

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
SANTIAGO - CHILE



“FOURIER APPROXIMATION OF THE JACOBIAN MATRIX
IN LARGE-SCALE NON-LINEAR LEAST SQUARE
PROBLEMS ”

JORGE SANHUEZA CÉSPEDES

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN INFORMÁTICA

Profesor Guía: Claudio E. Torres, Ph.D.
Profesor Correferente: Roberto Leon Ph.D.

Agosto - 2023

DEDICATORIA

A mis padres, mi familia y, en especial, a mi abuelo quien con su ejemplo me enseñó la importancia de de la familia, la dedicación y la generosidad.

AGRADECIMIENTOS

Agradezco a los profesores de la Universidad Federico Santa María cuya formación ha sido vital para llegar a este punto de mi vida, formando mi pensamiento crítico y mis valores. En particular, quiero agradecer al Profesor Claudio Torres que me ha entregado invaluable conocimientos como profesor y como persona.

Agradezco un millón a mi familia por el tremendo apoyo que me han dado en este gran proceso y en toda mi vida.

RESUMEN

Resumen— Esta memoria propone un enfoque novedoso para resolver sistemas de ecuaciones no lineales sobredeterminados de gran tamaño sin necesidad de almacenar explícitamente la matriz Jacobiana y su transpuesta. El enfoque utiliza técnicas de compresión de vectores basadas en la transformada de Fourier para generar una matriz Jacobiana “sparse” que puede almacenarse de manera explícita. Además, se desarrolla un algoritmo para calcular de manera eficiente el producto de la matriz Jacobiana transpuesta por un vector para problemas de mínimos cuadrados de gran tamaño.

Los objetivos de esta investigación son dos: introducir un método de aproximación efectivo para la matriz Jacobiana y proporcionar un algoritmo eficiente para calcular el producto de la matriz Jacobiana transpuesta con un vector. Al aprovechar la transformada de Fourier, nuestro enfoque logra importantes ahorros computacionales y reduce los requisitos de almacenamiento asociados con sistemas de gran escala. Esto resulta especialmente valioso en escenarios donde el almacenamiento explícito de la matriz Jacobiana y su transpuesta es inviable debido a la dimensionalidad del problema.

El comportamiento del enfoque propuesto se estudia ampliamente en el contexto del algoritmo ns-LSQR, que resuelve problemas de mínimos cuadrados utilizando aproximaciones de la matriz Jacobiana y su transpuesta. Nuestros hallazgos demuestran que el método de aproximación propuesto es viable en realidad; sin embargo, cuando disminuye el porcentaje de componentes en la aproximación, se requiere una mayor dimensión del subespacio de Krylov para garantizar la convergencia del método. Esto destaca el equilibrio entre la precisión de la aproximación y la eficiencia computacional. El algoritmo calcula de manera eficiente el producto de la matriz Jacobiana transpuesta con un vector mediante el uso de la aproximación propuesta y propiedades de la Transformada de Fourier con una complejidad computacional de $\mathcal{O}((n+1)m \log(m))$ permitiendo la solución de sistemas no lineales sobredeterminados de gran escala.

La importancia de esta investigación radica en su potencial para avanzar en el campo de la resolución de sistemas no lineales de gran escala al proporcionar un enfoque alternativo que evita las limitaciones de almacenamiento de la matriz Jacobiana y su transpuesta. Esto abre nuevas posibilidades para aplicar técnicas de optimización no lineal en escenarios prácticos donde el almacenamiento explícito no es viable. Trabajos futuros pueden involucrar la exploración del uso de otras transformadas matemáticas, como las transformadas wavelet, y evaluar el rendimiento de la aproximación en otros algoritmos de optimización.

Palabras Clave— Problema no lineal a gran escala, Sistemas de ecuaciones sobredeterminados, Mínimos cuadrados, Matriz Jacobiana, Transformada de Fourier.

ABSTRACT

Abstract— This undergraduate thesis proposes a novel approach to solving large-scale over-determined nonlinear systems of equations without the need for explicit storage of the Jacobian matrix and its transpose. The approach utilizes vector compression techniques based on the Fourier transform to generate a sparse Jacobian matrix that can be stored explicitly. Additionally, an algorithm is developed for efficiently computing the product of the Jacobian matrix transposed with a vector for large-scale least-squares problems.

The objectives of this research are twofold: to introduce an effective approximation method for the Jacobian matrix and to provide an efficient algorithm for computing the product of the transposed Jacobian matrix with a vector. By leveraging the Fourier transform, our approach achieves significant computational savings and reduces the storage requirements associated with large-scale systems. This makes it particularly valuable in scenarios where explicit storage of the Jacobian matrix and its transpose is infeasible due to the problem's dimensionality.

The behavior of the proposed approach is extensively studied within the context of the ns-LSQR algorithm, which solves least-squares problems using approximations of the Jacobian matrix and its transpose. Our findings demonstrate that the proposed approximation method is actually viable however when the percentage of components in the approximation decreases, a higher dimension of the Krylov subspace is required to ensure the convergence of the method. This highlights the trade-off between approximation accuracy and computational efficiency. The algorithm efficiently calculates the product of the transpose Jacobian matrix with a vector by using the proposed approximation and properties of the Fourier Transform with a computational complexity of $\mathcal{O}((n + 1) m \log(m))$, allowing the solution of large-scale overdetermined nonlinear systems.

The significance of this research lies in its potential to advance the field of solving large-scale nonlinear systems by providing an alternative approach that circumvents the storage limitations of the Jacobian matrix and its transpose. This opens up new possibilities for applying nonlinear optimization techniques in practical scenarios where explicit storage is not viable. Future work may involve exploring other mathematical transforms, such as wavelet transforms, and evaluating the performance of the approximation in other optimization algorithms.

Keywords— Large-scale nonlinear problems, Systems of over-determined nonlinear problems, Least squares, Jacobian matrix, Fourier Transform.

Acronyms

LSQR: An algorithm for Sparse Linear Equations and Sparse Least Squares.

ns-LSQR: non-symmetric LSQR, a modification of the LSQR algorithm.

GD: Gradient Descent algorithm.

SG: Spectral Gradient algorithm.

NCG: Nonlinear Conjugate Gradient.

NM: Newton Method.

QM: Quasi-Newton Method.

GN: Gauss-Newton Method.

MC: Max Components, approach defined in the document to approximate a matrix.

EO: Variation of MC

ALO: Variation of MC

DFT: Discrete Cosine Transform

FFT: Fast Fourier Transform

DCT: Discrete Cosine Transform

Contents

RESUMEN	iv
ABSTRACT	v
Acronyms	vi
List of Figures	ix
Introduction	1
Chapter 1: Problem Definition	4
1.1 Objectives	5
1.1.1 General Objective	5
1.1.2 Specific Objectives	5
Chapter 2: Literature Review	6
2.1 Classic Newton's method	7
2.2 Gauss-Newton's method	9
2.3 Levenberg Marquard	9
2.4 Jacobian-Free Newton-Krylov method	11
2.5 LSQR	14
2.6 Non-symmetric LSQR: ns-LSQR	14
Chapter 3: Approximations	17
3.1 Diagonal matrices	17
3.2 Max Components (MC) approach	19
3.3 Transforms	20
Chapter 4: Fourier Approximation of the Jacobian Matrix	22
4.1 The Discrete Fourier Transform and the Discrete Cosine Transform	22
4.2 The approximation	24
4.3 Jacobian approximation in large scale least squares	26
Chapter 5: Proposed Algorithms	28
5.1 Memory and Computational Complexity	29
Chapter 6: Numerical Experiments	31
6.1 Selected Functions and Their Jacobians	31
6.1.1 Explicit Jacobian of Test Problems	32
6.2 Fourier approximation: The use DFT vs the use DCT	34
6.3 Accuracy of the Jacobian Fourier Approximation	35
6.4 Accuracy of the Jacobian Fourier Approximation in large-scale non-linear least squares: ns-LSQR	37

Chapter 7: Conclusions	41
REFERENCIAS BIBLIOGRÁFICAS	43

List of Figures

1	Problem tree of the large-scale least squares problem.	4
2	Reconstruction errors of the Fourier approximations using the FFT and DCT keeping different percentages of components.	35
3	Relative error of the Jacobian matrix and the minimum eigenvalues of $\tilde{J}^T J$ of the Trigonometric and Logarithmic functions.	36
4	Relative error of the Jacobian matrix and the minimum eigenvalues of $\tilde{J}^T J$ of the Extended Powell and Penalty II functions.	37
5	Mean Relative error of the Jacobian matrix of the Linear function for different percentages of values kept in the approximations, with $n = 1000$ and $m = 2000$ of 30 trials. ALO and DCT ALO are on top of each other.	38
6	Relative error of the ns-LSQR method for the Random Linear Function achieved at each restart using $n = 1000$ and $m = 2000$	38
7	Relative error of the ns-LSQR method for the Extended Powell Function achieved at each restart using $n = 1000$ and $m = 1000$	39
8	Relative error of the ns-LSQR method for the Trigonometric Function achieved at each restart using $n = 1000$ and $m = 1000$	39
9	Relative error of the ns-LSQR method for the Penalty Function II achieved at each restart using $n = 1000$ and $m = 2000$	40
10	Relative error of the ns-LSQR method for the Random Linear Function achieved at each restart using $n = 1000$ and $m = 2000$	40

Introduction

Non-linear problems are prevalent in many fields of study such as Engineering (e.g., fluid dynamics, structural analysis), Mathematics (e.g., non-linear optimization), Biology (e.g., population dynamics), and Physics (e.g., non-linear wave equations). For instance, in fluid dynamics, the flow of air or liquids around objects can be highly non-linear and difficult to model analytically [1]. In population dynamics, non-linear models are often used to study the interactions between species in an ecosystem [2]. In all of these cases, solving non-linear systems of equations are crucial for understanding and predicting the behavior of complex systems.

Formally, a *non-linear system of equations* is expressed by,

$$\mathbf{F}(\mathbf{x}) = \begin{bmatrix} f_1(x_1, x_2, \dots, x_n) \\ f_2(x_1, x_2, \dots, x_n) \\ \vdots \\ f_m(x_1, x_2, \dots, x_n) \end{bmatrix} = \mathbf{0} \quad (0.1)$$

where $\mathbf{F} : \mathbb{R}^n \rightarrow \mathbb{R}^m$ is a map, $\mathbf{x}$ represent the root of the problem, in other words, where the value of $\mathbf{F}(\mathbf{x})$ is equal to $\mathbf{0}$, the zero vector in $\mathbb{R}^m$, however, not all systems have roots or vectors that satisfy all the equations simultaneously and need different methods for solving them. The n represents the number of components of the input $\mathbf{x}$, or the n unknowns of the problem, and m the components of the output vector formed by each of the m equations.

If $m < n$ the problem is called *Under-determined non-linear system of equations*, if $m > n$ the problem is an *Over-determined non-linear system of equations*. Otherwise is simply called a non-linear system of equations.

When solving a system of equations, our goal is to find values for each of the unknowns that simultaneously satisfy all the equations in the system. In the case of an overdetermined system of equations, where the number of equations is greater than the number of unknowns, the system is often inconsistent and does not have a solution.

In this work, our focus is on solving over-determined non-linear systems of equations, where even though an exact solution may not exist, we can approximate a solution that satisfies every equation. A commonly used method to approximate the solution of such systems is through the minimization of the sum of the squares of the residuals, known as a least-squares problem.

We define the function $g(\mathbf{x})$ as half the squared Euclidean distance between the function $\mathbf{F}(\mathbf{x})$ and the zero vector $\mathbf{0}$.

$$g(\mathbf{x}) = \frac{1}{2} \|\mathbf{F}(\mathbf{x}) - \mathbf{0}\|_2^2 \quad (0.2)$$

The least-squares problem aims to find the vector $\mathbf{x}$ where the value of $g(\mathbf{x})$ is minimized,

$$\min_{\mathbf{x} \in \mathbb{R}} g(\mathbf{x}) \quad (0.3)$$

Note that the factor of $\frac{1}{2}$ is used for mathematical convenience.

There are various methods available to solve optimization problems, including *Gradient Descent* (GD) [3], *Spectral Gradient* (SG) [4], *non-linear Conjugate Gradient* (NCG) [5], *Newton's Method* (NM) [6], *Quasi-Newton method* (QN) [7], *Gauss-Newton* (GN) [8], and others. These techniques are well-suited for small-scale problems where computational and storage requirements are manageable. However, when dealing with large-scale problems, these methods may encounter challenges due to high computational demands or the size of matrices exceeding available memory.

A crucial factor in the feasibility of these methods lies in the efficient calculation of the gradient of $g(\mathbf{x})$, which is represented as $\nabla g(\mathbf{x}) = J^T(\mathbf{x})\mathbf{F}(\mathbf{x})$, where $J(\mathbf{x})$ is the Jacobian matrix of $\mathbf{F}(\mathbf{x})$. The reliance on the Jacobian matrix, or its transpose, poses a potential memory bottleneck.

In reality, the suitability of these methods depends on whether there exists an efficient procedure to calculate the Jacobian matrix and its transpose with a vector. If such an algorithm is available, memory-related issues may be mitigated even for large-scale problems. Explicitly storing the Jacobian matrix could be feasible with the aid of this procedure, e.g. a specialized external software that calculates the product.

However, in cases where only the function $\mathbf{F}(\mathbf{x})$ is available without a straightforward method to calculate the Jacobian and its transpose, alternative approaches become necessary. Dealing with large-scale problems under these circumstances requires exploring specialized algorithms that can approximate or circumvent the need for the full Jacobian, offering more memory-efficient solutions. This is particularly crucial for maintaining computational efficiency in scenarios where memory constraints are a limiting factor.

As a result of these restrictions, two key questions arise: 1) How can we efficiently approximate the Jacobian matrix and its transpose within the memory limitations of large-scale least squares problems? and 2) How can we use this approximation to solve the least squares problem accurately and efficiently?

The document tries to respond to these questions in some logical order to finally propose an innovative way of approximating the Jacobian Matrix in the context of large-scale least squares via the use Fourier transform.

The manuscript is structured as follows: In Chapter 1 the context of the problem is set, the causes and consequences of the problem are made explicit and the goals of the memoir are stated. In Chapter 2, we review existing methods to solve non-linear least-squares problems and approximations of the Jacobian products. Chapter 3 provides the theoretical background for our proposed solution, which is presented in Chapter 4. In Chapter 5, we present numer-

ical experiments to evaluate the performance of our approach, and in the final chapter, we summarize our findings and highlight areas for future research.

Chapter 1

Problem Definition

The problem belongs to the field of Scientific Computing. This branch of Informatics “is the collection of tools, techniques, and theories required to solve on a computer the mathematical models of problems in science and engineering” [9].

As explained in the introduction, when dealing with the large-scale over-determined non-linear problem we can solve it by means of a least squares problem which requires access to the Jacobian matrix and its transpose.

A graphical representation of the problem is presented in Figure 1, depicting a problem tree. At the core and trunk of the tree lies the challenge of storing the Jacobian matrix for solving Equation (2.2). The root causes of this issue can be attributed to two primary factors: the scale of the problem, including the number of equations and variables involved, and the computational and memory capabilities of the system responsible for solving the problem. The consequences of this problem represented as the leaves of the problem tree, are the performance metrics of the minimization method employing an approximation of the Jacobian matrix instead of the unfeasible explicit matrix. The key performance metrics include computation time, solution accuracy, and memory utilization within the method.

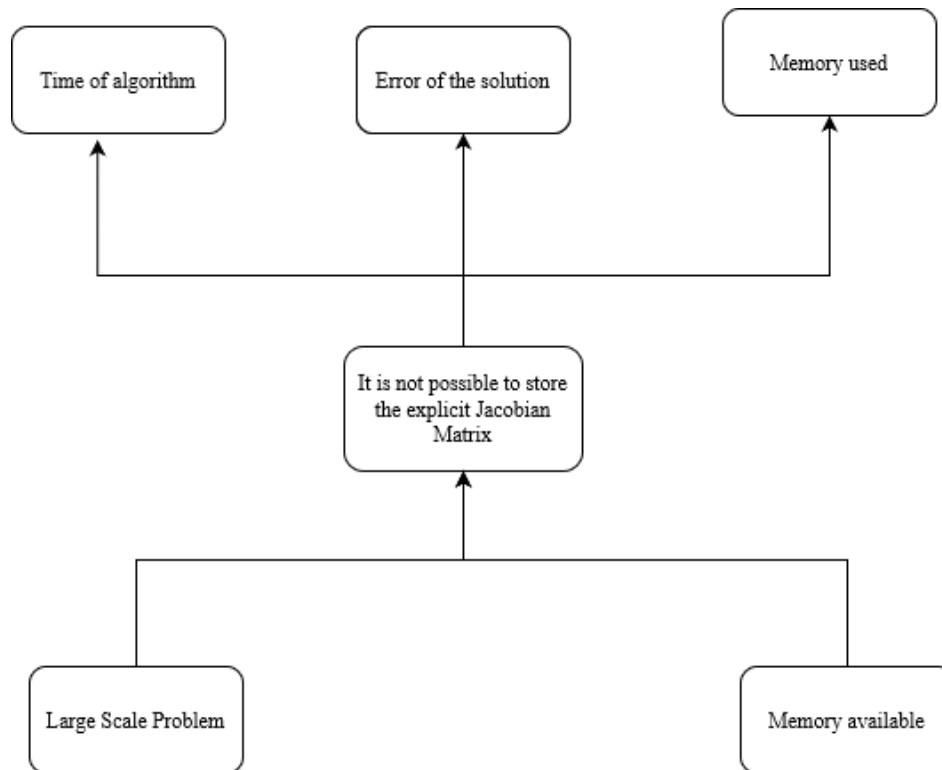


Figure 1: Problem tree of the large-scale least squares problem.

1.1 Objectives

1.1.1 General Objective

- The main objective of the problem is to efficiently approximate the Jacobian matrix and its transpose within the memory limitations of large-scale least squares problems

1.1.2 Specific Objectives

- To review and analyze existing methods for approximating the Jacobian matrix and its transpose in the context of large-scale least squares problems.
- To propose an innovative method for approximating the Jacobian matrix using the Fourier transform.
- To propose an innovative method for approximating the Jacobian matrix transposed product with this Fourier approximation.
- To evaluate the performance of the proposed methods through numerical experiments.

Chapter 2

Literature Review

In this section, we provide a literature review that explores various methods for solving over-determined non-linear systems of equations. Before delving into the details of these methods, we establish the notation that will be used throughout this document. The notation includes the $n \times n$ identity matrix denoted as I_n , the Euclidean two-norm denoted as $\|\cdot\|$, the zero vector in $\mathbb{R}^n$ denoted as $\mathbf{0}_n$, the $\mathbb{R}^n$ vector filled with ones denoted as $\mathbf{1}_n$, and the transpose of a matrix denoted as $()^T$.

An over-determined non-linear system of equations is given by,

$$\mathbf{F}(\mathbf{x}) = \mathbf{0}_n, \quad (2.4)$$

where the nonlinear function $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$ is defined as,

$$\mathbf{F}(\mathbf{x}) = \begin{bmatrix} f_1(\mathbf{x}) \\ f_2(\mathbf{x}) \\ \vdots \\ f_m(\mathbf{x}) \end{bmatrix} = \begin{bmatrix} f_1(x_1, x_2, \dots, x_n) \\ f_2(x_1, x_2, \dots, x_n) \\ \vdots \\ f_m(x_1, x_2, \dots, x_n) \end{bmatrix}.$$

The system is over-determined because we have more equations than unknowns, that is $m > n$

A classical approach for solving (2.4) is by means of solving an unconstrained minimization problem.

Let $g(\mathbf{x})$ be the function,

$$g(\mathbf{x}) = \frac{1}{2} \|\mathbf{F}(\mathbf{x})\|^2 = \frac{1}{2} \sum_{i=1}^m f_i^2(\mathbf{x}) \quad (2.5)$$

Then the minimization problem is given by the following equation,

$$\min_{\mathbf{x} \in \mathbb{R}^n} g(\mathbf{x}) \quad (2.6)$$

Various methods have been developed to solve this minimization problem, and in this section, we will introduce some of them.

A classical approach involves using root-finding methods based on the iteration:

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \Delta \mathbf{x}_i, \quad (2.7)$$

starting from an initial condition $\mathbf{x}_0$. The vector $\mathbf{x}_0$ can be an estimation of the solution or an arbitrary vector. Carefully choosing the initial guess can improve the convergence of the

method. The step Δx_i is computed by solving a particular linear system of equations, which will be reviewed in each classical method described in the following subsections.

Before describing the classical method for solving least squares problems, we want to quickly reference how large-scale systems of equations are actually being solved. In particular, we want to address the hybrid projection methods [10]. Hybrid projection methods combine iterative projection techniques with variational regularization methods to efficiently solve large-scale linear and nonlinear inverse problems [11] [12]. These methods first project the inverse problem onto a low-dimensional subspace using iterations like GMRES [13] or LSQR [14]. This allows approximating solutions to problems where explicitly forming the full system is infeasible. The projected problem is then solved using regularization methods like Tikhonov [15] [16] to handle ill-posedness and noise while adaptively selecting the regularization parameter [17] [18]. Hybrid methods leverage the strengths of both projection techniques and variational regularization in a synergistic framework [19]. They are widely used for large-scale problems across applications like image processing [20], tomography [21], and atmospheric modeling+[22] where direct methods are impractical. Hybrid projection methods enable stable inversion of ill-posed systems where the forward model or data may also be nonlinear [23].

2.1 Classic Newton's method

One of the most well-known methods for solving non-linear systems is *Newton's Method* (NM) [6]. It involves solving the following system of equations:

$$J(x_i) \Delta x_i^{NM} = -F(x_i) \quad (2.8)$$

where $J(x_i)$ represents the Jacobian matrix of F evaluated at x_i .

For Newton's Method to converge, the Jacobian matrix of F must be nonsingular in a neighborhood of x . The method exhibits local and quadratic convergence when the initial guess x_0 is sufficiently close to the root. However, as the dimension of the problem increases, explicitly computing the Jacobian matrix becomes memory-intensive. To ensure convergence for initial guesses that are far from the root, globalization techniques can be employed [24].

Algorithm 1 outlines the steps of Newton's Method.

It is important to note that Newton's Method is typically used for square matrices, where $F : \mathbb{R}^n \rightarrow \mathbb{R}^n$. When dealing with an over-determined system, Newton's Method may fail because the problem might not have a root and the Jacobian matrix is rectangular, making it unsuitable for a linear solver.

The Quasi-Newton Method is another variation of Newton's Method used for solving systems of nonlinear equations. However, unlike Newton's Method, which requires the calculation of the exact Jacobian matrix, the Quasi-Newton Method approximates the Jacobian matrix.

Algorithm 1 Algorithm of Newton's Method

Require: A function F , the number of equations m , the number of unknowns n , an initial guess x_0 .

Ensure: An approximation solution x_k for $\min_{x \in \mathbb{R}^n} \frac{1}{2} \|F(x)\|^2$.

for i from 0 to ∞ until some convergence criteria **do**

 Build $J(x_i)$

 Solve the linear system $J(x_i) \Delta x_i = -F(x_i)$.

if The new trial solution $x_i + \Delta x_i$ improves the current solution x_i **then**

$x_{i+1} = x_i + \Delta x_i$

else

$x_{i+1} = x_i$

end if

end for

return The current solution x_i .

At each iteration, the Quasi-Newton Method solves the linear system:

$$B(x_i) \Delta x_i = -F(x_i) \quad (2.9)$$

where $B(x_i)$ is an approximation of the Jacobian matrix at the current approximation x_i , Δx_i is the update to the current approximation, and $F(x_i)$ is the vector of function evaluations at the current approximation.

The key feature of the Quasi-Newton Method is that it avoids the computationally expensive calculation of the full Jacobian matrix by updating and refining the approximation $B(x_i)$ iteratively as the method progresses. This way, the method converges to the solution while using a more computationally efficient approach compared to standard Newton's Method.

A popular and widely used variant of the Quasi-Newton Method is the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm [25], which efficiently maintains the approximation of the inverse Hessian matrix, allowing for faster convergence.

Inexact Newton's Method [26] is a variation of Newton's Method used to solve systems of nonlinear equations. It involves solving a linear system, similar to Newton's Method but introduces a correction term to allow for approximate solutions. The method solves the linear system:

$$J(x_i) \Delta x_i = -F(x_i) + r(x_i) \quad (2.10)$$

where $J(x_i)$ is the Jacobian matrix, Δx_i is the update to the current approximation x_i , and $F(x_i)$ is the vector of function evaluations at the current approximation. The correction term $r(x_i)$ allows for approximate solutions with a tolerance condition given by $\frac{\|r_k\|}{\|F(x_k)\|} \leq \eta_k$.

Despite its name, Inexact Newton's Method is not limited to square systems and can handle both square and rectangular systems of equations, providing flexibility in handling various problem dimensions.

2.2 Gauss-Newton's method

The *Gauss-Newton's method* [8] is an adaptation of Newton's method specifically designed to handle over-determined problems.

While the obvious solution of (2.8) is given by

$$\Delta \mathbf{x}_i = - (J(\mathbf{x}_i))^{-1} \mathbf{F}(\mathbf{x}_i), \quad (2.11)$$

where $J(\xi)$ is the Jacobian matrix, and the inverse is only defined for square matrices. In the case of over-determined systems, we seek the "best solution" to the system, which can be obtained by solving the least-squares problem:

$$\min_{\Delta \mathbf{x}} \| -\mathbf{F}(\mathbf{x}_i) - J(\mathbf{x}_i)\Delta \mathbf{x}_i^{GN} \|. \quad (2.12)$$

By using the normal equations, we solve the following linear system of equations:

$$J^T(\mathbf{x}_i)J(\mathbf{x}_i)\Delta \mathbf{x}_i^{GN} = -J^T(\mathbf{x}_i)\mathbf{F}(\mathbf{x}_i). \quad (2.13)$$

However, Gauss-Newton's method still involves the computation of J^T and J , making it unsuitable for large-scale problems. Additionally, it requires that J be full rank. The algorithm 2 describes Gauss-Newton's Method

2.3 Levenberg Marquard

Levenberg-Marquardt (LM) [27] also known as damped least-squares, solves for

$$(J^T(\mathbf{x}_i)J(\mathbf{x}_i) + \lambda_i I_n) \Delta \mathbf{x}_i^{LM} = -J^T(\mathbf{x}_i)\mathbf{F}(\mathbf{x}_i) \quad (2.14)$$

where $\lambda_i > 0$ is a regularization parameter or damping factor. The Algorithm 3 describes the classic Levenberg-Marquard algorithm. The damping parameter ensures that the matrix on the left-hand side of the equation is full rank, which makes it possible to use a linear solver to solve (2.14).

This method has been extensively studied by many researchers to better its performance, but will highlight one by Chen (2016) [28] called *New Modified Levenberg-Marquardt*(NMLM).

Algorithm 2 Algorithm of *Gauss-Newton's Method*

Require: A function F , the number of equations m , the number of unknowns n , an initial guess x_0 , a tolerance η .

Ensure: An approximation solution x_k for $\min_{x \in \mathbb{R}^n} \frac{1}{2} \|F(x)\|^2$.

for i from 0 to ∞ until some convergence criteria **do**

 Build $J(x_i)$

 Build $J(x_i)^T$

 Solve the linear system $J^T(x_i) J(x_i) \Delta x_i^{NM} = -J^T(x_i) F(x_i)$

if The new trial solution $x_i + \Delta x_i$ improves the current solution x_i **then**

$x_{i+1} = x_i + \Delta x_i$

else

$x_{i+1} = x_i$

end if

end for

return The current solution x_i .

Algorithm 3 Algorithm of Levenberg-Marquardt

Require: A function F , the number of equations m , the number of unknowns n , an initial guess x_0 , a tolerance η , $\lambda_0 > 0$.

Ensure: An approximation solution x_k for $\min_{x \in \mathbb{R}^n} \frac{1}{2} \|F(x)\|^2$.

for i from 0 to ∞ until some convergence criteria **do**

 Build $J(x_i)$

 Solve the linear system $(J^T(x_i) J(x_i) + \lambda_i I_n) \Delta x_i = -J^T(x_i) F(x_i)$.

if The new trial solution $x_i + \Delta x_i$ improves the current solution x_i **then**

$x_{i+1} = x_i + \Delta x_i$

else

$x_{i+1} = x_i$

end if

 Compute λ_{i+1} from λ_i depending on the progress of x_{i+1} from x_i

end for

return The current solution x_i .

This method computes two additional approximate steps by solving a similar system of equations and employing line search to find a suitable trial step for the iteration using the three approximated steps. These additional steps are especially useful to save more Jacobian calculations and achieve a faster convergence rate.

2.4 Jacobian-Free Newton-Krylov method

Jacobian free methods do not need to explicitly store the Jacobian, instead use a matrix-vector product of the Jacobian times and arbitrary vector, to find a procedure to compute this product we consider the multivariate Taylor expansion of F

$$F(\mathbf{x} + \epsilon \mathbf{v}) = F(\mathbf{x}) + \epsilon J(\mathbf{x})\mathbf{v} + \dots \quad (2.15)$$

Leaving aside the high-order terms we get,

$$J(\mathbf{x})\mathbf{v} \approx \frac{F(\mathbf{x} + \epsilon \mathbf{v}) - F(\mathbf{x})}{\epsilon} \quad (2.16)$$

This is known as *Forward Difference* approximation. Similarly, we do the same with $F(\mathbf{x} - \epsilon \mathbf{v})$ and get a *Backward Difference* approximation, that is

$$J(\mathbf{x})\mathbf{v} \approx \frac{F(\mathbf{x}) - F(\mathbf{x} - \epsilon \mathbf{v})}{\epsilon} \quad (2.17)$$

Also using 2.16 with 2.17 we can derive the *Central Difference* approximation of $J(\mathbf{x}) \mathbf{v}$,

$$J(\mathbf{x})\mathbf{v} \approx \frac{F(\mathbf{x} + \epsilon \mathbf{v}) - F(\mathbf{x} - \epsilon \mathbf{v})}{2\epsilon} \quad (2.18)$$

Both Forward Difference and Backward Difference have an order of accuracy of $O(\epsilon)$, on the other hand, Central Difference has an order of second order or $O(\epsilon^2)$. The Algorithm 4 describes equation (2.18).

Algorithm 4 Algorithm of approximation of $J(\mathbf{x}) \mathbf{v}$ using Central Difference

Require: A function F , a step size $\epsilon > 0$, a vector $\mathbf{x} \in \mathbb{R}^n$, a vector $\mathbf{v} \in \mathbb{R}^n$.

Ensure: An approximation of Jacobian-vector product $J(\mathbf{x}) \mathbf{v}$.

$\mathbf{u} + \epsilon \mathbf{v}$

return $(F(\mathbf{x} + \mathbf{u}) - F(\mathbf{x} - \mathbf{u})) / (2\epsilon)$

The use of this approximations Jacobian-vector product using a Krylov subspace method to solve (2.14), e.g. GMREs, is called *Jacobian-Free Newton-Krylov method* (JFNK)[29].

We note that the classic JFNK method does not explicitly a method for computing the Jacobian- transposed-vector product so we cannot use it to solve (2.8). So we refer to a modified JFNK algorithm Sanhueza and Torres 2017 [30] where it proposes two strategies to approximate it. In the first approach, it calculates exactly the product via a linear combination of a Q orthogonal basis,

$$J^T(\mathbf{x})\mathbf{v} = \sum_{i=1}^n \alpha_i \mathbf{q}_i$$

Given that Q are orthonormal we have that

$$\begin{aligned} \mathbf{q}_k^T \cdot J^T(\mathbf{x})\mathbf{v} &= \mathbf{q}_k^T \cdot \left(\sum_{i=1}^n \alpha_i \mathbf{q}_i \right) \\ (J(\mathbf{x})\mathbf{q}_k)^T \cdot \mathbf{v} &= \alpha_k \underbrace{\mathbf{q}_k^T \mathbf{q}_k}_1 = \alpha_k, \end{aligned}$$

so each α_i can be computed

$$\alpha_i = \langle J(\mathbf{x})\mathbf{q}_i, \mathbf{v} \rangle.$$

Finally, the product of $J^T(\mathbf{x})\mathbf{v}$ is given by

$$J^T(\mathbf{x})\mathbf{v} = \sum_{i=1}^n \langle J(\mathbf{x})\mathbf{q}_i, \mathbf{v} \rangle \mathbf{q}_i$$

If we use n vectors we can compute the product exactly, if we use less it is an approximation. Algorithm 5 describes this procedure

Algorithm 5 Algorithm of approximation of $J^T(\mathbf{x}) \mathbf{v}$ via an orthogonal basis.

Require: A function F , a step size $\epsilon > 0$, a vector $\mathbf{x} \in \mathbb{R}^n$, a vector $\mathbf{v} \in \mathbb{R}^n$, $Q \in \mathbb{R}^{n \times k}$.

Ensure: An approximation of Jacobian-transpose-vector product $J^T(\mathbf{x}) \mathbf{v}$.

```

 $\mathbf{p} = \mathbf{0}_n$ 
for  $i = 0$  to  $k$  do
     $\mathbf{u} = J(\mathbf{x}_i)\mathbf{q}_1$  using Algorithm 4
end for
return  $\mathbf{p}$ 

```

The other method is to compute the transpose of the Jacobian matrix that does not require any additional storage, in contrast to the orthonormal basis method. However, this is an approximate method. Note that the function g in this section is not related to 2.5

Consider the function $g(\mathbf{x}) = \langle F(\mathbf{x}), \mathbf{v} \rangle$, where $\mathbf{v}$ is an arbitrary vector. We have,

$$g(\mathbf{x}) = \sum_{i=1}^m f_i(\mathbf{x})v_i,$$

where v_i is the i -th component of the vector $\mathbf{v}$ and $f_i(\mathbf{x})$ is the i -th component of F .

This follows from:

$$\frac{\partial g}{\partial x_k} = \left\langle \frac{\partial \mathbf{F}}{\partial x_k}, \mathbf{v} \right\rangle \quad (2.19)$$

where $\frac{\partial \mathbf{F}}{\partial x_k}$ is the k^{th} column of $J(\mathbf{x})$.

So the gradient $\nabla g(\mathbf{x})$ gives the desired transpose Jacobian-vector product $J^T(\mathbf{x})\mathbf{v}$. The gradient can be approximated component-wise using finite differences, which are parallelizable and matrix-free.

The gradient approximation allows computing an approximation of $J^T(\mathbf{x})\mathbf{v}$ without needing to store any additional basis vectors like the orthonormal basis method. The tradeoff is that the gradient approximation is inexact.

The overall algorithm to compute $J^T(\mathbf{x})\mathbf{v}$ using the gradient approximation is:

1. Define $g(\mathbf{x}) = \langle \mathbf{F}(\mathbf{x}), \mathbf{v} \rangle$
2. Approximate $\nabla g(\mathbf{x})$ using finite differences
3. Set $J^T(\mathbf{x})\mathbf{v} = \nabla g(\mathbf{x})$

This provides a parallelizable matrix-free approach to compute the desired transpose Jacobian-vector product.

Algorithm 6 Algorithm of approximation of $J^T(\mathbf{x}) \mathbf{v}$ via gradient

Require: A function F , a vector $\mathbf{x} \in \mathbb{R}^n$, a vector $\mathbf{v} \in \mathbb{R}^n$, $\epsilon > 0$.

Ensure: An approximation of Jacobian-transpose-vector product $J^T(\mathbf{x}) \mathbf{v}$.

```

 $g(\mathbf{x}) = \langle \mathbf{F}(\mathbf{x}), \mathbf{v} \rangle$ 
 $\nabla g = \mathbf{0}_n$ 
for  $i = 0$  to  $n$  do
     $\nabla g_j = (g(\mathbf{x}_i + \epsilon \mathbf{e}_j) - g(\mathbf{x}_i)) / \epsilon$    Compute j-th partial derivative of  $g$ 
end for
return  $\nabla g$ 

```

The first approximation has a computational complexity $\mathcal{O}(km)$ where k is the number of vectors used and space complexity of $\mathcal{O}(n)$ if it uses the canonical basis. On the other hand, the second uses $\mathcal{O}(nm)$ for computational and $\mathcal{O}(m)$ for space.

2.5 LSQR

The LSQR [14] method is an iterative algorithm used to solve large, sparse linear systems of equations of the form $Ax = b$. The system may be overdetermined, underdetermined, or inconsistent, and A is an $m \times n$ matrix, not necessarily square, symmetric, positive definite, or full rank. LSQR is also capable of finding the least-squares solution that minimizes the norm of the residual vector $b - Ax$.

The method is an iterative Krylov subspace projection method that builds a sequence of solution subspaces of small but increasing dimensions. Given $C \in \mathbb{R}^{n \times n}$, $d \in \mathbb{R}^n$, a Krylov subspace is defined of size k ,

$$\mathcal{K}_k(C, d) = \text{span}\{d, Cd, C^2d, \dots, C^{(k-1)}d\} \quad (2.20)$$

The key idea behind LSQR is to perform an implicit bidiagonal decomposition of matrix A , similar to a singular value decomposition (SVD). This decomposition involves generating two orthogonal bases that span the Krylov subspaces $\mathcal{K}_k(A^T A, A^T b)$ and $\mathcal{K}_k(AA^T, b)$ using the Golub-Kahan bidiagonalization process [31].

During each iteration, LSQR computes the approximate solution x_k by solving a smaller system involving the bidiagonal matrix B_k obtained from the bidiagonalization,

$$\begin{aligned} JV_k &= U_{k+1}B_k \\ J^T U_{k+1} &= V_{k+1}\hat{B}_k \end{aligned}$$

which reduces the problem

$$\min_{\Delta x \in \text{range}(V_k)} \|b - J\Delta x\|^2 = \min_{c_k \in \mathcal{R}^k} \|\|r\|e_1^{(k+1)} - B_k c_k\|^2$$

into a least squares problem of size $(k+1) \times k$.

This solution is refined at each step, and the process continues until a stopping criterion is met, such as reaching a certain tolerance for convergence or a maximum number of iterations.

The main advantages of LSQR are its ability to handle noisy or inconsistent data, its efficient handling of sparse matrices using only matrix-vector multiplications with A and A^T , and the option to use a preconditioner matrix for improved numerical properties and efficiency.

2.6 Non-symmetric LSQR: ns-LSQR

In Sanhueza Roman [32] proposes to solve (2.14) with LSQR, a large-scale sparse linear least squares solver. The LSQR method is mathematically equivalent to the Conjugate Gradient

(CG) method over (2.14). Therefore, it is a Krylov subspace method for the matrix $J^T J$ and the right-hand side vector $J^T B$.

LSQR method originally solves for

$$\min_{\Delta x \in \mathcal{R}^n} \|\mathbf{b} - J\Delta x\|$$

using the two recurrences at the k -th iteration, based on the Bidiagonalization process by Golub and Kahan

$$\begin{aligned} JV_k &= U_{k+1}B_k \\ J^T U_{k+1} &= V_{k+1}\hat{B}_k \end{aligned}$$

The author proposes to modify LSQR, using a routine to compute the matrix-vector product $J(x)v$ and a routine to *approximate* $J^T(x)v$, e.g. algorithm 5 or 6. This modifies the Bidiagonalization recurrences into

$$\begin{aligned} JV_k &= U_{k+1}\tilde{B}_k \\ \tilde{J}^T U_{k+1} &= V_{k+1}\tilde{B}_k \end{aligned}$$

which reduces the problem in a similar way,

$$\min_{c_k \in \mathcal{R}^k} \|\|r\|\mathbf{e}_1^{(k+1)} - \tilde{B}_k c_k\|^2$$

.

The author also proposes an approximation method of $J^T(x)v$ based on quantization of J , using a lower number of bits trading precision for space and adding some complexity to compute the products because we need to compress and decompress the matrices for its multiplication with an arbitrary vector. Each layer L can have a different number of bits b_i . The computational complexity of this approximations is $\mathcal{O}\left(mn \sum_{i=1}^L b_i\right)$

Algorithm 7 Algorithm of ns-LSQR

Require: A function F , a vector x_i to evaluate the Jacobian matrix, a stepsize $\epsilon > 0$ to be used in the finite-difference approximation, an approximation matrix $\tilde{J}$, a vector $\mathbf{b}$, a tolerance ν , a number of iterations t_{inner} and t_{outer} to control the restarts, an initial guess x_0 which usually is set to be the zero vector

Ensure: An approximation solution Δx_k

```

for  $i$  from 1 to  $t_{outer}$  do
     $r = b - Jx_0$  using finite difference approximation
     $\tilde{\mathbf{u}}_1 = r / \|r\|$ 
     $\tilde{\mathbf{v}}_1 = \tilde{J}^T r / \|\tilde{J}^T r\|$ 
    for  $k$  from 1 to  $t_{inner}$  do do
        Compute  $\tilde{\mathbf{u}}_{k+1}$  using a modified Gram Schmidt procedure
        Compute  $\tilde{\mathbf{v}}_{k+1}$  using a modified Gram Schmidt procedure
        Solve the least-square problem  $\min_{\tilde{\mathbf{c}}_k \in \mathcal{R}^k} \|\|r\| \mathbf{e}_1^{(k+1)} - \tilde{B}_k \tilde{\mathbf{c}}_k\|^2$ , for  $\tilde{\mathbf{c}}_k$ .
         $\Delta x_k = \tilde{V} \tilde{\mathbf{c}}_k$ 
        if The solution  $\Delta x_k$  fulfill a convergence criteria using the tolerance  $\nu$  then
            return  $\Delta x_k$ 
        end if
    end for
     $x_0 = x_0 + \Delta x_k$ 
end for
return  $\Delta x_k$ 

```

Chapter 3

Approximations

In this chapter, we delve into various approximations and approaches to approximate a generic matrix. Our discussion follows a step-by-step thought process that the author had in order to propose an approximation solution for the given problem. The main objective of the problem is to find an approximated matrix that occupies less memory than the original while ensuring that the columns of the matrix remain linearly independent.

First, we need to define what we understand of an approximation and, in specific, an approximation of a matrix.

Definition: An approximation can be defined as something that is relatively close in some sense or metric to another.

From this definition, two main questions arise: When and why do we need to approximate? And how can we determine whether an approximation is useful? In scientific computing, this topic is of fundamental importance. When solving mathematical problems or performing computations, we often work with representations and machines that have limitations. These limitations can include finite memory and processing power, which necessitate the need to find efficient ways to work around them. Additionally, measuring the similarity or usefulness of an approximation is crucial as it provides insights into its practical value.

In the context of approximating a generic matrix, our goal is to find an approximation that retains the same dimensions as the original matrix (i.e., the same number of rows and columns), but with values and/or norms that are relatively similar. The approximations we will study in this chapter balance the limitations of memory and the requirement of linearly independent columns with the amount of information retained in the matrix.

3.1 Diagonal matrices

Our first approach involves using a matrix with a defined structure, such as a diagonal matrix, where only the components of the diagonal are stored in a vector. A diagonal matrix is a matrix in which all entries, except those on the diagonal, are zero. In this case, each column has a value in a different row, making it one of the simplest forms of a linearly independent matrix.

Formally the matrix $D = (d_{i,j})$ with n rows and n columns is diagonal if

$$\forall i, j \in \{1, 2, 3, 4, 5, 6, \dots, n\}, i \neq j \rightarrow d_{i,j} = 0$$

Visually, a diagonal matrix can be represented as:

$$D = \begin{bmatrix} \times & 0 & \cdots & \cdots & 0 & 0 \\ 0 & \times & \cdots & \cdots & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & \cdots & \times & 0 \\ 0 & 0 & \cdots & \cdots & 0 & \times \end{bmatrix} \in \mathbb{R}^{n \times n}$$

Although diagonal matrices are typically defined for square matrices, we can extend this notion to rectangular matrices, which are often encountered in large-scale least-square problems where $m \gg n$.

With the original definition, the matrix looks like:

$$D = \begin{bmatrix} \times & 0 & \cdots & \cdots & 0 & 0 \\ 0 & \times & \cdots & \cdots & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ \vdots & \cdots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & \cdots & \times & 0 \\ 0 & 0 & \cdots & \cdots & 0 & \times \\ 0 & 0 & 0 & 0 & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots & \cdots \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \in \mathbb{R}^{n \times m}$$

A extended diagonal matrix $\hat{D} = (\hat{d}_{i,j})$ with n rows and m columns is diagonal if

$$\forall i, j \in \{1, 2, 3, 4, 5, 6, \dots, n\}, i \bmod m \neq j \rightarrow \hat{d}_{i,j} = 0$$

Visually, this can be represented as:

$$\hat{D} = \begin{bmatrix} \times & 0 & \cdots & \cdots & 0 & 0 \\ 0 & \times & \cdots & \cdots & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ \vdots & \cdots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & \cdots & \times & 0 \\ 0 & 0 & \cdots & \cdots & 0 & \times \\ \times & 0 & 0 & 0 & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots & \cdots \\ 0 & 0 & 0 & 0 & 0 & \times \end{bmatrix} \in \mathbb{R}^{n \times n}$$

One advantage of using diagonal matrices is that they can store a few entries of each column in a convenient way, such as a list or a low-dimensional matrix. Additionally, by definition,

each column in a diagonal matrix is linearly independent because each column has different row indices. Therefore, it is not possible to recreate one column using the others.

However, a major disadvantage is that we do not consider the values of each entry, so we may not retain components with valuable information.

Another alternative to consider for this analysis is the use of other fixed structures, such as tri-diagonal matrices. However, in some structures, the linearity of the columns is not guaranteed, and these also share the same disadvantage mentioned earlier.

3.2 Max Components (MC) approach

Instead of using a defined or rigid structure, another approach involves finding the components with the most information in each column and setting the others to zero. We can argue that the entries with the greatest magnitudes carry the most information, but this is not the only possible criterion. For example, we can analyze the variance of each column and choose the most representative components based on that.

In this case, we focus on the absolute values of each column, sort them, and identify the maximum components. This approach prioritizes the values of the entries rather than their positions as a fixed structure would. However, we need to develop an algorithm to ensure the linear independence of the columns.

We can present two alternatives for the Max Components (MC) approach:

- **Exactly One different (EO):** In this approach, we sort the absolute values of the first column and choose the k maximum components. Then, for the next columns, we use the same sorted indices but choose one different component to ensure independent columns.
- **At Least One different (ALO):** In this approach, we sort the absolute values of each column from right to left. We choose k maximum components for each column with the restriction that each row index is used exclusively in one single column.

The ALO Maximum Component approach considers each column separately, which can become problematic when k is close to the number of rows n . In such cases, it may occupy all row indices before reaching the last column, potentially resulting in linearly dependent columns.

The EO Maximum Component approach considers the first column and expands the list of occupied indices for subsequent columns. This solves the problem in the ALO approach and

allows us to choose up to $\hat{k} = n - 1$ components while still maintaining n linearly independent columns. However, a drawback is that we are bound to the first column's maximum value order, which may not retain the most information from the other columns.

The algorithms described above are deterministic. However, we can also explore stochastic variants of the MC approximations:

- **Stochastic Maximum Component:** Instead of choosing the maximum component, we assign a probability for each component based on its absolute value relative to the sum of absolute values of all the components in the column. This allows for similar approximations but with the possibility of choosing lower-ranked components, potentially enabling other columns to occupy the row indices that hold the maximum values in the first column.
- **Random Column Sorting:** In this variant, we randomly sort the columns before applying the maximum component approach. This accounts for the fact that the first columns may not always contain the most information or be the most important ones.

3.3 Transforms

So far, we have discussed choosing and preserving the components with the most information in each column. Another approach is to manipulate the matrix itself before approximating it. We can apply a mathematical transform to convert the generic matrix into an information/energy-dense matrix. In particular, we are interested in orthogonal transforms that allow us to retain the methodologies of choosing entries for approximation, such as ALO and EO explained in the previous section. After approximating, we will have a sparse but information-dense matrix. When necessary, we can use the inverse transform to obtain an approximation of the original matrix. Given a matrix A , the orthogonal transform O , and its inverse O^{-1} , the procedure would be the following:

Algorithm 8 Algorithm for approximation using a transformation.

Require: A matrix $A \in \mathbb{R}^{n \times m}$, an orthogonal transform O and its inverse O^{-1} , a procedure to truncate or quantize a matrix Q .

Ensure: An approximation of the matrix A , $\tilde{A} \in \mathbb{R}^{n \times m}$.

- 1: $T = O(A)$, Transform the matrix
 - 2: $\tilde{T} = Q(T)$, Quantize the matrix, e.g. $Q = \text{ALO}$
 - 3: $\tilde{A} = O^{-1}(\tilde{T})$ Inverse Fourier Transform of quantized matrix
 - 4: **return** $\tilde{A}$
-

This approach is based on compression algorithms with loss, which encode and quantize the input to use less memory. Upon decoding the output, we have an approximated result of the input. One widely used transformation in signal processing and data compression

is the *Discrete Cosine Transform* (DCT), which is a Fourier transform that only uses cosines instead of both cosines and sines. This choice of using only real numbers allows for simpler computation. Another transform that could be used is the wavelet transform.

By applying these transforms, we can compress the information in the matrix and reduce the amount of storage required. When needed, we can recover the original matrix by applying the inverse transform.

Chapter 4

Fourier Approximation of the Jacobian Matrix

The Jacobian matrix is a fundamental tool in optimization, providing information about the local behavior of a non-linear function and it is used in a variety of methods like Newton's method, Gauss-Newton's, Levenberg Marquard, Jacobian-Free Newton-Krylov, LSQR and many others to solve non-linear problems as explained in Chapter 1.1.2. However, computing the Jacobian can be computationally expensive, both in terms of computation time and memory requirements, particularly for large-scale problems.

In this chapter, we will present a novel approach for approximating the Jacobian Matrix using an orthogonal transform, particularly either the Discrete Fourier Transform (DFT) or the Discrete Cosine Transform (DCT), in combination with the max component approach (MC) introduced in the previous chapter. Our goal is to create a memory-efficient matrix that balances precision and computational efficiency. By using these techniques, we hope to reduce the memory requirements of the Jacobian approximation while maintaining reasonable accuracy.

We will provide a detailed description of the proposed method, including how the DFT/DCT and MC approaches are combined, and discuss its properties and advantages. Additionally, we will explore how this new approximation can be used in non-linear large-scale least squares problems and argue its implications.

By the end of this chapter, readers will have a clear understanding of our novel approach to approximating the Jacobian matrix, and how it can be used to efficiently solve memory-intensive optimization problems.

4.1 The Discrete Fourier Transform and the Discrete Cosine Transform

The *Discrete Fourier Transform* (DFT) and the *Discrete Cosine Transform* (DCT) are two commonly used orthogonal transforms in signal processing and numerical analysis. The DFT, in particular, is widely used in many fields, including image processing, audio processing, and data compression, to transform periodic signals from the time domain to the frequency domain. The DCT, on the other hand, is frequently used in image and video compression applications, where it provides an efficient representation of images and videos in terms of a small number of coefficients, and the data is not necessarily periodic.

The DFT is a mathematical transform that converts a finite sequence of equally-spaced samples of a signal from the time domain to the frequency domain. Let $\mathbf{x}$ be a discrete-time signal of length N . The DFT of $\mathbf{x}$ is defined by:

$$\mathbf{y}_k = \sum_{n=1}^N \mathbf{x}_n e^{-\frac{i2\pi}{N} kn}$$

where $\mathbf{y}_k$ is the k th frequency component of the signal in the frequency domain, and $\exp(-2\pi i kn/N)$ is a complex exponential term with frequency k/N . It is important to note that this transform takes a complex input and transforms it into a complex output. This transform will be denoted by the symbol $\mathcal{F}(\mathbf{x})$.

Furthermore, the transform can be also represented by a matrix-vector product in the form of:

$$\mathcal{F}(\mathbf{x}) = F\mathbf{x}$$

where

$$F = \begin{bmatrix} \omega_N^{0,0} & \omega_N^{0,1} & \dots & \omega_N^{0,(N-1)} \\ \omega_N^{1,0} & \omega_N^{1,1} & \dots & \omega_N^{1,(N-1)} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_N^{(N-1),0} & \omega_N^{(N-1),1} & \dots & \omega_N^{(N-1),(N-1)} \end{bmatrix}$$

and $\omega_N = e^{-i2\pi/N}$.

The *Inverse Discrete Fourier Transform* (IDFT) corresponds to the transform that reverts a finite sequence of equally-spaced samples of a signal from the frequency domain to the time domain, it is defined by:

$$\mathbf{x}_n = \frac{1}{N} \sum_{k=1}^N \mathbf{y}_k e^{i\frac{2\pi}{N} kn}$$

Note that is very similar to the DFT but we have a scaling factor $1/N$ and the signal of the complex exponential term is changed. Its matrix form is,

$$\mathcal{F}^{-1}(\mathbf{y}) = \frac{1}{N} F^* \mathbf{x}$$

where F^* corresponds to conjugate transpose of F .

The DFT computation involves a large number of complex multiplications and additions, which can be computationally expensive for long sequences. However, the Fast Fourier Transform (FFT) algorithm provides an efficient way to compute the DFT by reducing the number of operations required from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log N)$, where N is the length of the sequence. This makes the DFT and its inverse computationally feasible for large-scale problems and has made it a popular tool for signal processing and data analysis.

In the other hand the DCT is closely related to the DFT but uses only real numbers, which implies that both the input and the output are real. Let $\mathbf{x}$ be a discrete-time signal of length N .

The DCT of $\mathbf{x}$ is defined by:

$$\hat{\mathbf{y}}_k = \sum_{n=0}^{N-1} \mathbf{x}_n \cos \left[\frac{\pi}{N} \left(n + \frac{1}{2} \right) k \right]$$

where $\hat{\mathbf{y}}$ is the column vector of DCT coefficients, $\mathbf{x}$ is the column vector of the original signal.

Note that the equation provided is the formula for the Type-II DCT. There are other types of DCT that have different scaling factors or different cosine functions, but the DCT-II is the most commonly used type.

This transform will be denoted by the symbol $\mathfrak{F}(\mathbf{x})$ and its matrix form of the DCT is:

$$\mathfrak{F}(\mathbf{x}) = \mathbf{A}\mathbf{x}$$

where $\mathbf{A}$ is the DCT matrix defined as:

$$A_{k,n} = \sqrt{\frac{2}{N}} \cos \left[\frac{\pi}{N} \left(n + \frac{1}{2} \right) k \right], \quad k, n = 0, 1, \dots, N-1$$

The *Inverse Discrete Cosine Transform* (IDCT) is used to reconstruct a signal from its DCT coefficients. Let $\hat{\mathbf{y}}_k$ be the k th DCT coefficient of a signal $\mathbf{x}$, then the inverse DCT of the coefficients yields the original signal:

$$\mathbf{x}_n = \frac{2}{N} \sum_{k=0}^{N-1} \hat{\mathbf{y}}_k \cos \left[\frac{\pi}{N} \left(n + \frac{1}{2} \right) k \right]$$

The IDCT can also be expressed as a matrix multiplication using the DCT matrix and its inverse, where $\mathbf{y}$ is the column vector of DCT coefficients:

$$\mathfrak{F}^{-1}(\hat{\mathbf{y}}) = \frac{2}{N} \mathbf{A}^{-1} \mathbf{y}$$

4.2 The approximation

The DCT and DFT are mathematical techniques that can be used to transform a signal from the time domain into the frequency domain. These transforms can be used for signal compression or approximation by truncating the coefficients of the transformed signal.

To achieve signal compression, we first transform the signal into the frequency domain using the DCT or DFT. This yields a transformed signal that can be represented as a linear combination of cosine or complex exponential functions, respectively, with different amplitudes for each frequency component.

Next, we can quantize or truncate the transformed signal by zeroing out the amplitudes of the less important frequency components. By discarding these components, we obtain a lower-dimensional representation of the signal that can be stored more efficiently in a sparse vector.

The degree of truncation or quantization of the coefficients will determine the quality of the resulting approximation. A higher degree of truncation will result in a lower-quality approximation, while a lower degree of truncation will result in a higher-quality approximation.

Finally, we can transform the truncated transformed signal back to the time domain using the IDCT or IDFT. This produces an approximation of the original signal that only retains the lower frequency components that were not truncated. The resulting approximation will have a lower fidelity than the original signal, but it can still be useful in applications where a lower-dimensional representation is sufficient.

Overall, signal compression or approximation using the DCT or DFT can be beneficial in situations where storage space or bandwidth is limited. However, the degree of truncation or quantization should be carefully chosen to balance the trade-off between the quality of the approximation and the amount of compression achieved.

As explained in Chapter 2 we can estimate the matrix- vector product of the Jacobian times and arbitrary vector using *Finite Differences*. If we use the canonical vector $\mathbf{e}_i$ in the matrix-vector product we can estimate the column $\mathbf{J}_i$ of the Jacobian matrix.

So, given a Jacobian matrix $\mathbf{J}$ as a column matrix,

$$\mathbf{J} = \begin{bmatrix} \mathbf{J}_1 & | & \mathbf{J}_2 & | & \cdots & | & \mathbf{J}_n \end{bmatrix}$$

where each $\mathbf{J}_i \in \mathbb{R}^m$ is computed using *Finite Differences*. We apply a convenient mathematical transform to each column such as the Discrete Fourier Transform or Discrete Cosine Transform. We denote the transformed column as $\hat{\mathbf{J}}_i \in \mathbb{R}^m$ and the FFT matrix will be used as an example but the procedure is analogous if the DCT matrix is used.

$$\bar{\mathbf{J}}_i = \mathcal{F} \mathbf{J}_i$$

Then we truncate or assign zeros to the non-dominant frequencies, the notation $()_T$ will denote the truncated vector,

$$\hat{\mathbf{J}}_i = (\bar{\mathbf{J}}_i)_T$$

and apply the inverse transform in order to get an approximation of the original vector $\tilde{\mathbf{J}}_i$.

$$\tilde{\mathbf{J}}_i = \mathcal{F}^{-1} \hat{\mathbf{J}}_i$$

Finally, we apply this procedure to the whole matrix, we get the inverse transform of a sparse

matrix that can be stored efficiently and can be transformed when needed.

$$\begin{aligned}
 \tilde{J} &= \left[\tilde{J}_1 \mid \tilde{J}_2 \mid \cdots \mid \tilde{J}_n \right] \\
 &= \left[\mathcal{F}^{-1} \hat{J}_1 \mid \mathcal{F}^{-1} \hat{J}_2 \mid \cdots \mid \mathcal{F}^{-1} \hat{J}_n \right] \\
 &= \mathcal{F}^{-1} \left[\hat{J}_1 \mid \hat{J}_2 \mid \cdots \mid \hat{J}_n \right] \\
 &= \mathcal{F}^{-1} \underbrace{\hat{J}}_{\text{sparse}}
 \end{aligned}$$

This approximation gives a method to compute the Jacobian-vector product

$$\begin{aligned}
 J \mathbf{w} &\approx \tilde{J} \mathbf{w} \\
 &\approx \mathcal{F}^{-1} \hat{J} \mathbf{w}
 \end{aligned}$$

and the product Jacobian transposed with a vector

$$\begin{aligned}
 J^T \mathbf{w} &\approx \tilde{J}^T \mathbf{w} \\
 &\approx \hat{J}^T \left((\mathcal{F}^{-1})^T \mathbf{w} \right)
 \end{aligned}$$

.

We remark that without the truncation of the columns we have an exact result of the Jacobian-vector product and Jacobian-transpose-vector product.

4.3 Jacobian approximation in large scale least squares

As discussed in Chapter 2 the *Gauss-Newton* and *Levenberg-Marquardt* methods are popular algorithms for solving non-linear over-determined problems, such as those that arise in data fitting or optimization.

However, these methods require the computation and storage of the Jacobian matrix and its transpose, which can be computationally expensive and memory-intensive. In particular, for large-scale problems with many parameters and data points, the Jacobian matrix can become too large to store in memory, making the Gauss-Newton and Levenberg-Marquardt methods impractical.

To address this issue, there are several approaches that can be used to reduce the memory requirements of the Jacobian matrix. One common approach is to use an approximation of the Jacobian matrix like sparse matrix techniques, which exploit the sparsity structure of the Jacobian matrix to reduce the number of non-zero elements that need to be stored and computed. This variant is called the *Inexact Gauss-Newton* method or *Quasi-Newton Levenberg Marquardt* respectively.

Another approach is to use algorithms to compute the matrix-vector product without explicitly storing the Jacobian Matrix, the LSQR method.

Recall that LSQR requires two routines:

- A procedure to compute $J \mathbf{v}$, where $\mathbf{v} \in \mathbb{R}^n$ is an arbitrary vector.
- A procedure to compute $J^T \mathbf{w}$, where $\mathbf{w} \in \mathbb{R}^m$ is an arbitrary vector.

In [32] presented the *non-symmetric LSQR* (ns-LSQR) method where it modifies the requirements to:

- A procedure to compute $J \mathbf{v}$, where $\mathbf{v} \in \mathbb{R}^n$ is an arbitrary vector.
- A procedure to *approximate* $J^T \mathbf{w}$, where $\mathbf{w} \in \mathbb{R}^m$ is an arbitrary vector.

As described in Sanhueza [32] we have three cases where the iteration in ns-LSQR can breakdown:

- When the solution of the linear system $\tilde{J}^T J \Delta \mathbf{x} = \tilde{J}^T \mathbf{b}$ is found, but it may not necessarily be equal to the solution that minimizes $\min_{\Delta \mathbf{x} \in \mathbb{R}^n} \|\mathbf{b} - J \Delta \mathbf{x}\|^2$ since the linear system solved is not the normal equation of the least-square problem.
- When the matrix $\tilde{J}^T J$ is singular.
- When $\beta_{k+1,k} = 0$. However, in this case, it means that the solution of the least-square has been found, when the least-square is a zero residual problem.

If one of these breakdowns happens, the method cannot iterate anymore.

Furthermore, the use of the regularizer $\lambda > 0$ modifies equation as follows,

$$\left(\tilde{J}^T J + \lambda I \right) \Delta \mathbf{x} = \tilde{J}^T \mathbf{b}$$

which is the equivalent version of the damped normal equations. For the second breakdown case, a simple and effective way to handle the possible singularity of $\tilde{J}^T J + \lambda I$ is to modify λ shifting the eigenvalues of the matrix.

Note that when the exact value of J^T is used, the non-singularity is ensured just by adding $\lambda > 0$ since $J^T J$ will be symmetric and positive definite.

Chapter 5

Proposed Algorithms

In this section, we present algorithms for approximating a vector, the Jacobian matrix, and the Jacobian-Transpose-vector product for large-scale least squares problems.

Algorithm 9 shows how to approximate a vector using the Fourier transform. Steps 1 and 2 can be changed for another mathematical transform like the *Discrete Cosine Transform* or the *Wavelet Transform*.

Algorithm 9 Algorithm of the Fourier approximation of a vector.

Require: The Fourier Transform F , a vector $v \in \mathbb{R}^n$, a number of components k .

Ensure: An approximation of the vector using Fourier Transform $\tilde{v} \in \mathbb{R}^n$.

- 1: $\hat{v} = F v$, Fourier Transform of the vector
 - 2: $\hat{v} = (\hat{v})_T$, Keep the maximum k components
 - 3: $\tilde{v} = F^{-1} \hat{v}$ Inverse Fourier Transform of truncated vector
 - 4: **return** $\tilde{v}$
-

Let e_k^n be the n th canonical of dimension k , as the vector with a one in the i -th position and zero at all other positions. The Fourier approximation of the Jacobian Matrix for large-scale least squares is computed as follows,

Algorithm 10 Algorithm of the Fourier approximation the Jacobian Matrix

Require: The Fourier Transform F , a vector $x \in \mathbb{R}^n$, $\epsilon > 0$, a number of components per column k .

Ensure: A sparse representation of Jacobian matrix using Fourier Transform $\hat{J}(x)$.

- 1: Set $\hat{J}$ to the zero matrix $\in \mathbb{R}^{m \times n}$
 - 2: **for** $i = 0$ to n **do**
 - 3: $J_i = J(x) e_k^n$ using Algorithm 4
 - 4: $\hat{J}_i = F J_i$, Fourier Transform of the i -th column
 - 5: $\hat{J}_i = (\hat{J}_i)_T$, Keep the maximum k components
 - 6: **end for**
 - 7: **return** $\hat{J}$
-

Some implementation details,

- As it is the algorithm is parallelizable.
- To efficiently store the Fourier approximated matrix the components of $\hat{J}$ are stored in a *sparse matrix* and transformed when the approximation $\tilde{J}$ is needed.

- Step 4 can be switched to another transform like the *Discrete Cosine Transform* or the *Wavelet Transform*.

The third algorithm proposed is the Jacobian-Transpose-vector product discussed in section 3.3 is described in Algorithm 11

Algorithm 11 Algorithm to approximate $J^T w$ using Fourier Transforms

Require: The Fourier Transform F , the sparse matrix $\hat{J}$, a vector $v \in \mathbb{R}^n$.

Ensure: An approximation of $J^T w$.

- 1: $\hat{v} = (F^{-1})^T w$, Fourier Transform
 - 2: $p = \hat{J} \hat{v}$
 - 3: **return** $\text{Re}(p)$
-

where $\text{Re}(\cdot)$ represents the real component of a complex number, applied element-wise.

Some implementation details,

- When working exclusively with real vectors, using the FFT and IFFT involves utilizing imaginary components. However, in this scenario, we truncate the imaginary part of the transformed vector, which can introduce more errors in the approximation.
- Step 1 can be implemented efficiently, for example, if we use the DCT $(\mathfrak{F}^{-1})^T$ is equal to the transform $\mathfrak{F}$ and $(F^{-1})^T$ is equal to the transform F^{-1}

Note if the Algorithm 11 does not quantize the transformed column computes **exactly** the product $J^T v$ and when a matrix with linearly independent columns is needed in step 5 of the Algorithm 10 and Algorithm 11 can be obtained using ALO or EO explained in section 3. However, this approach makes the algorithm no longer parallelizable because the chosen components depend on the other columns.

5.1 Memory and Computational Complexity

We will now analyze the theoretical computational and memory complexities of the proposed algorithms:

Algorithm 9:

- **Computational Complexity:** The Discrete Fourier Transform (DFT) computed via the Fast Fourier Transform (FFT) can cost $\mathcal{O}(2n \log(n))$. Sorting the vector with a comparison-based algorithm can cost $\mathcal{O}(n \log(n))$ on average. Therefore, the total computational complexity is $\mathcal{O}(n \log(n))$.

- **Memory Complexity:** The memory complexity of Algorithm 9 depends on the data structure used. If we assume that we use arrays to store non-zero values, the memory complexity will be $\mathcal{O}(k \text{sizeof}(\text{double}) + k \text{sizeof}(\text{int}))$ to store the transformed vector and its indices.

Algorithm 10:

- **Computational Complexity:** This algorithm involves n transformations of vectors of size m . Using the FFT can cost $\mathcal{O}(n m \log(m))$. Sorting each transformed column to keep the maximum k components adds an additional $\mathcal{O}(n m \log(m))$ complexity. The total computational complexity is $\mathcal{O}(n m \log(m))$.
- **Memory Complexity:** The memory complexity of the sparse Jacobian matrix depends on the percentage of k components kept per column. If we assume that we use arrays to store non-zero values, the memory complexity will be $kn (\text{sizeof}(\text{double}) + 2k \text{sizeof}(\text{int}))$.

Algorithm 11:

- **Computational Complexity:** This algorithm uses the result of Algorithm 10, which has a computational complexity of $\mathcal{O}(n m \log(m))$. In this algorithm, a single transformation costs $\mathcal{O}(m \log(m))$. Therefore, the total computational complexity is $\mathcal{O}((m + 1) n \log(m))$.
- **Memory Complexity:** The memory complexity of Algorithm 11 will be similar to Algorithm 10, depending on the percentage of k components kept per column.

Overall, the memory complexity of the sparse Jacobian matrix is primarily bound to the number of non-zero components kept per column. As the number of non-zero components increases, the memory requirement will grow, affecting the overall performance and scalability of the algorithms, especially when dealing with large datasets.

Chapter 6

Numerical Experiments

In this section, we present and analyze the results of numerical experiments conducted to validate Algorithm (10) and Algorithm (11). Firstly, we provide an overview of the functions under study and their corresponding Jacobians. Secondly, we compare the use of the Fast Fourier Transform (FFT) and Discrete Cosine Transform (DCT) in the Fourier approximation of vectors. Thirdly, we assess the accuracy of the Jacobian Fourier approximation by varying the percentage of components per column and examining the singular values of the approximation and the product $\tilde{J}^T J$. Finally, we evaluate the performance of Algorithm (11) in solving non-linear least squares problems using ns-LSQR.

The experiments were computed in Julia ¹, and the FFT and DCT transformations used are from the FFTW library ². If it is not specified the vector used to evaluate the Jacobian matrix is normally distributed with mean 0 and standard deviation 1.

6.1 Selected Functions and Their Jacobians

In the numerical experiments, a diverse set of functions was selected to encompass various characteristics and structures. These functions were chosen based on their linearity or non-linearity, density, or sparsity, as well as whether the associated problems were square or over-determined. The selection criteria for these functions were influenced by the work of More (1989) [33] and Sanhueza Roman (2021) [32].

The functions are the following:

- *Random Linear Function*: This function involves n unknowns and m equations. It utilizes a matrix A of size $m \times n$, where the entries a_{ij} are real coefficients following a normal distribution with mean 0 and standard deviation 1. The linear function is defined as:

$$F(x) = A x$$

- *Extended Powell Singular Function*: This function consists of n unknowns and n equations, where $n = 4k, k \in \mathbb{N}$. For $i = 1 : k$, the nonlinear functions are defined

¹<https://julialang.org>

²<https://github.com/JuliaMath/FFTW.jl>

as

$$\begin{aligned} f_{4i-3}(\mathbf{x}) &= \mathbf{x}_{4i-3} + 10\mathbf{x}_{4i-2} \\ f_{4i-2}(\mathbf{x}) &= \sqrt{5}(\mathbf{x}_{4i-1} - \mathbf{x}_{4i}) \\ f_{4i-1}(\mathbf{x}) &= (\mathbf{x}_{4i-2} - 2\mathbf{x}_{4i-1})^2 \\ f_{4i}(\mathbf{x}) &= \sqrt{10}(\mathbf{x}_{4i-3} - \mathbf{x}_{4i})^2 \end{aligned}$$

- *Trigonometric Function*: This function involves n unknowns and n equations. For each $i = 1 : n$, the function is given by:

$$f_i(\mathbf{x}) = n - \sum_{j=1}^n \cos(\mathbf{x}_j) + i(1 - \cos(\mathbf{x}_i)) - \sin(\mathbf{x}_i)$$

- *Penalty Function II*: This function has n unknowns and $m = 2n$ equations. The definition of each function is given by,

$$\begin{aligned} f_1(\mathbf{x}) &= \mathbf{x}_1 - 0.2 \\ f_i(\mathbf{x}) &= \sqrt{a}(\exp(\mathbf{x}_i/10) + \exp(\mathbf{x}_{i-1}/10) - \mathbf{y}_i), 2 \leq i \leq n \\ f_l(\mathbf{x}) &= \sqrt{a}(\exp(\mathbf{x}_{l-n+1}/10) - \exp(-1/10)), n < l < 2n \\ f_{2n}(\mathbf{x}) &= \sum_{j=1}^n (n - j + 1)\mathbf{x}_j^2 - 1 \end{aligned}$$

where $a = 10^{-6}$ and $\mathbf{y}_i = \exp(i/10) + \exp((i-1)/10)$

- *Logarithmic Function*: It has n unknowns and m equations with $m \geq n$ defined by, for $i = 1 : m$

$$f_i(\mathbf{x}) = \mathbf{x}_{k+1}^{p+1} \log \left(\sum_{j=1}^n \mathbf{x}_j^2 + 1 \right) + \mathbf{x}_{k+1}$$

where $p = (i-1) \% n$ and $k = (i-1) - np$.

6.1.1 Explicit Jacobian of Test Problems

Although the premise of the problem is that we do not always have access to the definition of $F(\mathbf{x})$ and we obtained the Jacobian by means of a finite difference approximation (4), for the purpose of this document its important to know what kind of matrices are we actually approximating:

- *Random Linear Function*: The Jacobian correspond to the matrix A , a dense matrix.

- *Extended Powell Singular Function*: The Jacobian of this function looks like,

$$J_{eps} = \begin{bmatrix} A(1) & 0 & \dots & 0 \\ 0 & A(2) & \ddots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \dots & A(N) \end{bmatrix}$$

where $J \in \mathcal{R}^{n \times n}$ and $A \in \mathcal{R}^{4 \times 4}$ with $N = n \bmod 4$

$$A(i) = \begin{bmatrix} 1 & 10 & 0 & 0 \\ 0 & 0 & \sqrt{5} & -\sqrt{5} \\ 0 & 2(x_{4i-2}-2x_{4i-1}) & 4(x_{4i-2}-2x_{4i-1}) & 0 \\ 2\sqrt{10}(x_{4i-3}-x_{4i}) & 0 & 0 & -2\sqrt{10}(x_{4i-3}-x_{4i}) \end{bmatrix}$$

We notice that the columns of the matrix do not scale with the number of components, the column always has two components. This Jacobian is *Sparse*.

- *Trigonometric Function*. This Jacobian matrix is defined by,

$$J_{trig} = \begin{bmatrix} \sin x_1 + 1 & \sin x_1 - \cos x_1 & \sin x_2 & \dots & \sin x_n \\ \sin x_1 & \sin x_2 + 2 & \sin x_2 - \cos x_2 & \dots & \sin x_n \\ \sin x_1 & \sin x_2 & \sin x_2 & \dots & \sin x_n \\ \sin x_1 & \sin x_2 & \sin x_2 & \dots & \sin x_n \\ \vdots & \vdots & \vdots & \dots & \vdots \\ \sin x_1 & \sin x_2 & \dots & \sin x_n + n & \sin x_n - \cos x_n \end{bmatrix}$$

In this matrix the diagonal is the only different component on each column, furthermore, it is a dense matrix that scales with n .

- *Penalty Function II*: The definition of the Jacobian is given by,

$$J_{pII} = \begin{bmatrix} B \\ C \end{bmatrix}$$

where $B, C \in \mathcal{R}^{n \times n}$

$$B = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 & 0 \\ \frac{\sqrt{a}}{10} \exp \frac{x_1}{10} & \frac{\sqrt{a}}{10} \exp \frac{x_2}{10} & 0 & \dots & 0 & 0 \\ 0 & \frac{\sqrt{a}}{10} \exp \frac{x_2}{10} & \frac{\sqrt{a}}{10} \exp \frac{x_3}{10} & \dots & 0 & 0 \\ 0 & 0 & \frac{\sqrt{a}}{10} \exp \frac{x_3}{10} & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \dots & \frac{\sqrt{a}}{10} \exp \frac{x_{n-1}}{10} & 0 \\ 0 & 0 & 0 & \dots & \frac{\sqrt{a}}{10} \exp \frac{x_{n-1}}{10} & \frac{\sqrt{a}}{10} \exp \frac{x_n}{10} \end{bmatrix}$$

$$C = \begin{bmatrix} 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & \frac{\sqrt{a}}{10} \exp \frac{x_2}{10} & 0 & \dots & 0 & 0 \\ 0 & 0 & \frac{\sqrt{a}}{10} \exp \frac{x_3}{10} & \dots & 0 & 0 \\ 0 & 0 & 0 & \dots & \frac{\sqrt{a}}{10} \exp \frac{x_{n-1}}{10} & 0 \\ \vdots & \vdots & \vdots & \dots & 0 & \frac{\sqrt{a}}{10} \exp \frac{x_n}{10} \\ 2(n-1+1)x_1 & 2(n-2+1)x_2 & 2(n-3+1)x_3 & \dots & 2(n-(n-1)+1)x_{n-1} & 2(n-n+1)x_n \end{bmatrix}$$

This matrix is similar to the *Extended Powell Singular Function* in the sense that it doesn't matter the number of components, its columns always have 3 or 4 components.

- *Logarithmic Function*, this Jacobian looks like,

$$J_{log} = \begin{bmatrix} D_1 \\ D_2 \\ \vdots \\ D_{n-1} \\ D'_n \end{bmatrix}$$

$$D_p = \begin{bmatrix} (p-1)x_1^p L + x_1^{p+1}/L + 1 & x_1^{p+1}/L & \dots & x_1^{p+1}/L \\ x_2^{p+1}/L & (p-1)x_2^p L + x_2^{p+1}/L + 1 & \dots & x_2^{p+1}/L \\ \vdots & \vdots & \ddots & \vdots \\ x_n^{p+1}/L & x_n^{p+1}/L & \dots & (p-1)x_n^p L + x_n^{p+1}/L + 1 \end{bmatrix}$$

with $L = \log(\sum_j^n x_j + 1)$, $D_p \in \mathcal{R}^{n \times n}$ and $D'_p \in \mathcal{R}^{(m\%n) \times n}$. As we can see the matrix has different values in each component so it is also a dense matrix that scales with n and m .

6.2 Fourier approximation: The use DFT vs the use DCT

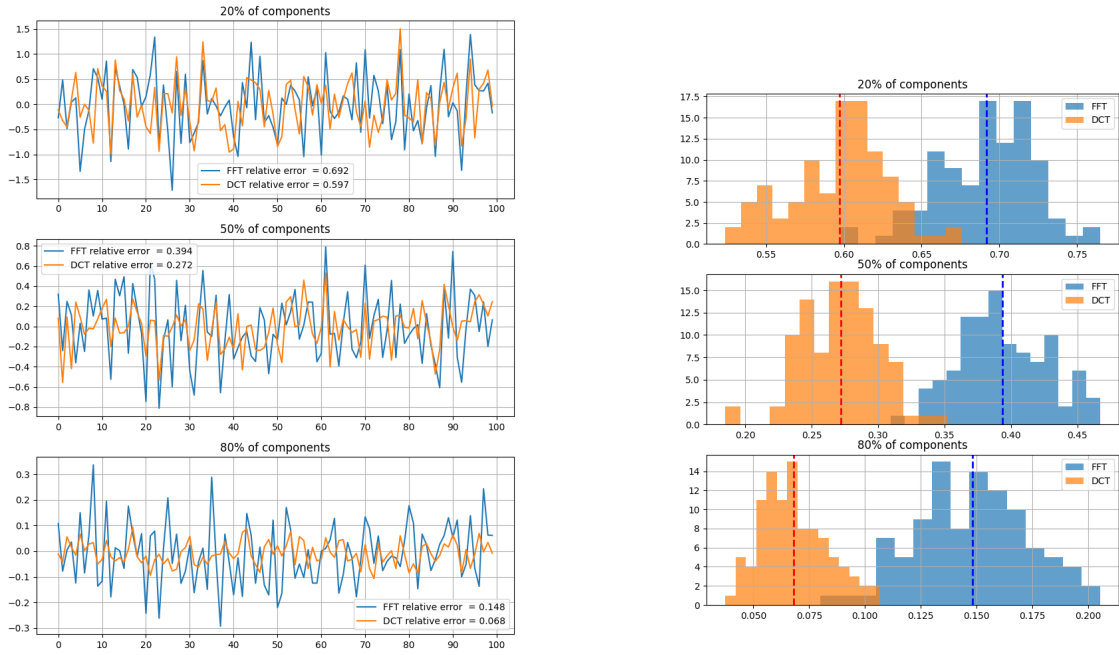
As discussed in Chapter 4.2 and mention in 4.3, our proposed approximation methods can be applied using various mathematical transforms. In this experiment, we specifically investigated the use of two Fourier-related transforms for the approximation of a random vector: the Discrete Fourier Transform (DFT) computed via the Fast Fourier Transform (FFT), and the Discrete Cosine Transform (DCT). We worked with only real numbers so if the transformation has a non-null imaginary component this is truncated.

Figure 2a showcases the reconstruction error of a randomly generated vector of size 100, where we retained 20%, 50%, and 80% of its original components. To evaluate the quality of the approximation, we calculated the relative error using the formula:

$$\frac{\|\mathbf{v} - \tilde{\mathbf{v}}_p\|_2}{\|\mathbf{v}\|_2}$$

Here, $\mathbf{v}$ represents the original random vector, and $\tilde{\mathbf{v}}_p$ denotes the Fourier-approximated vector obtained by considering $p\%$ of its components. Since this experiment involves stochastic elements, we repeated the process 100 times and computed the average. In the case of the FFT which consists of the use of imaginary parts, the final approximation truncates the imaginary part.

The results depicted in the figure indicate that as we increase the number of retained components, the relative error of both the DCT and FFT decreases. However, it is observed that the DCT consistently achieves a lower relative error compared to the FFT. Moreover, as the number of retained components increases, the difference in relative error between the DCT and FFT also becomes more pronounced. This can be attributed to the DCT's effectiveness



(a) Reconstruction error of the Fourier approximation on a random vector. The x-axis corresponds to the index of the components.

(b) Histogram of relative errors of 100 trials. Vertical lines represent the mean value.

Figure 2: Reconstruction errors of the Fourier approximations using the FFT and DCT keeping different percentages of components.

in compressing signals or vectors with a concentrated energy distribution, predominantly in low-frequency components. Moreover, when employing the FFT and IFFT, which involve the use of imaginary numbers, truncating the imaginary parts can introduce errors in the approximation when dealing with real vectors.

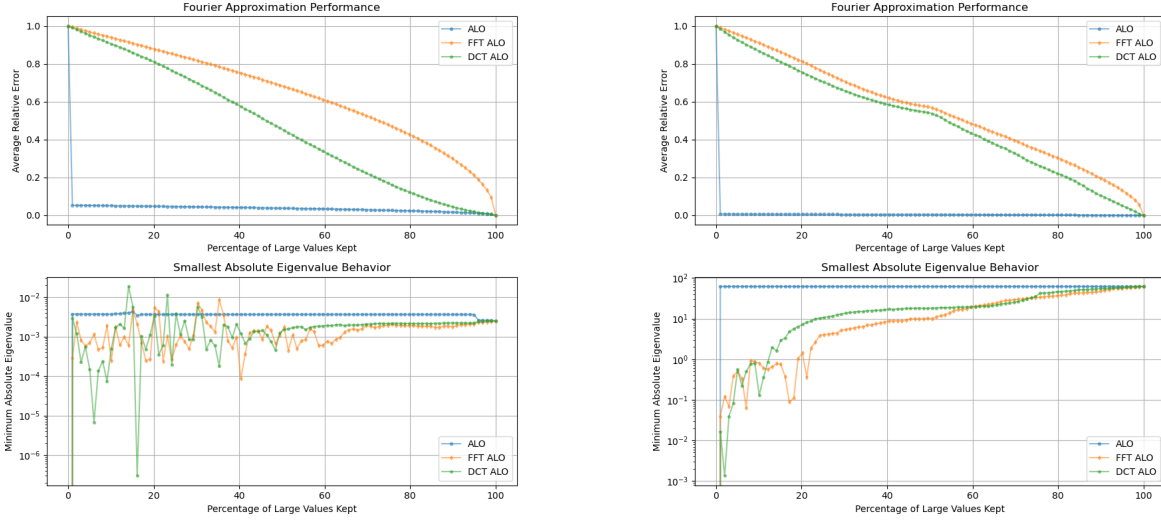
Overall, the figure demonstrates the trade-off between the number of retained components and the resulting approximation error. The DCT exhibits superior compression capabilities compared to the FFT, thereby offering a more efficient approximation method for vectors, at least when its components are normally distributed.

6.3 Accuracy of the Jacobian Fourier Approximation

This experiment aims to evaluate the accuracy of the proposed matrix approximation for the Jacobian of a non-linear problem. We compared three different approximations: the Jacobian Fourier Approximation using the Fast Fourier Transform (FFT) with the At Least One Different (ALO) method to ensure column linear independence when a low percentage of components are retained, the Jacobian Fourier Approximation using the Discrete Cosine Transform (DCT) with the ALO method, and a Jacobian Approximation using only the ALO

method. We selected these approximations to examine the impact of different transforms and the absence of a transform.

The relative error of the Jacobian matrix, denoted as $\frac{\|J - \tilde{J}_p\|_F}{\|J\|_F}$, was used to measure the accuracy of the approximations. Here, J represents the original Jacobian matrix of the function $F(x)$ evaluated at a randomly generated vector x , and $\tilde{J}_p$ is the Fourier-approximated Jacobian obtained by retaining $p\%$ of the components in each column. The Frobenius norm $\|\cdot\|_F$ is used to measure the magnitude of the matrices.



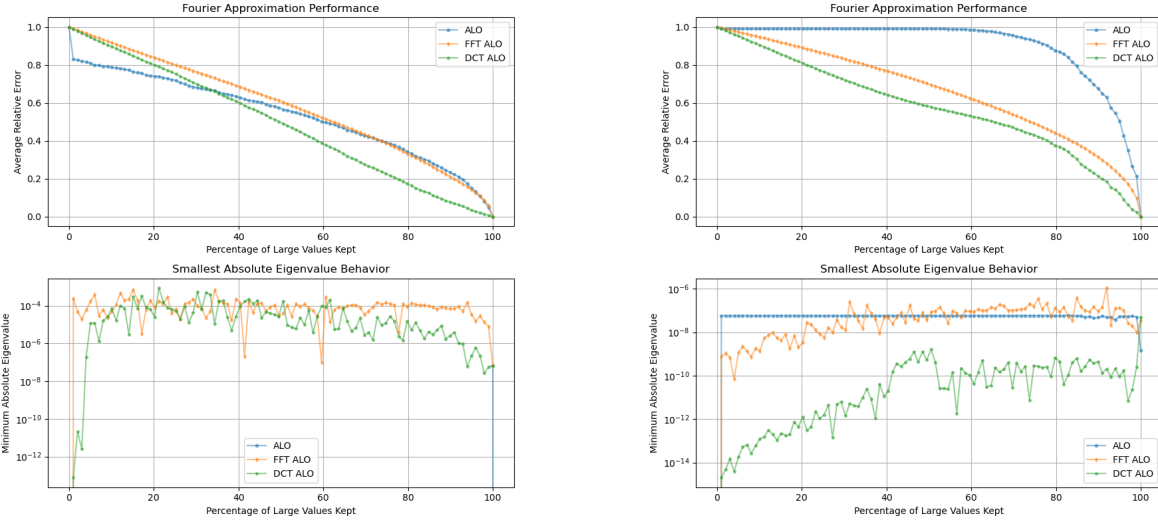
(a) Relative error of the Trigonometric function for different percentages, using $n = 1000$ and $m = 1000$.

(b) Relative error of the Logarithmic function for different percentages, using $n = 1000$ and $m = 2000$.

Figure 3: Relative error of the Jacobian matrix and the minimum eigenvalues of $\tilde{J}^T J$ of the Trigonometric and Logarithmic functions.

In Figure 3, we analyze the relative error of the Jacobian matrix and the behavior of the minimum eigenvalues of $\tilde{J}^T J$ for the Trigonometric and Logarithmic functions. When only the MC approach is used (the ALO curve) shows a significant drop in the relative error when only a small percentage of the largest components is retained in the approximation. At the same percentage, the minimum eigenvalues of $\tilde{J}^T J$ also experience a substantial increase. However, for the Extended Powell and Penalty II functions illustrated in Figure 4, the absence of transformations results in a drastic reduction at a high percentage and a similar performance compared to the cases with transformations respectively. Additionally, the Linear function (Figure 5) exhibits an equal behavior to a transformation (FFT). On the other hand, the use of transformations consistently reduces the relative error of the Jacobian matrix across all functions, with the DCT slightly outperforming the FFT in terms of error reduction.

Furthermore, the minimum eigenvalues of $\tilde{J}^T J$ exhibit a consistent pattern across all functions when transformations are employed. They become consistently non-zero beyond the 20% threshold of values retained in the approximation. This observation suggests that the



(a) Relative error Extended Powell function for different percentages, using $n = 1000$ and $m = 1000$.

(b) Relative error Penalty II function for different percentages, using $n = 1000$ and $m = 2000$.

Figure 4: Relative error of the Jacobian matrix and the minimum eigenvalues of $\tilde{J}^T J$ of the Extended Powell and Penalty II functions.

use of transformations produces more consistent results across multiple functions by steadily reducing the relative error, instead of experiencing drastic reductions at lower or higher percentages. Additionally, the transformations help stabilize the minimum eigenvalues, not without some variability, until reaching the 20% threshold. This behavior ensures a non-singular $\tilde{J}^T J$ and supports the reliability of the Jacobian matrix approximation.

These findings have implications for solving large-scale non-linear least-squares problems using the ns-LSQR algorithm. The proposed Jacobian Fourier Approximation offers a promising algorithm for compressing dense and sparse Jacobian matrix, enabling efficient computations in the product $\tilde{J}\mathbf{w}$. While exact product computation is not required, the singularity of $\tilde{J}^T J$ can lead to breakdowns in the method. However, starting from a 20% threshold, we can ensure the non-singularity property, at least in the five studied functions.

6.4 Accuracy of the Jacobian Fourier Approximation in large-scale non-linear least squares: ns-LSQR

This experiment aims to evaluate the performance of the Jacobian Fourier Approximation in solving large-scale non-linear least squares problems using the ns-LSQR method. Building upon the promising results from previous experiments that demonstrated improved compression, we focus on comparing the approximation using the Discrete Cosine Transform (DCT) with different percentages of components per column.

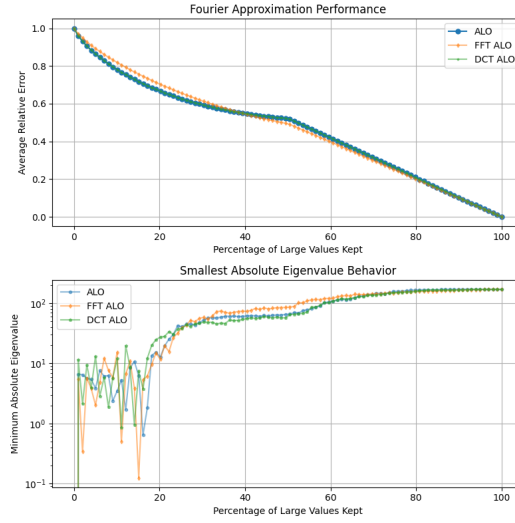


Figure 5: Mean Relative error of the Jacobian matrix of the Linear function for different percentages of values kept in the approximations, with $n = 1000$ and $m = 2000$ of 30 trials. ALO and DCT ALO are on top of each other.

In this experiment, we iterate the ns-LSQR method until the error $\|F(x_i)\|/\|F(x_0)\|$ falls below $1e-12$. To achieve this, the problems are modified to have the form $F(x) = G(x) - b$, where b is a known vector and $G(\cdot)$ represents the function under study. This modification allows us to set the minimization problem to zero. We perform up to 50 restarts and set the step size of the Finite Difference approximation of the Jacobian to $1e-6$. The initial guess is set to $y + 0.001w$ of the solution, where w is a normally distributed vector and y corresponds to the solution of the problem.

By evaluating the accuracy of the Jacobian Fourier Approximation within the context of large-scale non-linear least squares using ns-LSQR, we aim to assess its suitability for practical applications. This analysis provides insights into the performance and potential advantages of the proposed approximation technique in real-world scenarios.

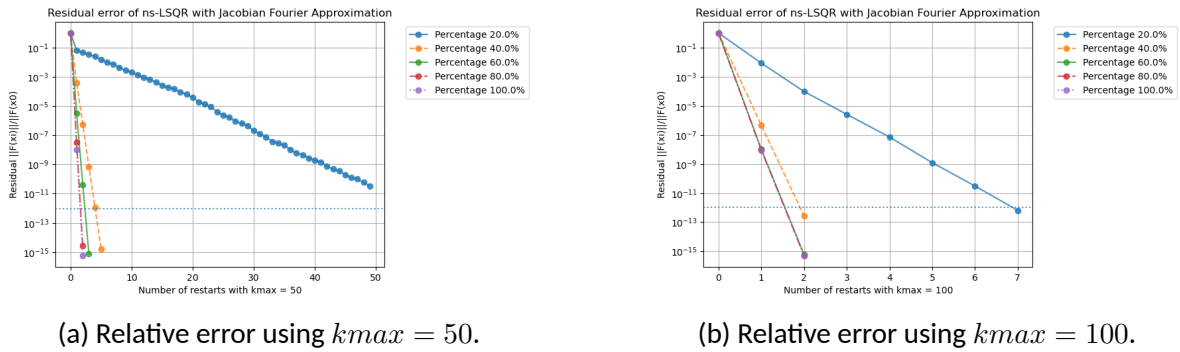
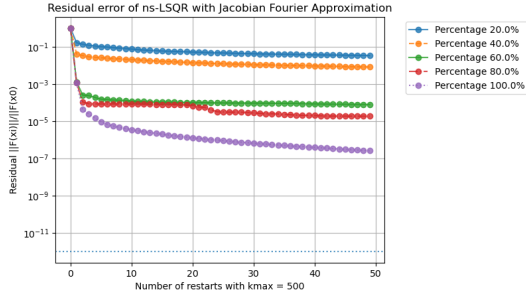
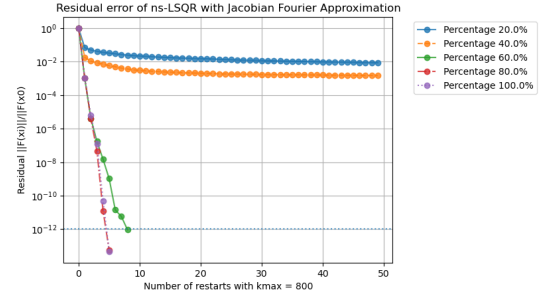


Figure 6: Relative error of the ns-LSQR method for the Random Linear Function achieved at each restart using $n = 1000$ and $m = 2000$.

For the Random Linear Functions (Fig 6), ns-LSQR performs well with a low $k_{\max} = 50$, which corresponds to the dimension of the Krylov subspace used in the iteration. We observe that less than 20 restarts are needed for up to 40% of the components, while more than 50 iterations are required for the remaining 20%. However, the method eventually converges. By increasing the dimension of the Krylov subspace, all percentages converge faster, requiring fewer restarts. For example, when using only a fifth of the components, only 7 restarts are needed.

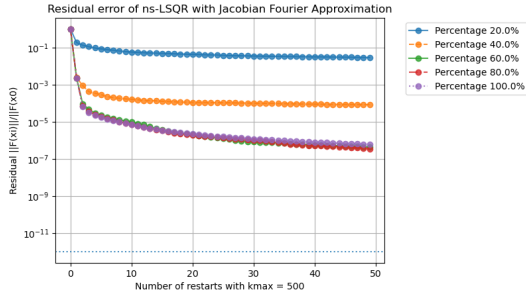


(a) Relative error using $k_{\max} = 500$.

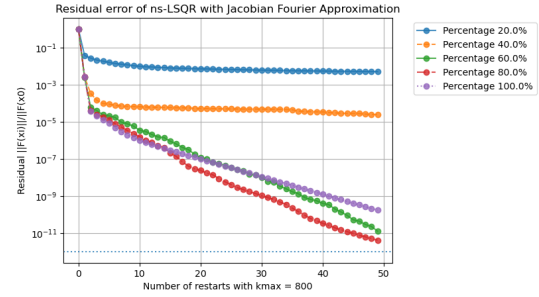


(b) Relative error using $k_{\max} = 800$.

Figure 7: Relative error of the ns-LSQR method for the Extended Powell Function achieved at each restart using $n = 1000$ and $m = 1000$.



(a) Relative error using $k_{\max} = 500$.

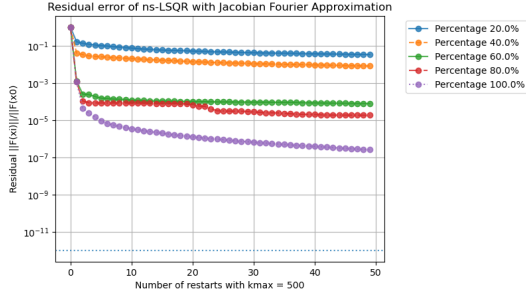


(b) Relative error using $k_{\max} = 800$.

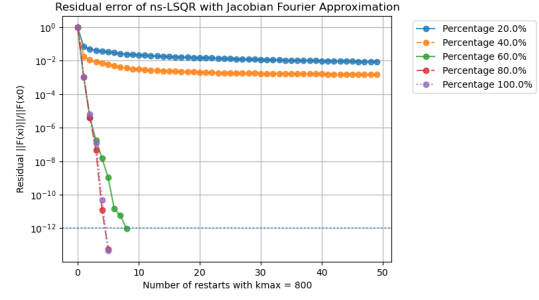
Figure 8: Relative error of the ns-LSQR method for the Trigonometric Function achieved at each restart using $n = 1000$ and $m = 1000$.

On the other hand, Figures 7, 8, and 9 demonstrate slow convergence for all percentages when $k_{\max} = 500$. Although higher percentages lead to slightly faster convergence, the solution is not found within 50 restarts. This suggests that the rate of convergence may not be sufficient to justify further iterations. However, when we increase the Krylov subspace to 800, we observe improved convergence up to 60%, which successfully finds the solution. Below this percentage, it is not worthwhile to continue iterating.

Figure 10 shows better convergence for $k_{\max} = 500$ up to 40%. However, the convergence rate for the last percentage is not significant enough to warrant further iterations. With $k_{\max} = 800$, all percentages are capable of finding the solution.

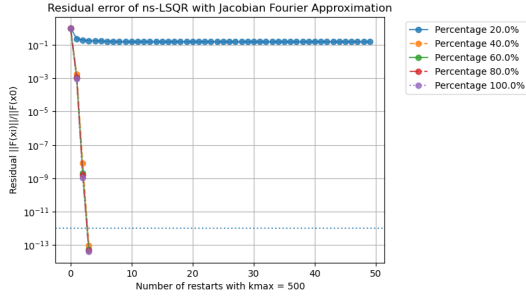


(a) Relative error using $kmax = 500$.

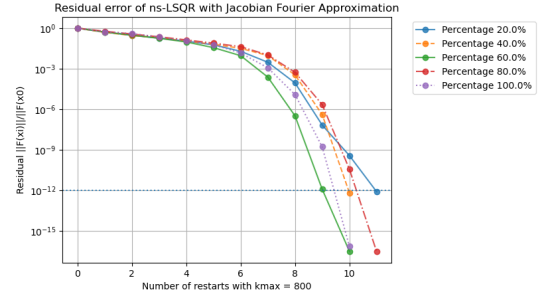


(b) Relative error using $kmax = 800$.

Figure 9: Relative error of the ns-LSQR method for the Penalty Function II achieved at each restart using $n = 1000$ and $m = 2000$.



(a) Relative error using $kmax = 500$.



(b) Relative error using $kmax = 800$.

Figure 10: Relative error of the ns-LSQR method for the Random Linear Function achieved at each restart using $n = 1000$ and $m = 2000$.

It is worth noting that when the entire Krylov space is used, the solution is always found. However, the method loses its efficiency since the goal is to use fewer vectors and consume less memory.

In conclusion, the experiment demonstrates that the Jacobian Fourier Approximation exhibits varying degrees of accuracy in solving large-scale non-linear least squares problems using ns-LSQR. The method performs well for Random Linear Functions with a low dimension of the Krylov subspace, but for other function types, convergence rates differ, and a careful selection of parameters is necessary. This analysis provides valuable insights into the performance characteristics of the approximation technique, aiding in its effective utilization for different problem scenarios.

Chapter 7

Conclusions

In this investigation, our main objective was to develop an efficient method for approximating the Jacobian matrix and its transpose in the context of large-scale non-linear least squares problems. Through the different chapters of this thesis, we have explored the theoretical background, proposed the approximation method, and showed its performance in numerical experiments.

In Chapter 2, we highlighted the significance of the Jacobian matrix and its importance in various applications. We gained insights into the matrix-vector and matrix-transposed-vector products and their computational complexities. The theoretical foundation in Chapter 3 laid the groundwork for our approach to approximating the Jacobian matrix. Chapter 4 presented our proposed approximation method, which utilizes mathematical transforms. Specifically, we suggested the use of the Discrete Cosine Transform (DCT) and Fast Fourier Transform (FFT) techniques, incorporating measures to ensure linear independence. In Chapter 5, we described the algorithms and techniques for efficiently computing the matrix-vector and matrix-transposed-vector products using the proposed approximation. By leveraging the properties of the transforms, we achieved significant computational efficiency gains compared to traditional approaches.

The numerical experiments presented in Chapter 6 provided valuable insights into the performance of the approximation method. We learned that employing these transforms effectively smooths the relative error of the approximations. We observed that the approximation with DCT outperformed the FFT, resulting in better compression and reduced relative errors. Moreover, we displayed that the approximation, regardless of the chosen transform, ensured the non-singularity of the product $\tilde{J}^T J$ for approximation percentages above 20% relevant information when the approximation is used in ns-LSQR.

Regarding the application of the approximation within the ns-LSQR method, we found it to be a viable approach. However, it is important to consider that as the percentage of components used in the approximation decreases, a higher dimension of the Krylov subspace is required to ensure the convergence of the method. This highlights the trade-off between approximation accuracy and computational efficiency.

As for future work, there are several promising avenues to explore. One possibility is to investigate the use of other mathematical transforms, such as the wavelet transform, to further improve the approximation method. Additionally, exploring the combination of multiple transforms per column, as employed in other compression techniques, could potentially enhance the performance of the approximation. Lastly, it would be valuable to assess the effectiveness of the proposed approximation method in other optimization algorithms and compare its performance against alternative approaches.

In conclusion, this document has successfully developed an efficient approximation method for the Jacobian matrix and its transpose in large-scale non-linear least squares problems. The proposed approach utilizing mathematical transforms, particularly the DCT, has demonstrated superior compression and accuracy compared to traditional methods. By providing computational efficiency gains and ensuring non-singularity, our approximation method shows great potential for practical applications in various fields.

REFERENCIAS BIBLIOGRÁFICAS

- [1] Ricardo AS Frantz, J-Ch Loiseau, and J-Ch Robinet. Krylov methods for large-scale dynamical systems: Application in fluid dynamics. *Applied Mechanics Reviews*, 75(3):030802, 2023.
- [2] TJ Clark and Angela D Luis. Nonlinear population dynamics are ubiquitous in animals. *Nature ecology & evolution*, 4(1):75–81, 2020.
- [3] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.
- [4] Yu-Hong Dai, Yakui Huang, and Xin-Wei Liu. A family of spectral gradient methods for optimization. *Computational Optimization and Applications*, 74:43–65, 2019.
- [5] Jorge Nocedal. Conjugate gradient methods and nonlinear optimization. In *Linear and nonlinear conjugate gradient-related methods*, pages 9–23. SIAM, 1996.
- [6] Jorge J Moré and Danny C Sorensen. Newton’s method. Technical report, Argonne National Lab., IL (USA), 1982.
- [7] Jorge Nocedal and Stephen J Wright. Quasi-newton methods. *Numerical optimization*, pages 135–163, 2006.
- [8] John E Dennis Jr and Robert B Schnabel. *Numerical methods for unconstrained optimization and nonlinear equations*. SIAM, 1996.
- [9] Gene H Golub, James M Ortega, et al. *Scientific computing and differential equations: an introduction to numerical methods*. Academic press, 1992.
- [10] Julianne Chung and Silvia Gazzola. Computational methods for large-scale inverse problems: a survey on hybrid projection methods. *Siam Review*, February 2023.
- [11] Dianne P O’Leary and John A Simmons. A bidiagonalization-regularization procedure for large scale discretizations of ill-posed problems. *SIAM Journal on Scientific and Statistical Computing*, 2(4):474–489, 1981.
- [12] Lothar Reichel and Ugochukwu O Ugwu. Tensor krylov subspace methods with an invertible linear transform product applied to image processing. *Applied Numerical Mathematics*, 166:186–207, 2021.
- [13] Martin Hanke. On lanczos based methods for the regularization of discrete ill-posed problems. *BIT numerical mathematics*, 41:1008–1018, 2001.
- [14] Christopher C Paige and Michael A Saunders. Algorithm 583: Lsqr: Sparse linear equations and least squares problems. *ACM Transactions on Mathematical Software (TOMS)*, 8(2):195–209, 1982.

- [15] Andrei N Tikhonov. Solution of incorrectly formulated problems and the regularization method. *Sov Dok*, 4:1035–1038, 1963.
- [16] A Tikhonov. Regularization of incorrectly posed problems. In *Soviet Math. Dokl.*, pages 1624–1627, 1963.
- [17] Julianne Chung, James G Nagy, Dianne P O’leary, et al. A weighted gcv method for lanczos hybrid regularization. *Electronic Transactions on Numerical Analysis*, 28(149-167):2008, 2008.
- [18] Misha E Kilmer and Dianne P O’Leary. Choosing regularization parameters in iterative methods for ill-posed problems. *SIAM Journal on matrix analysis and applications*, 22(4):1204–1221, 2001.
- [19] Silvia Gazzola, Paolo Novati, Maria Rosaria Russo, et al. On krylov projection methods and tikhonov regularization. *Electron. Trans. Numer. Anal*, 44(1):83–123, 2015.
- [20] Sebastian Berisha and James G. Nagy. Chapter 7 - iterative methods for image restoration. In Joel Trussell, Anuj Srivastava, Amit K. Roy-Chowdhury, Ankur Srivastava, Patrick A. Naylor, Rama Chellappa, and Sergios Theodoridis, editors, *Academic Press Library in Signal Processing: Volume 4*, volume 4 of *Academic Press Library in Signal Processing*, pages 193–247. Elsevier, 2014.
- [21] Per Christian Hansen, Jakob Jørgensen, and William RB Lionheart. *Computed tomography: algorithms, insight, and just enough theory*. SIAM, 2021.
- [22] Taewon Cho, Julianne Chung, Scot M Miller, and Arvind K Saibaba. Computationally efficient methods for large-scale atmospheric inverse modeling. *Geoscientific Model Development*, 15(14):5547–5565, 2022.
- [23] Eldad Haber, Uri M Ascher, and Doug Oldenburg. On optimization techniques for solving nonlinear inverse problems. *Inverse problems*, 16(5):1263, 2000.
- [24] 6. *Globally Convergent Modifications of Newton’s Method*, pages 111–154.
- [25] Dong C Liu and Jorge Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical programming*, 45(1-3):503–528, 1989.
- [26] Ron S Dembo, Stanley C Eisenstat, and Trond Steihaug. Inexact newton methods. *SIAM Journal on Numerical analysis*, 19(2):400–408, 1982.
- [27] Donald W. Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *Journal of the Society for Industrial and Applied Mathematics*, 11(2):431–441, 1963.
- [28] Liang Chen. A modified levenberg–marquardt method with line search for nonlinear equations. *Computational Optimization and Applications*, 65(3):753–779, 2016.

- [29] Heng-Bin An, Ju Wen, and Tao Feng. On finite difference approximation of a matrix-vector product in the jacobian-free newton-krylov method. *Journal of Computational and Applied Mathematics*, 236(6):1399–1409, 2011.
- [30] Ariel Sanhueza R. and Claudio E. Torres. A matrix-free algorithm based on Newton's method for overdetermined nonlinear system of equations. In *2017 36th International Conference of the Chilean Computer Science Society (SCCC)*, pages 1–9, Arica, October 2017. IEEE.
- [31] Gene Golub and William Kahan. Calculating the singular values and pseudo-inverse of a matrix. *Journal of the Society for Industrial and Applied Mathematics, Series B: Numerical Analysis*, 2(2):205–224, 1965.
- [32] Ariel Omar Sanhueza Roman. A Limited-memory Levenberg-Marquardt algorithm for solving large-scale nonlinear least-square problems. page 105.
- [33] Jorge J Moré, Burton S Garbow, and Kenneth E Hillstom. Testing unconstrained optimization software. *ACM Transactions on Mathematical Software (TOMS)*, 7(1):17–41, 1981.