

JUPE: JULIATMPACKAGE 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. Reader 1

Reader: Dr. Reader 2

© 2024 Lam Ngoc Ha

This thesis titled

JUPE: JULIATMPACKAGE 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. Reader 1

Dr. Reader 2

ABSTRACT

JUPE: JULIATMPACKAGE IMPLEMENTATION OF OPTIMIZATION ALGORITHM ANALYSIS USING LYAPUNOV FUNCTION

by Lam Ngoc Ha

Gradient-based iterative algorithms are widely used method for solving optimization problems, with well-known applications including in the fields of machine learning and data science. Savings in computational capacity can be made if the best performing algorithm is chosen for the optimization problem being solved through algorithm analysis. JuPE, the main goal of this paper, is a Julia package that aims to provide a streamlined and simple way of performing analysis on gradient descent and two of its accelerated variants at solving a class of optimization problem using an approached based on Lyapunov functions.

Contents

List of Tables	iii
List of Figures	iv
Dedication	vi
Acknowledgements	vii
Acronyms	viii
1 Introduction to JuPE	1
1.1 Optimization problems and algorithms	2
1.2 Algorithm analysis	3
1.3 Julia programming language	4
1.4 Overview	5
2 Literature Review	6
3 Lyapunov-based approach	7
3.1 Iterative algorithms as Lur’e problems	7
3.2 Interpolation condition	8
4 More	9
4.1 References and links	9
4.2 Figures and tables	9
References	11

List of Tables

4.1	Gravimetric analysis of silver halides in a 1.27-mL sample of sea water. . . .	10
-----	--------------------------------------------------------------------------------	----

List of Figures

4.1	Figure captions should be long and descriptive because people actually read them, unlike the rest of the text. (Left) A block diagram make using Tikz. (Right) A plot made using Pgfplots. Note that the figures are not scaled, so the text size in the figures is consistent with the rest of the document. The source files for the figures are in the <code>graphics/</code> folder.	10
-----	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----

Dedication

I would like to dedicate this thesis to...

Acknowledgements

I would like to acknowledge. . .

Acronyms

FG Fast Gradient

TCP/IP Transmission Control Protocol/Internet Protocol

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 tries to find the shortest path to a destination by choosing the direction of travel while under constraints such as traffic laws or avoiding road work. Gradient-based iterative algorithms are a prominent 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 the term encompasses many algorithms, each can solve any of the many types of optimization problems with varying levels of speed and accuracy, making it very useful the ability to compare algorithms on specified metrics and identifying the one best suited for a problem. But while these algorithms are widely, their analysis have only been available to those with an in depth knowledge of the underlying math until recently [1]. The main work of this thesis presents a tool for analysis of gradient-based algorithms' performance characteristics accessible to 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 problem. After the program is given a class of functions, the algorithm to be analyzed and the performance metrics, it returns a guarantee speed at which the algorithm inputted can solve any function in the provided 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:

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

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

Where $f(x)$ is the optimization function and X is a constraint set. Here, x is the input 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 automobiles, amongst many others. These models are only possible due to the minimizing of loss functions, a fundamental part of their training where a function is used to quantify the dissimilarity between a model's output and the target 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 x_k (k representing the current iteration number) 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:

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

Where α is the step size, an adjustable parameter of the algorithm. α can affect the speed at which the algorithm converges, or whether it converges at all, in which case overshooting occurs. 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 :

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

While Nesterov’s Fast Gradient (FG) evaluates the gradient at an interpolated point:

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

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

1.2 Algorithm analysis

Let us consider the problem of minimizing a simple quadratic function:

$$f(x) = x^2/2 - 3 * x + 4 \quad (1.5)$$

Using (GD) and equation 1.2, substituting step size α with values 0.2, 0.5, and 2, and picking a starting point of x_0 . By solving the problem with each variant of the algorithm and counting the number of iterations each runs for before reaching within 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 and accuracy 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 before solving an optimization problem can mean a more accurate solution can be found and in fewer iterations. And while the example is of a simple function where overshoot in the case of the third tuning 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 bigger and more complex. Using the same example of training large language and self-driving models, the training process has been and might need to go on for years as more training data is available and the models need to continously improve all while using vast amounts of computational power. As a result, even a small improvement in the performance of the algorithm used can lead to large savings in time and energy.

Considering the quadratic function example, while it yielded an analysis of the algorithms’ performance, it required solving the optimization problem. Not only would this be computationally costly for any problem large enough to warrant being optimized numerically in the first place, any benefit finding an algorithm better at solving that problem is negated as the 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 example being [2]. In this paper, the author designed and conducted experiments where how the algorithms perform in typical applications is recorded. While this approach is can provide a general evaluation of an algorithm's performance, it is done empirically. Instead, there exists approaches toward analyzing algorithms [3], [4], and [5] that aim 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 guarantee 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.

Users can pick from one of the algorithms provided in the package or create their own iterative first-order algorithm by specifying how it is updated. The class of functions can be provided by detailing the characteristic of the set, such as 1 strong 10 smooth convex function. Users can specify a performance measure, which can be how far the iterate x_k is from the goal x_* or any quadratic combinations of the iterates. 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) discuss the mathematical approach that JuPE utilize, 2) break down the code structure of the program and how it functions, and 3) show some of the analysis that the package has done.

2

Literature Review

There exist in the literature many approaches to performing algorithm analysis. In 2014, Drori and Teboulle first introduced the method of reformulating worst-case algorithm analysis into a semidefinite program (SDP). Taylor, Hendrickx and Glineur built upon this work by turning the SDP into a convex optimization problem, creating the performance estimation problem (PEP) methodology. PEP showed by creating a finite representation for a class of smooth strongly convex functions using closed-form necessary and sufficient conditions, computing the worst-case performance of a first order method can be reduced to a convex SDP whose size is proportionate with the number of iterations the algorithm is run.

IQCs here

Of the methodologies above, PEP has been implemented into a computer program in PESTO, a MATLAB toolbox and PEPit, a Python package. The program when given a first-order method, a class of function, a performance measure, and an initial condition, will find the worst-case performance automatically. PESTO and PEPit presented an easy way to use the PEP approach to analyze gradient-based algorithms.

The main contribution of this thesis paper is to create a package sharing the same goal as PESTO and PEPit of providing an accessible and fast way to analyze the performance of first-order methods, leveraging the approach presented in [], in the Julia programming language.

3

Lyapunov-based approach

JuPE performs algorithm analysis by using the technique layed out by Van Scoy and Lessard in [7]. While JuPE is a blackbox tool, understanding the mathematical approach on which it is based is prerequisite to understanding the package's code and functionalities.

The steps of this technique, which will be discussed in detail over the sections of this chapter, consists of 1) viewing the algorithm as a Lur'e problem, 2) Replacing the nonlinear gradient with interpolation conditions that represent the class of smooth strongly convex functions, 3) Use lifting matrices to tighten to the interpolation condition representations, and 4) Prove whether a convergence rate is guaranteed by solving a convex semidefinite program.

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:

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

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

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

The algorithm can then be put into state-space representation as:

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

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

$$u_k = \nabla f(y_k) \quad (3.2c)$$

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, this step can be applied to any other first order methods. Deriving an LTI system represented by matrices out of an algorithm not only creates a representation easy to understand and operate on, but also in next sections enable the representation of the past states of an algorithm and the forming of a Lyapunov function that will guarantee a convergence rate.

3.2 Interpolation condition

For the remainder of this paper, we will focus on the only class of function currently supported by JuPE, m strong L smooth convex function $f \in F_{m,L}$.

While an LTI system is relatively simple to solve, the addition of the nonlinearity representing the gradient of the function is not. This step replaces the gradient of a smooth strongly convex function with a set of conditions on the input and output of that linearity using the characteristics of the class of function. This step relies on a theorem first presented in [4] and reformatted in [7]:

Theorem 1 ([4], Thm. 4; [7], 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:

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

4

More

4.1 References and links

We use the `hyperref` package to produce a pdf with hyperlinks. We also use the `cleveref` package for facilitating references. Using the `\cref` command will automatically use the correct prefix. You can also refer to multiple things at once by using multiple arguments, or you can refer to a range using `\crefrange`. For more information, see the documentation¹.

References can be cited with the `\cite` command, which produces something like [?, ?, ?, ?]. The `cite` package ensures the citations are ordered nicely and compressed when possible.

4.2 Figures and tables

Figures. We use the standard `figure` environment for figures. Diagrams should be placed in separate files in the `graphics/` folder and should use the `standalone` package. You can then use `includegraphics` as in Figure 4.1 to include the figure in your document. If you are using `pdflatex`, you can also use `includegraphics` to include PDF, JPEG, PNG, or other formats, but not EPS.

Tables. The `booktabs` package, which includes commands such as `\toprule`, `\midrule`, and `\bottomrule`, can be used to make nice tables. In general, never use vertical lines to separate columns. For more style tips on how to make nice tables, see². Here is an example of a nice table³.

¹<http://mirrors.ctan.org/macros/latex/contrib/cleveref/cleveref.pdf>

²<https://people.inf.ethz.ch/markusp/teaching/guides/guide-tables.pdf>

³<https://lazyscientist.wordpress.com/2021/07/23/make-better-tables-in-latex-using-booktabs/>

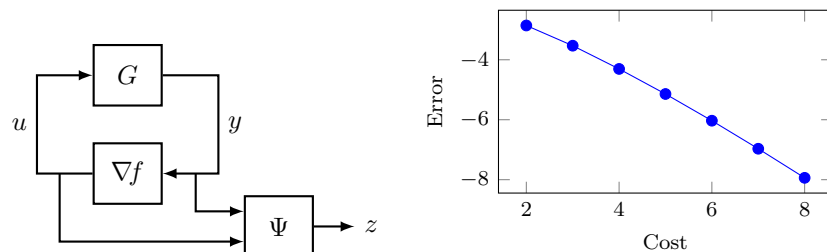


Figure 4.1: Figure captions should be long and descriptive because people actually read them, unlike the rest of the text. (Left) A block diagram made using Tikz. (Right) A plot made using Pgfplots. Note that the figures are not scaled, so the text size in the figures is consistent with the rest of the document. The source files for the figures are in the `graphics/` folder.

Table 4.1: Gravimetric analysis of silver halides in a 1.27-mL sample of sea water.

Qty of Sample	Test Tubes				Avg
	A	B	C	D	
Mass (g)	1.399	1.32	1.328	1.408	1.364
Density (g/mL)	1.10	1.04	1.05	1.109	1.07
Mass w/ Precipitate (g)	13.443	13.401	13.348	—	13.397
Mass AgCl (10^{-2} g)	9.0	9.2	8.7	—	8.9
Moles AgCl (10^{-4} mol)	6.28	6.42	6.08	—	6.50

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] 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.