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. Reader 1

Reader: Dr. Reader 2

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. Reader 1

Dr. Reader 2

ABSTRACT

JUPE: JULIA™PACKAGE 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 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 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 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 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 used, 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:

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

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

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 automotives, 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 the minimum of any function can be found by traversing it, 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 and iteratively updating it using the gradient of the function at the last iteration until it reaches a local minimum. 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 x_0 is the initial point, k is the current iteration number, $\nabla\ f(x_k)$ is the gradient of f at the current point x_k, and α is the step size. Following this update formula, in each iteration, x moves toward the goal x_*

As an adjustable parameter of the algorithm, α can affect the speed at which the algorithm converges, or whether it converges at all: If α is too small, the algorithm will have to run many iteration before reaching the minimum, but if α is too big, overshooting can occur where the iterations oscillates on either sides of the minimum without converging. 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:

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

Another algorithm that tries to solve GD's shortcomings is Nesterov's Fast Gradient (FG), which updates by evaluating the gradient at an interpolated point:

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

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

1.2 Algorithm analysis

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

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

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 can easily be identified and the number of iteration needed to find the minimum is small, the benefit of using an optimal algorithm only increases as the problem gets bigger and more complex. Looking again at the example of training large language and self-driving models, as more training data is available, the training process which has been going on for years will need to continue for many more, all of which consumes 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. For example, [2] presents the Adam algorithm and compared its performance to that of its peers by conducting experiments and drawing conclusions based on emperical evidence. While this approach can provide useful insights into how an algorithm performs, there exists other approaches toward analyzing algorithms such as [3], [4], and [5] that aim to find a mathematically provable performance guarantee of an algorithm over a class of functions. This worst-case guarantee is how JuPE perform analysis and will be refered to in this paper 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

Substantial literature has been published that aim to evaluate and compare the performance of algorithms using emperical evidiencce. In [2], a frequently cited paper, the author designed and conducted experiments where metrics such as cost per iteration of commonly used algorithms are recorded. The result gives a general idea of different algorithms' characteristic solving different problems.

There exist in the literature many approaches to deriving a performance bound of first-order optimization algorithms. 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 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. While the above mentioned two approaches give the performance bound in the form of a guarantee how close x_k is to the goal x_* after a fixed number of iterates, the method used in this package proves that the performance measure inputted by the users decreases at a guaranteed rate throught out the optimizing process.

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 follows the PEP methodology and presented an easy way to analyze gradient-based algorithms.

The main contribution of this thesis paper is to create an alternative computer program 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, leveraging the approach presented in [7], 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 prequisite 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 semidefinte 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:

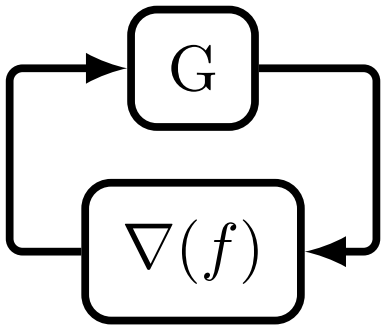

Figure 3.1: Block diagram representation of iterative algorithms

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:

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

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 $f \in F_{m,L}$.

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, our approach replaces the nonlinearity with a set of conditions on its input and output, which provides necessary and sufficient conditions under which there exists a function $f \in F_{m,L}$ that interpolates a given finite set of argument-function value-gradient triplets. These interpolation conditions depends on the characteristics of the function class and was first formulated 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:

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

Where the Euclidean-norm of a vector x is denoted as $||x||$. Basing on this theorem, [[7], Cor 4] presented a non-negative linear combination of inequalities to create a tight representation of the class of function, which can be rewritten in a way easier to implement into code as:
Corollary 2 ([7], Cor. 4)**.** Given a function $f \in F_{m,L}$ and let $y_k, ..., y_{k-l}$ be a sequence of iterates; $u_{k-i} = \nabla f(y_{k-i})$ and $f_{k-i} = f(y_{k-i})$ for $i \in 0, ...l$. Then the inequality:

holds for all $\Lambda \in \mathbb{R}^{(l+2)*(l+2)}$ and $\Lambda >= 0$, and $\Pi(\Lambda)$ and $\pi(\Lambda)$ are defined as:

This step in the methodology is similar to that in [1]. By using interpolation conditions, the gradient of smooth strongly convex class of function inputted is represented by non-negative inequalities. This combined with the use of Lyapunov-functions to prove convergence rate which will be presented in 3.4 allows analysis to be done be solving a convex program

3.3 Lifting algorithm

3.4 Lyapunov function certification

$$V(x) =< P, X[x;u] >=< X^T P, [x;u] > \tag{3.3}$$

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 updated state and the goal $(x_1 - x_s)^2$.
- 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

As both the transpose of a vector and the gradient of a function uses the notation " ' ", the transpose of a vector as mentioned in section 4.1 is an oracle. This oracle is created when a vector expression is defined and is stored inside a wrapper data structure. During the inner product 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 therefore attribute no constraint to the vector being sampled.

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 = $\mathrm{R}^n()$

```
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 [8] 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

In JuPE, when a variable expression is defined in an inner product space, which include states of an algorithm, the starting and goal points and the gradient of a function at a point, it is a vector. However, all three components needed to form the Lyapunov functions exist as real scalars. Constraints created by the interpolation conditions are applied to the inner product and norm of state and input vectors, which are real scalar as can be seen in 4.11, while the performance measure formed from state vectors has to be real scalars in order to be measured. Similarly, the state update matrix used to construct the Lyapunov function is formed as the inner product of state space matrices representing the algorithm and therefore are scalars.

As the JuMP modeling language does not support the expression and constraint data structures presented in chapter 4, how the components of the Lyapunov function are represented must be transformed to real numbers before they can be inputted into a JuMP model. And as all three components are created from real scalars formed from the states and inputs, JuPE achieve this by expressing these components as a linear function of real variable expressions. 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. Defining 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]

```
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

to find that expression. This matrix, 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.

```
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

As the performance measure is defined, a scalar expression is created. The linear form matrix of the performance measure is the first of the three components needed to form the Lyapunov function. For example, the performance measure in 4.1, which is defined as $(x0 - xs)^2$ and which evaluates into $|x0|^2 - < xs, x0 > - < x0, xs > + |xs|^2$, is:

5.2 Algorithm and state update

4.1 shows 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

```
𝒫 = vec(linearform( [x; u] => performance ))
9-element Vector{Int64}:
    -1
    1
    -1
    1
    0
    0
    0
    0
    0
```

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

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:

```
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^+

Following the steps presented in chapter 3, the Lyapunov function can begin to be formed thusly:

Optimization variable P An optimization variable P is defined 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

Lyapunov function for The first Lyapunov function L1 is formed as: L1 = $\mathcal{P}$ - X'*P.

Lyapunov function for decreasing The second Lyapunov function L2 is formed as: L2 = X^{+}'*P - ρ*X'*P.

5.3 Constraints

As presented in 4.3, the oracle created from the class of function provided as the input automatically create the constraints that interpolate that class of function. These constraints are linearized and added to the optimization in 2 steps:

Optimization variable multipliers For each constraint, two optimization variables are defined similar to optimization variable P. 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. Depending on the type of constraint.

Constraint on multiplier The JuMP variables multipliers are constrained in the JuMP model depending on the constraint expression they were created from: The multiplier is not constrained if the expression is constrained to be zero, is constrained to be non-negative if the expression is constrained to be non-negative, and is 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 can then be created and added to the Lyapunov functions.

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

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

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

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

$$\begin{bmatrix}\lambda_1 & \lambda_2\end{bmatrix} * \begin{bmatrix}(x0 - xs)^2 \\ (x0 - xs)^2 - 3 * |xs|^2\end{bmatrix} = M * \begin{bmatrix}|x0|^2 \\ < xs, x0 > \\ |xs|^2\end{bmatrix} \tag{5.2a}$$

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

In all three cases, the resulting linear form matrix is the linear form of the constraint expression scaled by the JuMP variable multipliers. As 2 multiplier variables are defined for each constraints, the resulting linear form matrices are each transformed into a vector and added the two Lyapunov functions.

5.4 Derived feasibility and bisection search

5.5 Result

After running JuPE to perform algorithm analysis on three 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

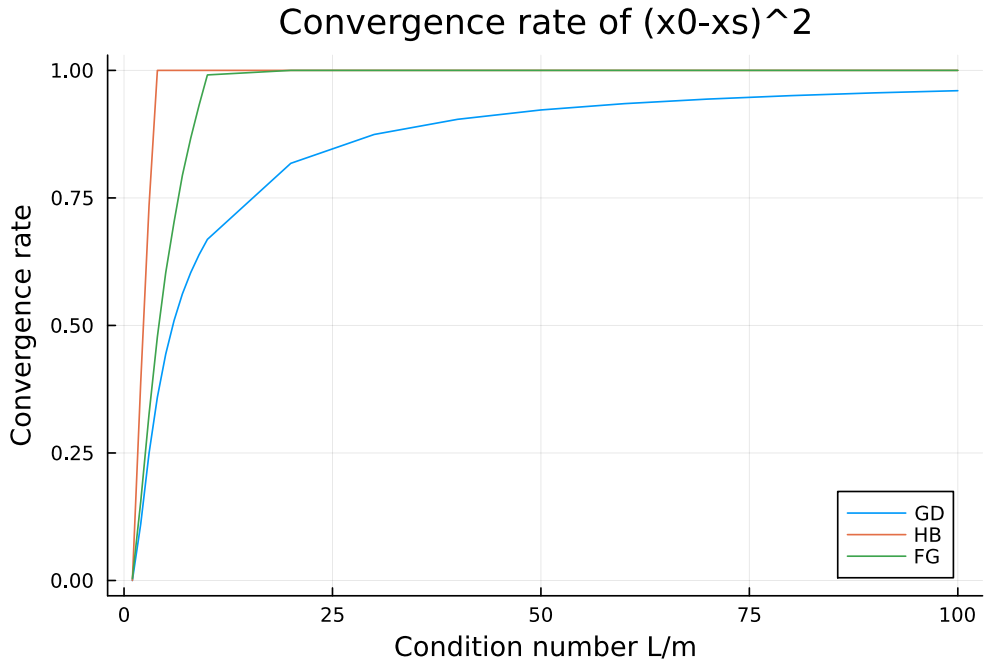

Figure 5.7: Convergence rate

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.

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