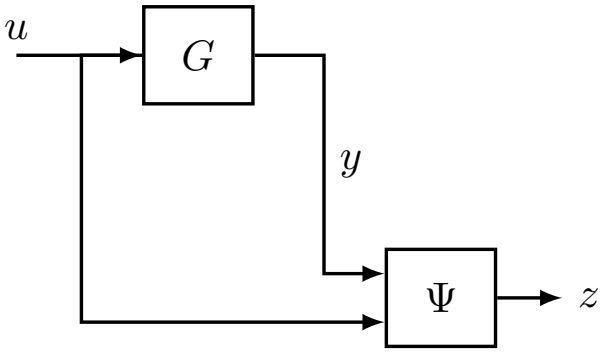


Figure 3.2: Block diagram of iterative algorithms with gradient replaced by interpolation condition blackbox

gradient block is replaced with blackbox Ψ and characterized by constraints on the block's input and output y and u. 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 and can be derived from the function class. The interpolation condition of m strong L smooth convex functions was first formulated in [4] and reformatted in [9] as:

Theorem 1 ([4], Thm. 4; [9], 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:

$$2(L-m)(f_i - f_j) - mL||y_i - y_j||^2 + 2(y_i - y_j)^T(mu_i - Lu_j) - ||u_i - u_j||^2 \geq 0 \text{ for } i, j \in I$$

As an example, if we define x_0 as the initial iterate of an algorithm optimization a 1 strong 10 smooth convex function ($f \in F_{1,10}$), $\nabla f(x_0)$ as the gradient of x_0 at f, x_s as the minimizer where $\nabla f(x_s) = 0$, we can define set I of 1 as including s and 0 and get the inequality:

$$-20||x_0||^2 - 20||x_s||^2 + 40\langle x_0, x_s\rangle + 22\langle \nabla f(x_0), x_0\rangle - 22\langle \nabla f(x_0), x_s\rangle - 2||\nabla f(x_0)||^2 \geq 0 \quad (3.3)$$

when satisfied interpolates 1 strong 10 smooth convex functions. Theorem 1 can be rewritten

as:

$$\Lambda \begin{bmatrix} -mL & 2mL & -mL & 2(m+L) & -2(m+L) & -2m \end{bmatrix} \begin{bmatrix} ||y_i||^2 \\ \langle y_i, y_j \rangle \\ ||y_j||^2 \\ \langle \nabla f(y_i), y_j \rangle \\ \langle \nabla f(y_j), y_i \rangle \\ ||\nabla f(y_j)||^2 \end{bmatrix} \geq 0 \tag{3.4}$$

For all Λ such that $\Lambda_{i,j} \geq 0$. Using this equation, 3.3 can be scaled by 10 and rewritten as:

$$\Lambda \begin{bmatrix} -2 & 4 & -2 & 2.2 & -2.2 & -0.2 \end{bmatrix} \begin{bmatrix} ||x_0||^2 \\ \langle x_0, x_s \rangle \\ ||x_s||^2 \\ \langle \nabla f(x_0), x_0 \rangle \\ \langle \nabla f(x_0), x_s \rangle \\ ||\nabla f(x_0)||^2 \end{bmatrix} \geq 0 \tag{3.5}$$

The left hand side of each constraint, such as 3.4 is added to the Lyapunov functions.

Transpose interpolation condition

The performance measure and the left hand side of the interpolation conditions are formulated from the norm and inner products of the state vector the end point vector, with the Lyapunov function detailed in the next section formulated using the same elements along with the gradient of the function used by the algorithm. These expressions are elements of the Gram-matrix of the vector containing every expressions the algorithm uses to update the iterates and goal vector expression. For the interpolation condition 3.4 and a gradient descent algorithm which derive the first iterate x_1 using the gradient at the initial state x_0, the Gram matrix is:

$$\begin{bmatrix} ||x0||^2 & \langle xs, x0 \rangle & \langle \nabla f(x0), x0 \rangle \\ \langle x0, xs \rangle & ||xs||^2 & \langle \nabla f(x0), xs \rangle \\ \langle x0, \nabla f(x0) \rangle & \langle xs, \nabla f(x0) \rangle & ||\nabla f(x0)||^2] \end{bmatrix} \tag{3.6}$$

The elements of the Gram matrix are interpolable - meaning there exists vectors $x0$, x_s and $\nabla f(x0)$ with which they exists, if and only if the Gram matrix is positive semidefinite.

This constraint similar to those created from interpolation conditions are later added to the Lyapunov function.

3.3 Lyapunov function certification

In the final step of the Lyapunov method, Lyapunov function is created as a linear function of the state-space representation of the algorithm, an optimization variable P, and the constraints created by the interpolation conditions scaled by optimization variable λ. Theorem 7 proved a convergence rate can be guaranteed if there exist a $\lambda \geq 0$ and P which makes the Lyapunov function nonpositive. Therefore, a minimum convergence rate can be guaranteed by solving for P and λ so that the Lyapunov function would satisfy 2 linear matrix inequalities. The mathematical basis of JuPE follows this approach with some modification.

Define $\mathbf{x_k} = \xi_k - \xi_s$, the Lyapunov function takes the form:

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

Where P is a 2x2 optimization variable. If given an algorithm with state-space representation ξ_k and whose Lyapunov function satisfy the conditions:

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

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

Then the convergence rate of that algorithm is guaranteed to be larger than ρ for every function in the function class. While the Lyapunov function is quadratic in $\mathbf{x_k}$, due to the characteristics of matrix trace, we can see that $V(\mathbf{x})$ is linear in $\mathbf{x}^T\mathbf{x}$, reducing the inequalities to linear matrix inequalities. For gradient descent where the augmented state ξ_k is just the state x_k, the Lyapunov function can be simplified into:

$$\begin{aligned} V(\mathbf{x_k}) &= tr[(x_k - x_s)^T P (x_k - x_s)] \\ &= tr[P(x_k - x_s)(x_k - x_s)^T] \\ &= tr[P(x_k x_k^T - x_s x_k^T - x_k x_s^T + x_s x_s^T)] \\ &= \begin{bmatrix} P & -2P & P \end{bmatrix} \begin{bmatrix} ||x_k||^2 \\ \langle x_k, x_s \rangle \\ ||x_s||^2 \end{bmatrix} \end{aligned}$$

Where $x_k x_s^T = x_s x_k^T = \langle x_k, x_s \rangle$. Similarly, $V(\mathbf{x_{k+1}})$ can be defined as:

$$\begin{aligned} V(\mathbf{x_{k+1}}) &= V(x_k - \alpha \nabla f(x_k)) \\ &= tr[(x_k - \alpha \nabla f(x_k) - x_s)^T P (x_k - \alpha \nabla f(x_k) - x_s] \\ &= tr[P(x_k - \alpha \nabla f(x_k) - x_s)(x_k - \alpha \nabla f(x_k) - x_s)^T] \\ &= \begin{bmatrix} P & -2P & P & 2P\alpha & -2P\alpha & P\alpha^2 \end{bmatrix} \begin{bmatrix} ||x_k||^2 \\ \langle x_k, x_s \rangle \\ ||x_s||^2 \\ \langle \nabla f(x_k), x_k \rangle \\ \langle \nabla f(x_k), x_s \rangle \\ ||\nabla f(x_k)||^2 \end{bmatrix} \end{aligned}$$

The Lyapunov functions inequalities can now be completed by adding the constraints created by function class and transpose interpolation conditions. These constraints similar to the Lyapunov functions are a linear function of an optimization variable and elements of the Gram matrix. The transformation in **??** was done so that each nonnegative constraint is scaled by a nonnegative obtimization variable Λ, and the solver can search for Λ just as it does for P so that the linear matrix inequality is satisfied. For each constraint created by the interpolation conditions, an optimization variable Λ is created and 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. The left hand side of 3.2 and the trace of the Gram matrix scaled by Λ is added to both Lyapunov function. The final linear matrix inequality for gradient descent would be:

$$\begin{bmatrix} 1-P-2\Lambda \\ -2+2P+4\Lambda \\ 1-P-2\Lambda \\ 2.2\Lambda \\ -2.2\Lambda \\ -0.2\Lambda \end{bmatrix}^T \begin{bmatrix} ||x_k||^2 \\ \langle x_k, x_s\rangle \\ ||x_s||^2 \\ \langle \nabla f(x_k), x_k\rangle \\ \langle \nabla f(x_k), x_s\rangle \\ ||\nabla f(x_k)||^2 \end{bmatrix} + \Lambda tr(\begin{bmatrix} ||x0||^2 & \langle xs, x0\rangle & \langle \nabla f(x0), x0\rangle \\ \langle x0, xs\rangle & ||xs||^2 & \langle \nabla f(x0), xs\rangle \\ \langle x0, \nabla f(x0)\rangle & \langle xs, \nabla f(x0)\rangle & ||\nabla f(x0)||^2] \end{bmatrix}) \leq 0$$

$$\begin{bmatrix} P-P\rho^2-2\Lambda \\ -2P+2P\rho^2+4\Lambda \\ P-P\rho^2-2\Lambda \\ 2P\alpha+2.2\Lambda \\ -2P\alpha-2.2\Lambda \\ P\alpha^2-0.2\Lambda \end{bmatrix}^T \begin{bmatrix} ||x_k||^2 \\ \langle x_k, x_s\rangle \\ ||x_s||^2 \\ \langle \nabla f(x_k), x_k\rangle \\ \langle \nabla f(x_k), x_s\rangle \\ ||\nabla f(x_k)||^2 \end{bmatrix} + \Lambda tr(\begin{bmatrix} ||x0||^2 & \langle xs, x0\rangle & \langle \nabla f(x0), x0\rangle \\ \langle x0, xs\rangle & ||xs||^2 & \langle \nabla f(x0), xs\rangle \\ \langle x0, \nabla f(x0)\rangle & \langle xs, \nabla f(x0)\rangle & ||\nabla f(x0)||^2] \end{bmatrix}) \leq 0$$

By using Lyapunov functions, we have reduced the optimization problem to a pair of linear matrix inequalities in the variables P and Λ. The optimization problem can now be solved for any convergence rate ρ whether there exist P and Λ so that the inequality are satisfied, proving whether ρ can be guaranteed. The smallest convergence rate guarantee can be found by performing bisection search for the smallest value ρ between 0 and 1 with which the optimization problem is feasible.

4

Code Components

To in order to perform algorithm analysis with JuPE, the user need 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.

```
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 = (x1-xs)^2
end

@show rate(performance)
```

Figure 4.1: Analysis Example

In the example code, the user:

- Define the class of function f and its gradient f' by calling one of the provided functions. In this example, the class of function is 1 - 10 sector bounded 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 α= 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 is decreasing after each iteration of the algorithm.
- Call the rate function perform automated algorithm analysis.

With the calling of the rate function, JuPE derives every necessary input from the performance measure and performs analysis to return a rate of 0.6687164306640625. This means for the provided algorithm and every function in the class, the convergence rate of the performance measure is guaranteed to match or exceed the result worst-case guarantee rate.

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

Figure 4.2: Analysis result

JuPE performs algorithm analysis by following the set of instructions presented in section 3 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, or formulating and solving a convex optimization problem, while keeping the users' interaction with the program simple. This chapter goes into the code that constitutes JuPE and enables these functionalities.

4.1 Expressions

Expressions are data structures used to represent mathematic variables and enable the implementation of the steps 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.

```
@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 4.3: Example of a variable expression representing a vector

Decomposition expression represents scalars and is some linear combination of scalar variable or decomposition expressions. Its decomposition is a dictionary containing how many of each expression that form the decomposition expression.

```
decomposition_exp = x0 + 2xs
@show decomposition_exp
decomposition_exp = x0 + 2 xs

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

Figure 4.4: Example of a decomposition expression

4.2 Algebra

Expressions in JuPE 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 example 4.3, expressions are defined to be in R^n, an inner product space pre-defined in JuPE which encompasses n-dimensional real numbers.

JuPE supports operations that characterize inner product spaces between expressions. The examples that demonstrate these operations use vector expressions x0 and y0 in 4.3

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 4.5: 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. JuPE perform the multiplication or division of an expression by scaling the value or decomposition of an expression.

Transpose of a vector

In JuPE, 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 JuPE to keep track of whether an expression is the tranpose of another.

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

Vector in R^n
        Label: c
        Decomposition: 2 y0 - 3 x0
        Associations: Dual => c*
```

Figure 4.6: Example of multiplication operation

```
@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 4.7: 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 <y0, x0> can be found as $y0^T * x0$. JuPE 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.

Squared norm

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

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

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

Figure 4.8: Example of an inner product between two vector expressions

```
norm = x0^2

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

Figure 4.9: Example of norm of vector expression

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 JuPE by oracles, data strucutres 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 4.1, 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)

As an oracle is sampled, JuPE 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 4.1:

```
@show(f'(x0))
(f')(x0) = ∇f(x0)

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

Figure 4.10: Example of an expression created by sampling an oracle

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

Transpose oracle

The transpose of a vector is coded in Julia to be an oracle, whic 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 an 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 seciton 3.2.

4.4 States

JuPE 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 4.1, and the next state is stored in the "next" field of a state expression.

4.5 Label macro

JuPE uses a macro to keep the process of providing inputs to the program simple as some of the rules of programming might be difficult to navigate. While JuPE stills works without the label macrom, it makes JuPE as accessible as possible both in terms of entering input and interpreting results with these functionalities:

Define During the inputing process, users can define a new expression or oracle with only one line specifying its trait, instead of the usual steps of declaring a new object and filling in its fields that typically exist in programming. The macro will calls the necessary functions to create the object, assign every relevant field as well as updating every object associated with the one being created.

Describe When an expression is referenced, JuPE will describe the expression and any relevant field without the user having to access it.

Label 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 4.1 were labeled with its variable name, an expression defined outside of the labeling macro would not: x3 = $R^n()$

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

Figure 4.12: Example of an unlabeled expression

In the special case of an expression created from sampling an oracle representing the gradient of a function f , the macro would assign the expression the label $\nabla f(x0)$ following common math notation.

```
@show(f'(x0))
(f')(x0) = ∇f(x0)

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

Figure 4.13: Example of a labeled oracle

4.6 Constraints

During the analysis process, constraints are created by interpolation conditions in section 3.2, as well as when the user wish to define constraints on the problem being optimized by iterative algorithm being analyzed. In order to keep track of these constraints while keeping the process of defining them simple, JuPE 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 JuPE supports include:

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

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

Positive or negative semidefinite 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 4.11 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 begin
        (x0-xs)^2 <= 1
end
```

Figure 4.14: Example of user added constraint

4.7 Performance measure

Part of JuPE's required inputs is the performance measure, the convergence rate of which JuPE finds the worst-case guarantee through algorithm analysis. In 4.1, 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 in itself an optimization as defined in section 1.1, with 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, JuPE uses JuMP, 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 JuPE uses the SCS [10] 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.

5

Analysis Process and Result

The Lyapunov function approach require 3 components which forms the optimzation problem described in chapter 3 to produce a worst-case performance convergence rate: the state update matrices, the constraints formed by the interpolation conditions and the performance measure. These components are derived from the input provided to JuPE 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. JuPE 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 JuPE search for the lowest feasible convergence rate.

This chapter details the analysis process, including how these steps are performed for component and how the optmization problem is formed and solved to derive worst-case performancce convergence rate. In addition, the analysis result of three algorithms over smooth strongly convex functions will also be detailed as an example of the result produced by algorithm analysis.

Real scalars and linear form

All three components needed to form the linear matrix inequality - the Lyapunov functions $V(x_k)$ and $V(x_{k+1})$ and the left hand side of each constraint - from equations 3.4, 3.8 and 3.11 are functions linear in the elements of the Gram matrix and optimization variables. On the other hand, the JuMP modeling language does not support the expression and constraint data structures presented in chapter 4, and the LMIs of **??** must be transformed into a function linear in the optimization variables and real numbers before it can be formed inside the JuMP model. JuPE achieve this by creating a vector of every elements of the Gram matrix and expressing every component that forms the LMIs as a linear function of this vector. 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 4.1

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.

3. The initial state and input vector [x; u] can now form every expression in the optimization problem, which means any real scalar expression can be transformed into a linear function of [x; u] by finding the matrix or vector with which to multiply [x; u] to find that expression. 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 initial and state input vector present in the expression's decomposition dictionary.

```
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 4.1

```
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 4.1

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 ($||\xi_k - \xi_s||^2$ in 3.8). For example, the performance measure in 4.1, which is defined as $(x0 - xs)^2$ and which evaluates into $|x0|^2 - < xs, x0 > - < x0, xs > + |xs|^2$, has the linear form:

```
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 4.1, the algorithm to be analyzed is inputted into JuPE first by defining an initial state and how a '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.

```
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 a state vector expressions and a scalar formed from a state expression

The initial and update state vectors are created in 5.1 and 5.2, their linear form matrices is the second of the three components needed to formulate the Lyapunov function and can be formed as:

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 LMIs are then created as:

```
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

X⁺ = linearform([x; u] => x⁺)
     4x9 Matrix{Real}:
     1  0  0  0   0          0          -0.181818  0           0
     0  1  0  0  -0.181818   0           0         0.0330579
        -0.181818
     0  0  1  0   0         -0.181818   0         0           0
     0  0  0  1   0          0          0         0           0
```

Figure 5.6: Linear form state matrices x and x^+

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

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

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, two optimization variables λ and μ are as JuMP variables. These variables are the λ in 3.4. If the constraint is created from the interpolation condition for the transpose of state vectors and is applied to a matrix of real scalar expressions, the optimization variable will be a matrix sharing the same size with the matrix constrained. 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 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 λ is a 3x3 JuMP variable. In all three cases, the resulting linear form matrix is the linear form of the constraint expression scaled by the JuMP variable multipliers. For each constraints, 2 linear form matrices are created and added to the two Lyapunov functions, completing the final LMIs

5.4 Derived feasibility and bisection search

Following equations 3.7 in section 3.3, the LMIs can now be solved: the solver of the JuMP model is called to optimize the problem and find the varibles P, λ and μ for which the LMIs is satisfied, and a convergence rate ρ can be guaranteed.

Due to the formulation of the Lyapunov functions, a convergence rate of 1 means the algorithm is not converging and a convergence rate of 0 means the algorithm converge after a single iterate. JuPE performs bisection search, also known as binary search, for the smallest value ρ between 0 and 1 that makes the optimization problem feasible, calling performing 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 $1E-5$ is returned as the guaranteed convergence rate, and the analysis process is complete.

5.5 Result

After running JuPE to perform algorithm analysis on 3 algorithms (GD), (HB), and (FG) on classes of m strong L smooth convex function where the condition number L/m are chosen between 1 and 10, we get

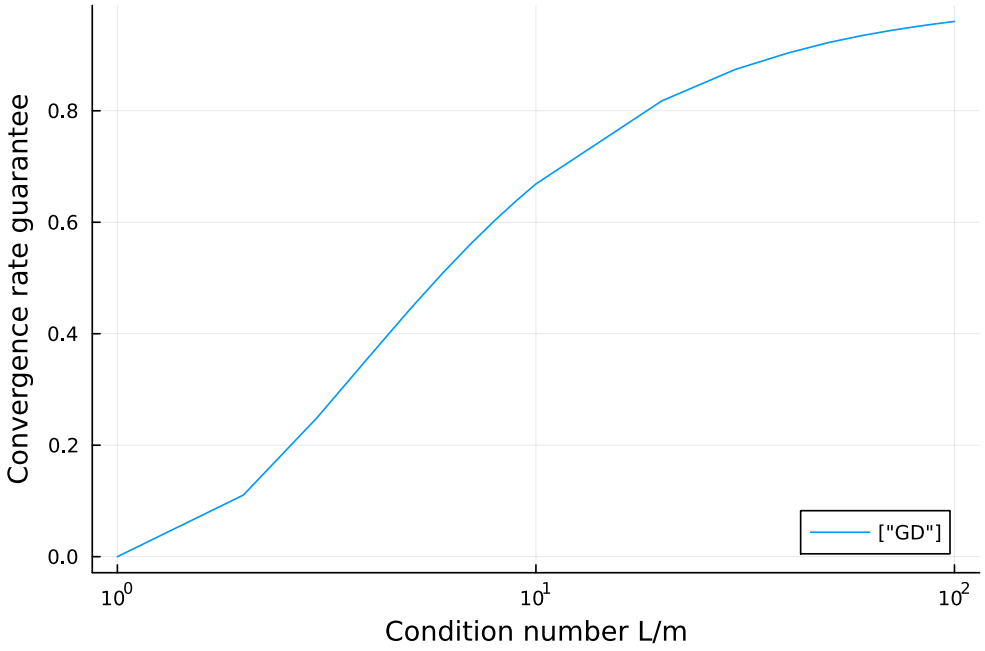


Figure 5.7: Convergence rate guarantee for 3 algorithms over m strong L smooth convex function

References

[1] 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.

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

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

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

[5] 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.

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

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

[8] 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.

[9] 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.

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