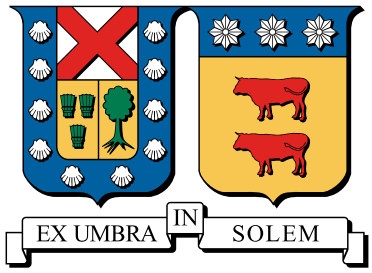
2021-05

A Limited-memory Levenberg-Marquardt algorithm for solving large-scale nonlinear least-square problems

Sanhueza Román, Ariel Omar

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
VALPARAÍSO - CHILE



A Limited-memory Levenberg-Marquardt algorithm for solving large-scale nonlinear least-square problems

Ariel Omar Sanhueza Román

Thesis presented in partial fulfillment of the requirements for the degree of
Magíster en Ciencias de la Ingeniería Informática

Advisor: Claudio E. Torres, Ph.D.
Internal Co-referent: Marcelo Mendoza, Ph.D.
External Co-referent: Timothy Sauer, Ph.D.
Committee Chair: Marcelo Mendoza, Ph.D.

May 28, 2021

THESIS TITLE:

A Limited-memory Levenberg-Marquardt algorithm for solving large-scale nonlinear least-square problems

AUTHOR:

Ariel Omar Sanhueza Román

Thesis presented in partial fulfillment of the requirements for the degree of Magíster en Ciencias de la Ingenieria Informatica of the Universidad Técnica Federico Santa María.

Claudio E. Torres _____
Advisor

Timothy Sauer _____
External Co-referent

Marcelo Mendoza _____
Committee Chair and Internal Co-referent

Valparaíso, Chile.
March, 2021.

Dedication

Este proceso lo cerré en una época difícil para todos. A pesar que todo lo malo que le ocurrió a mucha gente, en distintos aspectos de la vida, uno tiene que saber ver lo positivo dentro de los problemas, y aprovecharlo. En mi caso, la pandemia me permitió enfocarme de forma mucho más eficiente en mi magíster, y creo que gracias a eso pude tardar lo que demoré con el esfuerzo que puse. Y por el mismo contexto, la gente que me acompañó en este proceso, sobre todo al final de este, tomó un rol mucho más relevante.

Quiero agradecer a mi familia que me aguantó durante todo lo que duró este proceso. Mis padres me ayudaron antes y durante la pandemia, y sin su apoyo durante este tiempo me habría sido imposible lograr lo que he logrado. Me apoyaron en muchas cosas, tanto académicas como fuera de ella. Siempre les estaré agradecido por esto (y mucho más). Quiero agradecer también a mis hermanos, que no me ayudaron nada en el magíster, pero sin ellos mi vida habría sido muchísimo más aburrida y la pandemia habría sido mucho más estresante Son los mejores, aunque a veces tengan actitudes que revientan, igual son los mejores para mi. Siempre lo serán.

Obviamente, un agradecimiento infinito a mi polola, Renata, que me ha acompañado durante muchos procesos que he tenido. Ha estado de cerca conmigo en muchos momentos, comenzando desde que estaba en la universidad, terminando mi pregrado, durante mi postgrado y ahora, finalizando este magíster. Fue un apoyo indispensable para mi: me escuchó cuando necesitaba ser escuchado, me aconsejó cuando lo necesité, y siempre me apoyó en todas las decisiones que he tomado, aunque algunas parezcan ridículas o poco cuerdas a ojos ajenos. Siempre estuvo de mi lado. Creo que su apoyo tiene un valor especial, pues se que estamos en sintonía en muchas cosas de la vida, y por lo tanto, se que me entendió cuando necesitaba ayuda. Por esta misma sintonía, que se traduce en empatía y comprensión, es que sus consejos y apoyo han sido relevantes, importantes y acertados. Sin tu apoyo, no se si habría podido dar todo lo que he dado en este proceso. Probablemente habría desistido de muchas cosas que logré, y también, posiblemente habría tomado caminos que ahora no me parecen los más adecuados. Independientemente de lo que depare el futuro para nosotros, que uno no puede asegurar, tú tienes un lugar importante en mi, y eres parte del resultado de lo que soy hoy en día. Eso no cambiará jamás. También gracias por dejarme colarme en tu departamento, aunque parezca chistoso, logré avanzar un montón durante ese tiempo.

Los amigos también tienen parte clave en esto, y en particular, los que estuvieron sufriendo conmigo durante este tiempo. Los cabros siempre ayudan a pasar la época de forma mucho más grata, y creo que sin su compañia durante esta época de estudios, habría sido todo mucho mas difícil, tanto académica como emocionalmente. Yo igual soy catete, medio raro para algunas cosas, pero los cabros siempre apañaron a todo. Con muchos sufrí, reclamé, "lloré" (figurativamente), pero se pasó súper bien. Las reuniones entre clases en el lab, juegos, pizzas, camping (y nuestro fracaso en Navarino Pibarra, inolvidable), etc, son recuerdos que jamás se olvidarán. Estoy seguro que seguirán habiendo más risas y leseos duranto un largo tiempo más (Algún día saldrá ese Catán). Son grandes gente, no cambien nunca.

Quiero agradecer también a Claudio, quién me guió y apoyó en distintos momentos de este proceso. A pesar que mis proyectos a futuro fueron cambiando con el tiempo, me ayudó en distintas maneras y le estaré muy agradecido por todo esto. Además, llevamos trabajando juntos por harto tiempo y espero que luego de este proceso, la buena onda siga allí. Y no dudo que seguirá.

Finalmente, para cerrar esta sección, quiero mencionar que si bien este periodo fue académico, agradezco distintas experiencias e intereses que afloraron durante estos años. Comencé a desarrollar un interés en aprender cosas distintas a mi profesión, y eso es algo que me a aportado muchísimo a mi forma de ver y entender el mundo. Adquirí dos hermosos hobbies, que son el trekking/camping (gracias Chelo) y el interés por la pintura, y tengo el presentimiento que me acompañarán durante el resto de mi vida. Descubrí nuevos intereses por algunas áreas, como las matemáticas, y redescubrí el interés por otras que pensé que ya no me interesaban. Fue sufrido, sí, pero al final todo se acaba y este periodo no fue la expepción, y ahora me llevo un monton de elementos, experiencia y conocimientos que me acompañarán durante largo tiempo. Así son las épocas, tienen un inicio, un desarrollo, y un final. Siempre ha sido así, y siempre lo será.

Cambio y fuera.

Acknowledgements

Powered@NLHPC: This research was partially supported by the supercomputing infrastructure of the NLHPC (ECM-02).

CCTVal: This work was funded by ANID PIA/APOYO AFB180002

Resumen

En esta tesis se propone un *solver* para sistemas de ecuaciones no-lineales sobredeterminados de gran escala, basado en Levenberg-Marquardt. Tradicionalmente, el método de Levenberg-Marquardt requiere la solución de un sistema de ecuaciones lineales que involucra la matriz Jacobiana y su transpuesta. Este sistema de ecuaciones lineales es equivalente a las ecuaciones normales de un problema de mínimos cuadrados lineal. Desafortunadamente, Levenberg-Marquardt no es un método adecuado para problemas de gran escala debido al requerimiento de la matriz Jacobiana. Cuando la dimensión del problema es grande, el cálculo y almacenamiento explícito de la matriz no es posible, debido al alto uso de memoria. La gran mayoría de los algoritmos para problemas de gran escala están diseñados para problemas de optimización sin restricciones, y estos requieren del gradiente de la función objetivo, el cual involucra a la Matriz Jacobiana transpuesta. Si bien existen algoritmos *matrix-free* de bajo costo computacional que aproximan el producto de la matriz Jacobiana por un vector, aproximar el producto de la transpuesta de la matriz Jacobiana por un vector requiere un mayor costo de cómputo. En esta tesis se propone el uso de Levenberg-Marquardt en conjunto con un nuevo algoritmo, basado en LSQR, llamado nsLSQR. El método nsLSQR resuelve un problema de mínimos cuadrados lineal utilizando una aproximación con diferencias finitas y una aproximación, de cualquier índole, de la matriz Jacobiana transpuesta. Para aproximar el producto de la matriz Jacobiana transpuesta por un vector, se propone un algoritmo, basado en cuantización, para aproximar la matriz Jacobiana. Mediante el uso de cuantización, el uso de memoria es reducido significativamente, respecto a una matriz explícitamente almacenada. Combinando la aproximación cuantizada y nsLSQR, se propone un algoritmo, el cual denominamos lm-nsLSQR (Levenberg-Marquardt nsLSQR), que permite minimizar el residuo de un sistema de ecuaciones no-lineales sobredeterminado de gran escala.

Palabras clave: large-scale problems, overdetermined system of equations, nonlinear problems, Levenberg-Marquardt, Krylov subspace, LSQR, Quantization.

Abstract

In thesis a method to solve large-scale overdetermined nonlinear system of equations is proposed. The method is based on the Levenberg-Marquardt method. Traditionally, the Levenberg-Marquardt requires the solution of a linear system of equations that contains the Jacobian matrix and its transpose. This linear system of equation is equivalent to the normal equation of a linear least-square problem involving the Jacobian matrix. Unfortunately, the Levenberg-Marquardt is not a suitable method for large-scale problems due to the requirement of the Jacobian matrix. For large-scale problems, the computation and explicit storage of a matrix is prohibited due to its high memory usage. Some algorithms for large-scale problems have been designed but most of them are aimed for unconstrained optimization problems and they require the gradient of the objective function, which involves the transpose of the Jacobian matrix. There are some computationally cheap matrix-free methods to approximate the product of the Jacobian matrix times a vector. However, to approximation of the transpose of the Jacobian times a vector is not cheap to compute, and matrix-free algorithms usually demands a high computational effort. In this thesis, the Levenberg-Marquardt is used in combination with a novel solver for linear least-square problem, called nsLSQR. The nsLSQR method is based on LSQR, and solves a linear least-square problem using a matrix-free finite difference approximation to compute the product of the Jacobian matrix times a vector, and use any available approximation for the transpose of the Jacobian matrix times a vector. To approximate the product involving the transpose of the Jacobian matrix, in this thesis an approximation method is proposed, based on Quantization. This quantization builds a low-memory approximation of the Jacobian matrix, and the transpose of this approximation is used. The quantized approximation use significantly less memory that a explicitly stored double precision matrix. Combining the quantized approximation and nsLSQR, the proposed algorithm, which will be called lm-nsLSQR (Levenberg-Marquardt nsLSQR), can handle large-scale nonlinear problems.

Keywords: large-scale problems, overdetermined system of equations, nonlinear problems, Levenberg-Marquardt, Krylov subspace, LSQR, Quantization.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

The need for finding solutions of nonlinear problems arise commonly in many fields of study. Some examples where nonlinear problem are used is in modeling natural or physical phenomenon and in Machine Learning or data fitting problems. In the former, the desired behavior that is needed to be modeled is approximated by means of a nonlinear equation. In particular, the usage of differential equations is not rare. Although linear models can be used for simple problems, more complex behavior requires a higher degree or more complex functions Jouha, Oualkadi, Dherbécourt, Joubert, and Masmoudi 2018; P. Hu, Cao, Zhu, and J. Li 2010. When these PDE involve several terms and its degree of non-linearity is high, analytical solutions are not always easy to compute. Some problems even are not know if an analytical solutions exists under given initial conditions. For this reason, numerical solutions is a method to handle, study and visualize the models designed for these phenomenon. In the latter example, a model and its parameters are "trained" or fitted using a dataset or available data. The aim is to produce a model capable of predict future outcomes or events related to the given data. A measure of error is used to optimize and decrease the error produced by the model over the data. Later, the trained model can be used to guess new events or data that has not been seen yet by the model. Although linear models can be used, its application are very restricted and usually nonlinear models are preferred for more difficult problems. Neural Networks is a typical nonlinear model used in Machine Learning problems Pouyanfar, Sadiq, Y. Yan, Tian, Tao, Reyes, Shyu, S.-C. Chen, and Iyengar 2018; Du and Stephanus 2018, and its training process usually involves a nonlinear objective function. The training or fitting of these models demands high computational cost and efficient algorithms are desired. Regardless of the application or field, when a nonlinear problem is tried to be solved by numerical means, the nonlinear problem takes the form of a *nonlinear system of equations*.

Formally, a nonlinear system of equations is expressed in the following form,

$$\mathbf{F}(\mathbf{x}) = \mathbf{0}, \tag{1.1}$$

where $\mathbf{F} : \mathbb{R}^n \to \mathbb{R}^m$ is a map, $\mathbf{x}$ represent the root of the problem and $\mathbf{0}$ is the zero vector in $\mathbb{R}^m$. If $m < n$ the problem is called *Under-determined nonlinear*

system of equations, if $m > n$ the problem is an *Over-determined nonlinear system of equations*. Otherwise is simply called a nonlinear system of equations.

From a variety of classical approaches, we remark at least two to handle these kind of problems,

- The usage of Newton's method and its related algorithms, like Quasi-Newton methods.
- Application of unconstrained optimization methods. The approach is applied, usually, over the residual of the function $\mathbf{F}$.

For each of these approaches, several classical algorithms have been developed along these years Kelley 1999; Nocedal and S. Wright 2000; Chong and Zak 2013; Kelley 1995. Some popular methods are the Newton's method, Quasi-Newton methods, gradient-based methods and trust-region methods. Most of these algorithms are still used to our days, either in their initial form or with modification to improve their performance to general or specific problems.

The focus of this thesis are over-determined problems, which usually appears in nonlinear least square problems, discretization of functional or data fitting of nonlinear models. An approach for solving an over-determined problem is by means of solving it in a least-square sense. This problem is usually solved as an unconstrained optimization problem, where its residual is minimized. This is possible by defining a function g as,

$$g(\mathbf{x}) = \frac{1}{2}\|\mathbf{F}(\mathbf{x})\|_2^2, \tag{1.2}$$

and solve the minimization problem given by,

$$\min_{\mathbf{x}\in\mathbb{R}^n} g(\mathbf{x}). \tag{1.3}$$

The incorporation of $\frac{1}{2}$ in (1.2) is for later convenience in the computation of derivatives.

Among the popular methods for solving such problems Nocedal and S. Wright 2000; Chong and Zak 2013 are *Steepest descent* with line search, *trust-region methods*, *Nonlinear Conjugate gradient method*, Newton and Quasi-Newton methods, *Genetic Algorithms* and some *Derivative-Free methods*. Although all those methods are possible to use over unconstrained optimization problems, not all are suitable for solving (1.3). This is mostly due to that general unconstrained optimization method does not use the fact that g depends on $\mathbf{F}$. For example, most of these methods requires the gradient of g explicitly to compute some vectors or approximate the Hessian matrix. However, note that

$$\nabla g(\mathbf{x}) = J^T(\mathbf{x})\,\mathbf{F}(\mathbf{x}), \tag{1.4}$$

where $J(\mathbf{x})$ is the Jacobian matrix of $\mathbf{F}$. Such gradient is not possible to compute easily without forming the Jacobian matrix explicitly. Another issue, which is the

main focus of this work, is related to large-scale problems. Most of these methods are suitable for small and medium scale problems, whose computational and storage requirements remains under acceptable ranges. Unfortunately, for large-scale problems, some methods have issues to handle it due to high computational requirements or matrices, overwhelming the available memory. As said previously, most of these methods requires the gradient of g which depends on the transpose of the Jacobian matrix of $\mathbf{F}$. Forming an approximation of the Jacobian matrix for small or medium-scale problems is not an issue at all since the available memory in the computational system may be enough for building and storing the matrix explicitly. This is not the case for large-scale problems where storing a matrix of size $m \times n$ is prohibited. Some methods have been developed for solving large-scale problems, like L-BFGS, nonlinear Conjugate Gradient and related methods, but those requires the gradient which is an issue when solving (1.3). The gradient issue and its dependence of the Jacobian matrix may be solved using Automatic Differentiation Nocedal and S. Wright 2000 but this requires the use of external toolkits. To solve large-scale problems, like usually found in Machine Learning, it is common to use the *Steepest descent* or its stochastic modification Pouyanfar, Sadiq, Y. Yan, Tian, Tao, Reyes, Shyu, S.-C. Chen, and Iyengar 2018. However, those methods usually exhibit a slow convergence. Thus, in this thesis, we will present a method for solving (1.3) based in the *Levenberg-Marquardt method* Marquardt 1963. This algorithm was developed for solving *Nonlinear least square problems* and is related to the Gauss-Newton method. Although this method was developed for this type of problems, it has been used in other problems and this is an active research topic Lin, O'Malley, and Vesselinov 2016; Henn 2003; Finsterle and Kowalsky 2011; Bao, C. K. W. Yu, J. Wang, Y. Hu, and Yao 2019. To handle the large-scale component of the problem, a matrix-free approximation and a quantized representation of the Jacobian matrix will be used.

In chapter 2 we will review the current State of the Art related to nonlinear equations for large-scale problems. Chapter 3 will describe some theoretical background, related to the problem and how it will be handled in this thesis. Chapter 4 will explain nsLSQR, a proposed algorithm for solving large-scale linear least-square problems, designed to be used in the Levenberg-Marquardt method or related algorithms. In chapter 5, the Quantization approximation is explained, which is one of the central parts of our nonlinear solver and complements the nsLSQR algorithm. Chapter 6 will explain the remaining components related to Levenberg-Marquardt, the combination of the quantized approximation and nsLSQR, and will present the final algorithms. Chapter 7 contains numerical experiments, testing the different parts of the proposed methods and showing their results. Finally, but of no lesser importance, chapter 8 shows the conclusion of our work and set some possible lines for future work.

Chapter 2

State of the art

Over-determined nonlinear system of equations arise commonly in different fields and it is not rare to use classical algorithms to solve such problems. Since solving non-trivial nonlinear problems is hard, iterative methods are usually used. A classical approach for solving nonlinear problems is the usage of *Line-search methods* Nocedal and S. Wright 2000. Let suppose that an unconstrained minimization problem is desired to be solved, like the one presented in (1.3). For a current approximation of the minimum, namely $\mathbf{x}_i$, these methods compute the next approximation in the following iterative manner,

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \alpha_i \, \mathbf{p}_i,$$

for $\alpha_i > 0$ some real scalar and $\mathbf{p}_i$ some direction vector or step. Different values of the step $\mathbf{p}_i$ generates different line-search methods. Usually, the value of $\mathbf{p}_i$ is a descent direction vector Nocedal and S. Wright 2000, i.e. a vector that satisfy $\mathbf{p}_i^T \nabla g(\mathbf{x}_i) < 0$. This is an important desired property since for a sufficiently small value of $\alpha_i > 0$, we have that

$$g(\mathbf{x}_i - \alpha_i \nabla g(\mathbf{x}_i)) < g(\mathbf{x}_i).$$

This shows that the value of the steplength α_i also has an important role and, in general, there is no a golden rule for its computation.

If α_i is too small, the method suffers from a slow convergence. On the contrary, if α_i is too large, there is no guarantee that the method may converge. A simple approach is to modify its value, for example, using a backtracking method, until some properties or inequalities hold Nocedal and S. Wright 2000.

A popular line-search method is the *Steepest descent method* Nocedal and S. Wright 2000, which is a well-known method in the Machine Learning research community. The Steepest descent method use as a step direction the negative of the gradient of the function that is minimized, i.e. $\mathbf{p}_i^{\mathrm{SD}} = \nabla g(\mathbf{x}_i)$. The method is based in the fact that the negative of the gradient of a function points to the direction of maximum decrease of a function Nocedal and S. Wright 2000. Unfortunately, it is not widely used, mostly due to its slow convergence rate Nocedal and S. Wright 2000 and, in the Machine learning community, also due to its high computational cost. In

Machine Learning problems, the gradient is composed by the sum of the gradient of the loss function for each element in the dataset or training set. This requires the evaluation of the gradient over the entire dataset, which is an issue for modern high-demanding models, like deep neural networks, that requires a very large amount of data to achieve a good prediction performance. A preferred version of the Steepest descent method is the *Stochastic Descent gradient method* Pouyanfar, Sadiq, Y. Yan, Tian, Tao, Reyes, Shyu, S.-C. Chen, and Iyengar 2018. This variation of the classic gradient method avoids the need of loading a large-scale dataset, reducing the computational requirements of the method, by randomly choosing elements of the dataset to be used in the gradient computation.

The *Spectral Gradient Method*, globalized in Raydan 1997, is a gradient method where the calculation of the steplength differs from a steepest descent approach. The method propose a fixed expression to compute the steplength, which has a low computational cost and storage requirement. Despite its simple structure, in recent years it has gained attention in the scientific community Dai and Zhang 2001; Biglari and Solimanpur 2013; Hongwei, Z. Liu, and Dong 2017; Mohammad and Waziri 2019. A modification to the Spectral method was proposed by La Cruz et al. Cruz and Raydan 2003; Cruz, Martínez, and Raydan 2006.

Another member in the set of line-search methods, that is also suitable for some large-scale problems, is the *Nonlinear Conjugate Gradient method* Nocedal and S. Wright 2000; W. Cheng, Xiao, and Q.-J. Hu 2009. Its usage is mostly due to its simplicity and low-memory requirements. It is based on the linear Conjugate Gradient method Saad 2003 but use the gradient of the objective function to optimize. The step size is defined as follows,

$$\mathbf{p}_{i+1}^{\mathrm{NCG}} = -\nabla g(\mathbf{x}_i) + \beta_i \mathbf{p}_i^{\mathrm{NCG}},$$

where β_i is a scalar that ensures that $\mathbf{p}_{i+1}^{\mathrm{NCG}}$ and $\mathbf{p}_i^{\mathrm{NCG}}$ are conjugate. For the linear Conjugate Gradient Method, the value of β_i may be analytically computed. For nonlinear problems, this is not possible. How to compute its value is an active research field in the scientific community, and different approaches and values have been proposed. See Nocedal et al. Nocedal and S. Wright 2000, Cheng et al. W. Cheng, Xiao, and Q.-J. Hu 2009, Dai et al. Dai and Kou 2013, and Yuan et al. Yuan, T. Li, and W. Hu 2020 for a survey of different computation strategies.

Newton-based methods are another classical approach to solve nonlinear problems, which are based on the *Newton's method* Sauer 2011. For solving a nonlinear system of equations like (1.1), the Newton's method build iteratively the solution as

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \Delta\mathbf{x}_i^{\mathrm{NM}},$$

where the *Newton step* $\Delta\mathbf{x}_i^{\mathrm{NM}}$ is computed from the following system of linear equations

$$J(\mathbf{x}_i)\,\Delta\mathbf{x}_i^{\mathrm{NM}} = -\mathbf{F}(\mathbf{x}_i) \Leftrightarrow \Delta\mathbf{x}_i^{\mathrm{NM}} = -J^{-1}(\mathbf{x}_i)\,\mathbf{F}(\mathbf{x}_i), \tag{2.1}$$

where $J(\mathbf{x}_i)$ corresponds to the Jacobian matrix of $\mathbf{F}$ evaluated at $\mathbf{x}_i$. The described iteration procedure is performed until convergence. An important property of the

Newton's method is that converges quadratically to the solution Kelley 1995, for a sufficiently close initial guess $\mathbf{x}_0$. The method is locally convergent Kelley 1995, but there are different approaches to make the method globally convergent Kelley 1995; Press, Teukolsky, Vetterling, and Flannery 1992. An important requirement of the method, that is related to its memory storage requirements and the computational cost of the method, is the Jacobian matrix. For large-scale problems, the storage of this matrix is forbidden. An important modification to the Newton's method is the *Jacobian-Free Newton-Krylov method* (JFNK) Knoll and Keyes 2004. This method is a matrix-free variation of the Newton's method, where the linear system is solved using a Krylov-based linear solver, like GMRes Saad 2003; Sauer 2011. *Matrix-free* methods do not need explicit access to matrices but only requires its action over arbitrary vectors, i.e. requires a procedure to compute the matrix-vector product. Since such methods do not requires the matrix explicitly, they are suitable for large-scale problems. For the JFNK, its matrix-free property is obtained by means of two elements:

- The first one is the use of a Krylov-subspace method to solve the linear system at each iteration. Krylov subspace linear solvers operates over the matrix in a matrix-vector product only. Thus, if a procedure to compute the product of the Jacobian matrix times a vector is given, Krylov subspace methods can solve the linear system without forming the matrix.

- The second one, is the usage of a *finite difference approach* to approximate the required product. An example of a second-order approximation of the product $J(\mathbf{x}_i)\,\mathbf{v}$ is given by the following equation,

$$J(\mathbf{x}_i)\mathbf{v} \approx \frac{\mathbf{F}(\mathbf{x}_i + \varepsilon\mathbf{v}) - \mathbf{F}(\mathbf{x}_i - \varepsilon\mathbf{v})}{2\varepsilon}. \tag{2.2}$$

  A higher-order approximation is possible, however this increase the computational cost of such approximations. First-order approximation is also possible at a single new function evaluation.

Both, Newton's method and the JFNK method are restricted to nonlinear problems where $m = n$, that is, its number of unknowns equals its number of equations. This is not the case for overdetermined nonlinear systems of equations. Other examples of method for large-scale problems for system of nonlinear equations with equal number of equations and unknowns are Leong, Hassan, and Yusuf 2011; W. Cheng, Xiao, and Q.-J. Hu 2009; Noor] and Waseem 2009; Abubakar and Kumam 2018; Cruz, Martínez, and Raydan 2006; Cruz and Raydan 2003. An approach to use Newton's method to solve overdetermined problem is by considering the property that a minimum of (1.3) must satisfy $\nabla g(\mathbf{x}) = \mathbf{0}$. Note that $\nabla g : \mathbb{R}^n \to \mathbb{R}^n$. Therefore, the Newton's method can be used over the gradient of $g(\mathbf{x})$. This would require the Jacobian matrix of $\nabla g(\mathbf{x})$ or, equivalently, the Hessian of $g(\mathbf{x})$. Let $H_g(\mathbf{x})$ be the Hessian matrix of $g(\mathbf{x})$. The equivalent to the Newton equation (2.1) is given by

$$H_g(\mathbf{x}_i)\,\Delta\mathbf{x}_i^{\mathrm{NM}} = -\nabla g(\mathbf{x}_i), \tag{2.3}$$

and the update for the current solution is computed as

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \Delta\mathbf{x}_i^{\mathrm{NM}} = \mathbf{x}_i - H_g^{-1}(\mathbf{x}_i)\,\nabla g(\mathbf{x}_i). \tag{2.4}$$

An important remark of this approach is that the convergence to a minimum is not guarantee. The reason for this is that the property of $\nabla g(\mathbf{x}) = \mathbf{0}$ is also satisfied in a local maximum. Therefore, solving $\nabla g(\mathbf{x}) = \mathbf{0}$ using the Newton's method may convergence to a local minimum, a local maximum or even a saddle point.

Quasi-Newton methods Dennis and Moré 1977 is another branch of methods to solve nonlinear problems that are widely used, and are closely related to Newton's method. Quasi-Newton methods are based on the idea of using an approximation or "similar matrix" instead of $J(\mathbf{x}_i)$ in (2.1) or an approximation of its inverse. In general, Quasi-Newton methods for optimization or nonlinear problems, are based on a quadratic model of the objective function at the current solution point, which is given by the following equation,

$$m_i(\mathbf{x}) = g(\mathbf{x}_i) + \nabla g(\mathbf{x}_i)^T\,(\mathbf{x} - \mathbf{x}_i) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_i)^T\,B_i\,(\mathbf{x} - \mathbf{x}_i),$$

where B_i is the Hessian matrix of $g(\mathbf{x})$, following the notation used in Nocedal and S. Wright 2000. Note that this is just a second order Taylor expansion for $g(\mathbf{x})$ at $\mathbf{x}_i$. The main proposal of Quasi-Newton methods is to avoid the computation of the true Hessian matrix. Instead, these methods update it at every iteration, starting from an initial matrix B_0. To compute the next matrix B_{i+1}, given B_i, Quasi-Newton methods solves the following optimization problem,

$$\min_{B_{i+1}} \|B_{i+1} - B_i\|, \tag{2.5}$$

subject to B_{i+1} being symmetric positive-definite and that the following equation holds,

$$B_{i+1}\mathbf{s}_i = \mathbf{y}_i, \tag{2.6}$$

where $\mathbf{s}_i = \mathbf{x}_{i+1} - \mathbf{x}_i$ and $\mathbf{y}_i = \nabla g(\mathbf{x}_{i+1}) - \nabla g(\mathbf{x}_i)$. Equation (2.6) is known as the *Secant equation*. The minimization problem (2.5) subject to the mentioned restriction has a unique solution for B_{i+1} Nocedal and S. Wright 2000. The use of different norms in (2.5) give rise to different Quasi-Newton methods. An important Quasi-Newton method is given by the DFP update formula, named by Davidon-Fletcher-Powell. The DFP updates is given by

$$B_{i+1} = (I - \rho_i\mathbf{y}_i\mathbf{s}_i^T)\,B_i\,(I - \rho_i\mathbf{s}_i\mathbf{y}_i^T) + \rho_i\mathbf{y}_i\mathbf{y}_i^T,$$

where

$$\rho_i = \frac{1}{\mathbf{y}_i^T\mathbf{y}_i}.$$

Although having an expression to compute B_{i+1} easily from B_i, usually its inverse is used. This reduce the computational complexity of the overall method by replacing

a solution of a linear system of equation by a simple matrix-vector product. The inverse of B_i, which is denoted by $H_i = B_i^{-1}$, can be easily computed using the *Sherman–Morrison–Woodbury formula* Nocedal and S. Wright 2000. Using this formula, the value of H_{i+1} can be obtained as,

$$H_{i+1} = H_i - \frac{H_i\,\mathbf{y}_i\mathbf{y}_i^T\,H_i}{\mathbf{y}_i^T\,H_i\,\mathbf{y}_i} + \frac{\mathbf{s}_i\mathbf{s}_i^T}{\mathbf{y}_i^T\mathbf{s}_i}.$$

The previously described approach is effective, but the BFGS method Nocedal and S. Wright 2000, and its limited-memory version for handling large-scale problems, L-BFGS Nocedal and S. Wright 2000, are preferred due to its performance. The idea of the BFGS updating formula is similar to the DFP formula, but the secant equation is applied over the approximated inverse of the Hessian, H_{i+1}, instead of the approximated Hessian B_{i+1}. That is, the secant equation is given by

$$H_{i+1}\mathbf{y}_i = \mathbf{s}_i.$$

So, the BGS optimization problem used to find H_{i+1} is the following,

$$\min_{H_{i+1}} \|H_{i+1} - H_i\|, \tag{2.7}$$

subject to H_{i+1} being symmetric positive-definite and to the secant equation (2). The update formula in BFGS, analogous to the formula used in DFP, is

$$H_{i+1} = (I - \rho_i\mathbf{s}_i\mathbf{y}_i^T)\,H_i\,(I - \rho_i\mathbf{y}_i\mathbf{s}_i^T) + \rho_i\mathbf{s}_i\mathbf{s}_i^T.$$

The selection of the initial approximation, H_0, is not a trivial task and different approach may be used. Classical options are the Identity matrix or an approximated Hessian computed by Finite Difference Nocedal and S. Wright 2000. The solution of the optimization problem, for instance (1.3), is computed iteratively as

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \alpha_i H_i \nabla g(\mathbf{x}_i), \tag{2.8}$$

where α_i is chosen to ensure convergence. Note that the computation given by (2.8) is similar to (2.4) since H_i in the BFGS notation is the approximation of the inverse of the Hessian matrix of $g(\mathbf{x})$. An important advantage of BFGS over Newton's method applied directly over $\nabla g(\mathbf{x})$ is that the step $-H_i\nabla g(\mathbf{x}_i)$ is a descent direction for $g(\mathbf{x})$ at $\mathbf{x}_i$, since

$$\nabla g(\mathbf{x}_i)^T\ (-H_i\nabla g(\mathbf{x}_i)) = -\nabla g(\mathbf{x}_i)^T H_i \nabla g(\mathbf{x}_i) < 0.$$

This is ensured since H_i is positive-definite by construction. Using the Newton's method directly over $\nabla g(\mathbf{x})$ requires the true Hessian of $g(\mathbf{x})$, which does not necessarily is positive-definite and, therefore, may produce a non-descent direction. This descent-direction property, its superlinear convergence Nocedal and S. Wright 2000 and its low computational requirements Nocedal and S. Wright 2000, makes BFGS a widely used method for general unconstrained optimization problems. However, for

large-scale problems the method may require a prohibited amount of memory storage since the matrices H_i are usually dense. The Limited-memory BFGS Nocedal and S. Wright 2000, or L-BFGS, is a modification to BFGS aimed to use BFGS for large-scale problems. Note that the matrix H_i is used in a matrix-vector product form, in (2.8). Also note that H_{i+1} depends from H_i, ρ_i, $\mathbf{y}_i$ and $\mathbf{s}_i$. This analysis may be repeated for H_i, recursively. So H_{i+1} depends from H_0 and all previous ρ_i, $\mathbf{s}_i$ and $\mathbf{y}_i$. Using this, L-BFGS approximates the matrix-vector product of H_{i+1} and a vector by storing a fixed number of previous pairs $\{\mathbf{y}_i, \mathbf{s}_i\}$ and build the desired product of H_{i+1} implicitly using these vectors. Since storing all pairs from 0 to the current iteration i is not feasible, L-BFGS only maintains an small fixed number of such pairs of vectors. Due to its limited-memory property and good performance, it has been used for training deep neural networks Badem, Basturk, Caliskan, and Yuksel 2017; X. Liu, S. Liu, Sha, J. Yu, Z. Xu, X. Chen, and Meng 2018; Bottou, Curtis, and Nocedal 2018 or large-scale problems Nocedal and S. Wright 2000. More Quasi-Newton method exists and is an active research topic. For example, in Leong et al. Leong, Hassan, and Yusuf 2011 and Hassan et al. Mohammad and Santos 2018,a Quasi-Newton approach is used to build diagonal matrices, reducing considerably the required memory of the algorithm.

Another popular method used to solve nonlinear problems, is the Gauss-Newton Sauer 2011, which is based on the Newton's method. The idea of the the Gauss-Newton method begins with the same approach used by the Newton's method to solve an overdetermined nonlinear system of equations. In (2.3), the matrix $H_g(\mathbf{x})$ corresponds to the Hessian matrix of $g(\mathbf{x})$, which is given by,

$$H_g(\mathbf{x}) = J^T(\mathbf{x})\, J(\mathbf{x}) + \sum_{i=1}^{m} f_i(\mathbf{x})\, H_{f_i}(\mathbf{x}),$$

where $J(\mathbf{x})$ is the Jacobian matrix of $\mathbf{F}$, $f_i : \mathbb{R}^n \to \mathbb{R}$ is the i-th component of the function $\mathbf{F}$ and $H_{f_i}(\mathbf{x})$ denotes the Hessian matrix of $f_i(\mathbf{x})$. Since usually it is not easy to compute H_g explicitly and also does not guarantee that will decrease the function $g(\mathbf{x})$, since the step $\Delta\mathbf{x}_i$ in (2.3) is not a descent direction for $g(\mathbf{x})$, the Gauss-Newton method approximates H_g using the first-order derivative information only, i.e. $H_g(\mathbf{x}) \approx J^T(\mathbf{x})\, J(\mathbf{x})$. Using this approximation, the Gauss-Newton method update the solution as

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \Delta\mathbf{x}_i^{\mathrm{GN}},$$

where the step $\Delta\mathbf{x}_i^{\mathrm{GN}}$ is obtained solving the following linear system of equations

$$J^T(\mathbf{x}_i)\, J(\mathbf{x}_i)\Delta\mathbf{x}_i^{\mathrm{GN}} = -\nabla g(\mathbf{x}_i) = -J^T(\mathbf{x}_i)\, \mathbf{F}(\mathbf{x}_i). \tag{2.9}$$

Note that the Jacobian matrix $J(\mathbf{x}_i)$ is of size $m \times n$. Therefore, the matrix $J^T(\mathbf{x}_i)\, J(\mathbf{x}_i)$ is a square matrix of size $n \times n$. Under suitable assumptions, the method is locally convergent Nocedal and S. Wright 2000. A backtracking-line-search strategy may be used to make it a globally convergent method Nocedal and S. Wright 2000. Also, if the matrix $J^T(\mathbf{x}_i)\, J(\mathbf{x}_i)$ dominates over $\sum_{i=1}^{m} f_i(\mathbf{x})\, H_{f_i}(\mathbf{x})$

in H_g, then the method may achieve quadratic convergence Nocedal and S. Wright 2000. Another important property of Gauss-Newton is that its step $\Delta \mathbf{x}_i^{\text{GN}}$ is a descent direction, when $J(\mathbf{x}_i)$ is full-rank and $\nabla g(\mathbf{x}_i) \neq \mathbf{0}$. This is easy to prove, since the matrix $J^T(\mathbf{x}_i)\, J(\mathbf{x}_i)$ is positive-definite if $J(\mathbf{x}_i)$ is full-rank. Therefore, we have that

$$\langle \nabla g(\mathbf{x}_i), \Delta \mathbf{x}_i^{\text{GN}} \rangle = -\nabla g(\mathbf{x}_i)^T \left(J^T(\mathbf{x}_i)\, J(\mathbf{x}_i) \right)^{-1} J^T(\mathbf{x}_i)\, \mathbf{F}(\mathbf{x}_i) < 0,$$

since the inverse of a positive-definite matrix is also positive-definite. This is an important property, that is not shared with the Newton's method applied over $\nabla g(\mathbf{x})$.

A mayor drawback of the Gauss-Newton method is when the matrix $J(\mathbf{x}_i)$ is rank-deficient or the matrix $J^T(\mathbf{x}_i)\, J(\mathbf{x}_i)$ is nearly singular. The Levenberg-Marquardt method Marquardt 1963; Moré 1978; Nocedal and S. Wright 2000 use a regularization parameter $\lambda > 0$, sometimes called damping factor, to overcome these issues. In general, the Levenberg-Marquardt method is preferred over Gauss-Newton, and has been used successfully in neural networks Ibrahimy, Ahsan, and Khalifa 2013; Smith, B. Wu, and Wilamowski 2019 and also is an active research field Bilski, Kowalczyk, and Grzanek 2018; Lin, O'Malley, and Vesselinov 2016; L. Chen 2016a; L. Chen 2016b; Huang and Ma 2019. Instead of solving (2.9), the Levenberg-Marquardt method compute the step as

$$\left(J^T(\mathbf{x}_i)\, J(\mathbf{x}_i) + \lambda I_n \right) \Delta \mathbf{x}_i^{\text{LM}} = -J^T(\mathbf{x}_i)\, \mathbf{F}(\mathbf{x}_i), \tag{2.10}$$

where I_n denotes the $n \times n$ identity matrix. The update of the current solution is given by $\mathbf{x}_{i+1} = \mathbf{x}_i + \Delta \mathbf{x}_i^{\text{LM}}$. Since the matrix $J^T(\mathbf{x}_i)\, J(\mathbf{x}_i)$ is guaranteed to be at least semi-positive definite, the regularization parameter $\lambda > 0$ makes the matrix positive-definite. Since the matrix $J^T(\mathbf{x}_i)\, J(\mathbf{x}_i) + \lambda I$ is positive definite, the step $\Delta \mathbf{x}_i^{\text{LM}}$ is a descent direction for $g(\mathbf{x})$. Also, note that the step is given by

$$\Delta \mathbf{x}_i^{\text{LM}} = -\left(J^T(\mathbf{x}_i)\, J(\mathbf{x}_i) + \lambda I_n \right)^{-1} J^T(\mathbf{x}_i) \mathbf{F}(\mathbf{x}_i)$$

This means that

- Larger values of λ makes the matrix λI_n to be dominant over $J^T(\mathbf{x}_i)\, J(\mathbf{x}_i)$. Therefore, $J^T(\mathbf{x}_i)\, J(\mathbf{x}_i) + \lambda I_n \approx \lambda I_n$ and the step $\Delta \mathbf{x}_i^{\text{LM}} \approx (\lambda I_n)^{-1} J^T(\mathbf{x}_i) \mathbf{F}(\mathbf{x}_i) = \frac{1}{\lambda} J^T(\mathbf{x}_i) \mathbf{F}(\mathbf{x}_i)$, that is, similar to an small gradient descent step.
- Smaller values of λ makes $J^T(\mathbf{x}_i)\, J(\mathbf{x}_i)$ to be dominant over I_n, making the step closer to a Gauss-Newton step.

This impact of λ is important since taking larger values produce an step close to a gradient descent approach, which is know to be convergent for an small steplength. Smaller values produce an step close to a Gauss-Newton step and, therefore, improving the convergence. How to modify the value of λ at each iteration is a key element in the Levenberg-Marquardt method and is still an important research topic Fan and Pan 2009; Cui, Zhao, B. Xu, and Gao 2017. Marquardt proposed a simple method of a constant factor used to increase or decrease the damping factor Marquardt 1963, depending if the objective function is reduced. A more sophisticated

approach was lately developed in Fletcher 1971. In Moré 1978, the work was extended and a more numerically stable method was proposed, where a trust-region like approach was used in the computation of the damping factor. Also, the inequality $g(\mathbf{x}_k + \Delta\mathbf{x}_k(\lambda)) < g(\mathbf{x}_k)$ was not used and the criteria for increase or decrease the damping factor was related to the ratio between the actual reduction in the nonlinear function and the predicted reduction of the linear model. This implementation is more reliable than the original proposed by Marquardt and is still used in modern implementation of some libraries, like the implementation contained in the *Scipy* library[1], for Python. Although the method posses good convergence properties and general performance, it is not suitable for large-scale problems. Since it requires the Jacobian matrix and its transpose, large-scale problem requires a prohibitive amount of storage requirements. For this reason, the Levenberg-Marquardt method is preferred for small or medium-scale problems. Note that using an approximation like (2.2), it is possible to compute the matrix-vector product $J(\mathbf{x}_i)\mathbf{v}$ in a matrix-free approach. Unfortunately, to the author's knowledge, there is no a similar low computational cost approximation for the product between the transpose of the Jacobian matrix and a vector. It is possible to compute the product $J^T(\mathbf{x}_i)\mathbf{w}$ in a matrix-free approach Sanhueza and Torres 2017, but it has a high computational cost.

The more important issues for Newton type method, for large-scale problems, are

- Most of method requires a Jacobian matrix, the Hessian matrix or an approximation of it. For large-scale problem, the storage of such matrix is prohibited. However, the L-BFGS method is capable of handling such issue.

- All previous algorithms, if used for solving (1.3), requires the gradient of $g(\mathbf{x})$. This, for large-scale problem, is an issue since

$$\nabla g(\mathbf{x}) = J^T(\mathbf{x})\mathbf{F}(\mathbf{x}).$$

  Constructing and storing explicitly the matrix $J^T(\mathbf{x})$ is not feasible, and a matrix-free approximation demands a high computational cost. If a cheap procedure to compute the gradient of $g(\mathbf{x})$, then previously described algorithm are capable of handling efficiently a large-scale problem.

As noted, all previous issues are related to the computation of a large Jacobian matrix or a gradient. An important approach to handle such issue is the use of *Automatic Differentiation* Nocedal and S. Wright 2000; Griewank and Walther 2008, also called *Algorithmic Differentiation*. Automatic Differentiation is an important tool that is based in the idea that every function is a composition of fundamental functions like sine, cosine, exponential and so on. Using the *Chain rule*, the derivatives are computed implicitly. Using their *forward mode* or *reverse mode*, different ways to compute the desired derivative, it is possible to compute dot products or matrix-vector products between gradients, Jacobian matrix and other structures. This procedure is

[1] `https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.least_squares.html`

used in machine learning for training Baydin, Pearlmutter, Radul, and Siskind 2017 or to perform products needed by several non derivative-free methods like BFGS (or its limited memory version), Newton method, Levenberg-Marquardt and other algorithms W. Xu, Zheng, and Hayami 2016; Safiran, Lotz, and Naumann 2016; X. Fu, S. Li, Fairbank, Wunsch, and Alonso 2015. Also, another advantage of Automatic Differentiation is that is suitable for parallelization Förster and Naumann 2013. It is important to note that Automatic Differentiation is different from Symbolic differentiation, where there is a symbolic data structure that represent each function. One of the main advantages of these types of methods is that does not suffer from truncation errors Griewank and Walther 2008 and, even if it is similar to symbolic mathematics, it is much faster. Unfortunately, this approach requires that functions to be programmed using specific toolkits to performs the required derivatives and this requirement is not possible to meet when functions and methods were already programmed or implemented previously, or if their are an output from another program or toolkit.

Given this context, in this thesis we propose a method for solving over-determined nonlinear system of equations by means of an unconstrained minimization approach. This problem is solved by the Levenberg-Marquardt method. As described, the Levenberg-Marquardt method requires the Jacobian matrix and its transpose. To handle large-scale problems, our approach combines the use of an approximation based on finite-difference, to approximate the product of the Jacobian matrix, and *Quantization* is used to approximate the transpose of the Jacobian matrix using a restricted-memory approach. To solve the inner least-square problem, we also propose a method based on LSQR, called *nsLSQR* or *Non-Symmetric LSQR*.

The idea of a low-memory or inaccurate approximation is not new and has been tested in different methods and approaches. In Carson and Higham 2018 use the *Iterative refinement* process to solve a linear system of equation, using three different precision achieving good results and accelerating the solution process. Although they use a higher precision to perform matrix factorization, their residual are computed in a possible lower precision. In Bergou, Gratton, and Vicente 2016 the idea of multi-precision approach is used in the Levenberg-Marquardt and gradients are evaluated using a probabilistic models. Inspired for this work, in Bellavia, Gratton, and Riccietti 2018 the Levenberg-Marquardt method is used to solve a a nonlinear least-square problem, assuming that a full precision evaluation is expensive in comparison with a low precision evaluation. Those work show that solving a problem with a lower precision is a suitable approach.

The idea of using quantization as an approximation approach was inspired by the *Sparsification* concept and later from the quantization itself. The sparsification process generate a sparse matrix from an initial dense matrix using some criteria Achlioptas and Mcsherry 2007, while Quantization is a lossy compression method Sayood 2006 where a large set of values, possible an infinite range, is mapped to a discrete set of values. Quantization, as an important element in our proposal, is a very well studied field Gray and Neuhoff 1998 and has been used in many areas. One of their

recent and most popular uses is in deep neural networks, where their long training times and high memory consumption needed to store their weights makes difficult to use this methods in small devices such as phones. Aiming to reduce the neural network size, quantization has been used in neural networks successfully. Their use is not new Fiesler, Choudry, and Caulfield 1990 but its study and improvement are still an active research topic Courbariaux, Bengio, and David 2015; P. Wang and J. Cheng 2017; J. Wu, C. Yu, S. Fu, C. Liu, Chien, and Tsao 2019; Yang, Deng, S. Wu, T. Yan, Xie, and G. Li 2020.

Chapter 3

Theoretical background

In this chapter, the general mathematical model of our approach is given. Although some equations were already defined in chapter 1, we will define again some of them in this chapter for better readability of the explanation. Also the following notation will be used for the rest of this thesis: I_n denotes the $n \times n$ identity matrix, $\|\cdot\|$ denotes the Euclidean two-norm, $\mathbf{0}_n$ denotes a zero vector in $\mathbb{R}^n$ and $\mathbf{1}_n$ denotes a vector in $\mathbb{R}^n$ whose coefficients are all equal to one.

An over-determined nonlinear system of equation is given by the following problem,

$$\mathbf{F}(\mathbf{x}) = \mathbf{0}, \tag{3.1}$$

where the (nonlinear) function $\mathbf{F} : \mathbb{R}^n \to \mathbb{R}^m$ is defined as,

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

As commented in chapter 1, a classical approach for solving (3.1) is by means of solving an unconstrained minimization problem. Recall that since we are considering an overdetermined problem, we have that $m \geq n$. Let $g(\mathbf{x})$ be the function defined as,

$$g(\mathbf{x}) = \frac{1}{2}\|\mathbf{F}(\mathbf{x})\|^2 = \frac{1}{2}\sum_{i=1}^{m} f_i^2(\mathbf{x}). \tag{3.2}$$

Then, the minimization problem to be solved is given by the following equation

$$\min_{\mathbf{x}\in\mathbb{R}^n} g(\mathbf{x}). \tag{3.3}$$

It is clear that a solution of (3.1) is also a solution for (3.3), but the converse is not necessarily true. There are an extensive amount of methods available to solve this problem, each one with different properties. Some of them were already discussed in chapter 2. Also mentioned in chapter 2, large-scale problems are difficult to solve

for all discussed methods. The aim of this thesis is to propose a method capable of solving large-scale overdetermined nonlinear system of equations, and the purpose of this chapter is to give the general mathematical description required to introduce our method. Our proposed method contains several inner components, and some of them will be described here. Elements that requires a more extensive explanation and detailed equations will be explained in subsequent chapters.

To begin with the explanation, the method proposed in this thesis will use the Levenberg-Marquardt method, as a nonlinear solver. As briefly mentioned in chapter 2, the Levenberg-Marquardt method is based on the following iteration

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \Delta\mathbf{x}_i, \tag{3.4}$$

starting from an initial guess $\mathbf{x}_0$. This initial guess can be a known estimated solution or any other vector, although usually choosing carefully an initial guess can improve the convergence of nonlinear methods. The value of the step $\Delta\mathbf{x}_i$ is computed by solving the following system of linear equations

$$\left(J^T(\mathbf{x}_i)J(\mathbf{x}_i) + \lambda_i I_n\right)\Delta\mathbf{x}_i = -J^T(\mathbf{x}_i)\mathbf{F}(\mathbf{x}_i), \tag{3.5}$$

where $J(\mathbf{x}_i)$ corresponds to the Jacobian matrix of $\mathbf{F}$, evaluated at $\mathbf{x}_i$, and $\lambda_i > 0$ is a regularization parameter or damping factor. In practice, the value of λ_i is used to avoid a singular matrix in the linear problem (3.5) and also is used for a trust-region approach Kelley 1999, to get a globally convergent method. As explained in chapter 2, the step computed by Levenberg-Marquardt is a descent direction for $g(\mathbf{x})$ and also the modification of the value of λ_i is used to speed-up the convergence of the method when we are close to a solution or ensure the convergence when far from the solution, using a step closer to a descent gradient approach. In general, the Levenberg-Marquardt method converge quadratically under a set of assumptions Fan and Pan 2009. Also mentioned in chapter 2, the damping factor has an important role in the performance of the method and its an active research topic. The Algorithm 1 describe the general algorithm of the Levenberg-Marquardt.

Algorithm 1 Algorithm of Levenberg-Marquardt

Require: A function $\mathbf{F}$, the number of equations m, the number of unknowns n, an initial guess $\mathbf{x}_0$, a tolerance η, $\lambda_0 > 0$.

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

1: **for** i from 0 to ∞ until some convergence criteria **do**
2: Build $J(\mathbf{x}_i)$
3: Solve the linear system $\left(J^T(\mathbf{x}_i)J(\mathbf{x}_i) + \lambda_i I\right)\Delta\mathbf{x}_i = -J^T(\mathbf{x}_i)\mathbf{F}(\mathbf{x}_i)$.
4: **if** The new trial solution $\mathbf{x}_i + \Delta\mathbf{x}_i$ improves the current solution $\mathbf{x}_i$ **then**
5: $\mathbf{x}_{i+1} = \mathbf{x}_i + \Delta\mathbf{x}_i$
6: **else**
7: $\mathbf{x}_{i+1} = \mathbf{x}_i$.
8: **end if**
9: Compute λ_{i+1} from λ_i depending on the progress of $\mathbf{x}_{i+1}$ from $\mathbf{x}_i$.
10: **end for**
11: Return the current solution $\mathbf{x}_i$.

Several method can be used to solve (3.5). Since the matrix is positive-definite, a Conjugate-Gradient Sauer 2011; Saad 2003 or Cholesky-decomposition Sauer 2011 are reasonable choices. It is important to note that the problem (3.5) are the normal equations Sauer 2011 for the following regularized linear least-square problem

$$\min_{\Delta\mathbf{x}_i\in\mathbb{R}^n} \|\mathbf{F}(\mathbf{x}_i) - J(\mathbf{x}_i)\Delta\mathbf{x}_i\|^2 + \lambda_i\|\Delta\mathbf{x}_i\|^2, \tag{3.6}$$

which is also equivalent to the following unregularized linear least-square problem

$$\min_{\Delta\mathbf{x}_i\in\mathbb{R}^n} \left\| \begin{bmatrix} \mathbf{F}(\mathbf{x}_i) \\ \mathbf{0}_n \end{bmatrix} - \begin{bmatrix} J(\mathbf{x}_i) \\ \sqrt{\lambda_i} I_n \end{bmatrix} \Delta\mathbf{x}_i \right\|^2. \tag{3.7}$$

Since (3.5) is equivalent to (3.6) or (3.7), then a linear least-square solver is also a possible choice to compute the step $\Delta\mathbf{x}_i$. There are different methods for solving linear least-square problems and some are suitable for dense or sparse problems. An important method, related to the Krylov subspace family, is the LSQR algorithm Paige and M. A. Saunders 1982a. The LSQR method is mathematically equivalent to the Conjugate Gradient method over the Normal equations of the linear least-square problem, but more stable and suitable for large sparse problems. An important property of LSQR, shared with other Krylov subspace methods, is that it does not require explicitly the matrix of the linear least-square but the product of the matrix times a vector. In particular, LSQR applied over (3.6) requires two routines:

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

If both procedures are given to LSQR, it is also suitable for large dense problems. This means that LSQR may solve a least-square problem in a matrix-free way.

However for large-scale problems we have the following issues,

- If the approximation or computation of an explicit matrix $J(\mathbf{x}_k)$, whose size is $m \times n$, does not fit in the available system memory, the problem (3.5) nor (3.6) can be solved.
- Sparse data structures is an interesting option. Unfortunately, it is not always possible to know if the Jacobian matrix will be sparse at all iterations.
- The computation of the product of the transpose of the Jacobian matrix times a vector requires a higher computational cost compared to the Jacobian matrix times a vector.

The third reason given need a more detailed explanation. The Jacobian-Free Newton-Krylov (JFNK) method Knoll and Keyes 2004, briefly described in chapter 2, use a finite-difference approach to approximate the product of the Jacobian matrix and a vector. If LSQR is used to solve (3.6) or (3.7), this finite-difference approximation can be used to compute the product $J(\mathbf{x}_k)\mathbf{v}$. Unfortunately, to the author's knowledge, a simple expression like (2.2) to compute the product $J^T(\mathbf{x}_k)\mathbf{w}$ at a low computational cost does not exist. There are some Matrix-Free approaches to compute the transpose of the Jacobian matrix times a vector Sanhueza and Torres 2017, but those demands extremely high computational cost. Therefore, it is not possible to compute the missing required product, i.e. $J^T(\mathbf{x}_k)\mathbf{w}$, with a low computational cost.

In this thesis, to solve the inner least-square problem found in the Levenberg-Marquardt method, a variant of LSQR method is proposed, which we call *nsLSQR* (Non-symmetric LSQR). This algorithm will be described in chapter 4. The main advantage of this method is that can solve the least-square problem (3.6) or (3.7) using any approximation of the product $J^T(\mathbf{x}_k)\mathbf{w}$, that is, does not require the exact product result. That is, the algorithm may solve a least-square problem even if the used approximation performs poorly. However, the usage of a good approximation improves greatly the convergence rate of nsLSQR. To propose a complete functional method, in this thesis we propose the use of *Quantization* to approximate the Jacobian matrix at a low memory cost. Such approximation will be used to approximate the needed product $J^T(\mathbf{x}_k)\mathbf{w}$. The proposed approximation depends of the number of bits desired to use per element in the approximation matrix, that is if more bits are used, the quality of the approximation improves. The importance of the quantized approximated matrix is that an small number of bits may be used to approximate the Jacobian matrix, reducing considerably the memory required for its storage. Since using an small number of bits increase the error produced by the quantized approximation, the resulting vector is an approximation and therefore cannot be used in LSQR. For this,the usage of nsLSQR is required. This is the reason why it was designed and proposed.

Chapter 4

nsLSQR: non-Symmetric LSQR

In this section, the nsLSQR method will be explained in detail. nsLSQR is based in the LSQR, briefly explained in section 3. The LSQR method requires two routines to be used: a procedure to compute $J(\mathbf{x}_i)\mathbf{v}$ and a routine to compute $J^T(\mathbf{x}_i)\mathbf{w}$. As explained in the same section, the former product is easy to compute using a finite-difference approximation. However, the latter product pose some difficulties regarding its memory requirement, for large-scale problems. The main contribution of the nsLSQR is the capability of solving a linear least-square problem requiring only two procedures:

- A procedure to compute $J(\mathbf{x}_i)\mathbf{v}$.
- A routine to *approximate* $J^T(\mathbf{x}_i)\mathbf{w}$.

The first procedure is the same requirement as the LSQR method. However, for the second requirement, *any* routine that given a vector $\mathbf{w}$ of size m and returns a vector of size n can be used. This is an important property since a more cheap approximation can be used and still the problem will be solved. However, using a poor approximation may slow down the convergence of nsLSQR considerably. See section 7 for numerical experiments showing this issues. In particular, the nsLSQR method was proposed to be used in combination with the quantized approximation, which will be explained in detail in chapter 5. However, it is important to remark that nsLSQR can handle the linear least-square problem with any approximation. For this reason, the explanation and development of LSQR and nsLSQR will not be tied to the quantized approximation in any case.

Before explaining and giving the details about the nsLSQR method, we will introduce the LSQR method first. The description in this section is connected to the solution of (3.7). For simplicity, the arguments of the Jacobian matrix $J(\mathbf{x}_i)$ will be omitted and will be treated as a matrix J. Also, the vector $\mathbf{F}(\mathbf{x}_i)$ from (3.7) will be replaced by a vector $\mathbf{b}$ to make the notation more general. Also, the vector $\mathbf{e}_i^{(j)}$ denotes the i-th canonical vector of $\mathbb{R}^j$, that is, a vector of size j whose its coefficients are all equal to zero except for the i-th coefficient, which is 1.

4.1 LSQR method

Let us consider the following least-square problem,

$$\min_{\Delta \mathbf{x} \in \mathbb{R}^n} \|\mathbf{b} - J\Delta\mathbf{x}\|^2. \tag{4.1}$$

At the moment, we will not consider the regularization factor, i.e. $\lambda = 0$. How to solve a regularized problem will be explained later in this chapter.

The LSQR Method is an algorithm to mainly solve a sparse linear system of equations. It is mathematically equivalent to the Conjugate Gradient method for the Normal equations Paige and M. A. Saunders 1982a of (4.1). Therefore, it is a Krylov subspace method for the matrix $J^T J$ and the right-hand side vector $J^T \mathbf{b}$. LSQR is based on the Bidiagonalization process proposed by Golub and Kahan Golub and Kahan 1965. The method produces the following two recurrences at the k-th iteration,

$$J\, V_k = U_{k+1}\, B_k, \tag{4.2}$$

$$J^T\, U_{k+1} = V_{k+1}\, \hat{B}_{k+1}, \tag{4.3}$$

where B_k and $\hat{B}_{k+1}$ are lower and upper bidiagonal matrices, respectively, $V_k \in \mathbb{R}^{n\times k}$ and $U_k \in \mathbb{R}^{m\times k}$ are matrices such that in exact arithmetic $V_k^T V_k = I_k$ and $U_k^T U_k = I_k$ holds. The recurrence requires a vector to begin with. For LSQR, the recurrences (4.2) and (4.3) begin with vectors $\mathbf{u}_1$ and $\mathbf{v}_1$, defined as follows,

$$\mathbf{u}_1 = \frac{\mathbf{b}}{\|\mathbf{b}\|}, \tag{4.4}$$

$$\mathbf{v}_1 = \frac{J^T\mathbf{b}}{\|J^T\mathbf{b}\|}. \tag{4.5}$$

Note that the column space of V_k is an orthogonal basis of dimension k for $\mathbb{R}^n$. Therefore, if we compute an approximate solution for (4.1) restricted to the columns pace of V_k, then we have that $\Delta\mathbf{x} = V_k\mathbf{c}_k$ for some $\mathbf{c}_k \in \mathbb{R}^k$. Thus,

$$\begin{aligned}
\mathbf{b} - J\,\Delta\mathbf{x} &= \|\mathbf{b}\|\,\mathbf{u}_1 - J\,V_k\,\mathbf{c}_k, \\
&= \|\mathbf{b}\|\,U_{k+1}\,\mathbf{e}_1^{(k+1)} - U_{k+1}\,B_k\,\mathbf{c}_k, \\
&= U_{k+1}\left(\|\mathbf{b}\|\,\mathbf{e}_1^{(k+1)} - B_k\,\mathbf{c}_k\right).
\end{aligned}$$

Therefore, the problem (4.1) is reduced to

$$\min_{\Delta\mathbf{x}\in\mathrm{range}(V_k)} \|\mathbf{b}-J\Delta\mathbf{x}\|^2 = \min_{\mathbf{c}_k\in\mathbb{R}^k} \left\|U_{k+1}\left(\|\mathbf{b}\|\mathbf{e}_1^{(k+1)} - B_k\mathbf{c}_k\right)\right\|^2 = \min_{\mathbf{c}_k\in\mathbb{R}^k} \left\|\|\mathbf{b}\|\mathbf{e}_1^{(k+1)} - B_k\mathbf{c}_k\right\|^2, \tag{4.6}$$

which is a least-square problem whose matrix B_k is of size $(k+1)\times k$. Note that the matrix J of (4.1) is of size $m \times n$. Therefore, compute a solution of (4.6)

requires fewer elemental operations than solving (4.1), specially if k is considerably smaller than n. The previous process is the mathematical core of LSQR. Using the recurrences (4.2) and (4.3), which are easy to compute since the matrices V_k and U_k are bidiagonal, an efficient QR decomposition and the bidiagonal structure of B_k allows the LSQR method to solve the small least-square problem (4.6) efficiently. This efficient procedure makes LSQR a powerful method for solving large sparse linear problems. To compute the recurrences (4.2) and (4.3), the algorithm needs the routines to compute the action of J and J^T over arbitrary vectors Paige and M. A. Saunders 1982b. If efficient and matrix-free procedures are available, then the LSQR method is computationally and memory efficient Paige and M. A. Saunders 1982b.

For a regularized least-square problem, i.e. for $\lambda > 0$, LSQR use a simple approach. Note that the following regularized least-square problem

$$\min_{\Delta\mathbf{x}\in\mathbb{R}^n} \|\mathbf{b} - J\Delta\mathbf{x}\|^2. + \lambda\|\mathbf{x}\|^2, \tag{4.7}$$

is equivalent to the following "augmented" non-regularized least-square problem

$$\min_{\Delta\mathbf{x}\in\mathbb{R}^n} \left\| \begin{bmatrix} \mathbf{b} \\ \mathbf{0}_n \end{bmatrix} - \begin{bmatrix} J \\ \sqrt{\lambda} I_n \end{bmatrix} \right\|^2. \tag{4.8}$$

Thus, LSQR solves (4.7) by means of solving the unregularized problem (4.8). Using this approach, the algorithm does not need modifications to handle regularized problems.

4.2 Modification of LSQR: nsLSQR

Let us suppose that we have an approximation of J, denoted as $\tilde{J}$, which is a constrained-memory approximation of J. A constrained-memory approximation restricts the amount of memory available for the storage of J. In particular, consider that the constrained-memory approximation $\tilde{J}$ may range from a coarse approximation to a very accurate one. Recall that the Jacobian matrix in (4.1) is not explicitly available; only the function $\mathbf{F}(\mathbf{x})$, which can be evaluated at any point. Usually, the function $\mathbf{F}$ is nonlinear but also linear instances are possible if defined as $\mathbf{F}(\mathbf{x}) = \mathbf{b} - A\,\mathbf{x}$, for a matrix $A \in \mathbb{R}^{m\times n}$. Under this assumptions, we have that

- The product $J\,\mathbf{v}$ can be approximated using a finite-difference approximation. This is possible since we have the function $\mathbf{F}$.
- Since we have the approximation $\tilde{J}$, we may approximate $J^T\,\mathbf{w}$ as $\tilde{J}^T\,\mathbf{w}$.

Now, due to the approximations used, we need to modify the identities presented in equations (4.2) and (4.3) since the *Bidiagonalization Process* proposed by Golub and Kahan Golub and Kahan 1965 will no longer be valid. The reason for this is that it depends on the transpose of the matrix of the least-square problem. Under

our suppositions, the transpose of the Jacobian matrix is not available but just the approximation $\tilde{J}$, whose accuracy is unknown. This means that unlike B_k and $\hat{B}_{k+1}$, the matrices obtained will no longer be bidiagonal.

Note that in LSQR the identities (4.2) and (4.3) are used to obtain $\mathbf{u}_{k+1}$ and $\mathbf{v}_{k+1}$, respectively. Moreover, the matrix V_k obtained defines the subspace from where the solution of the least square is computed. Therefore, a non-accurate or approximate computation of (4.3) modifies space generated by the range of the matrix V_k. This implies that the new computed vector $\mathbf{v}_{k+1}$ will be different from the computed in LSQR and the resulting algorithm will no longer produce the same Krylov subspace as LSQR. However, as long as the matrix V_k has linearly independent column vectors, it is a valid subspace or basis for the purpose of solving a reduced least-square problem. This is the key idea for our proposed extension.

For solving (4.1) with nsLSQR, we first compute the equivalent vectors of (4.4) and (4.5) as follows,

$$\tilde{\mathbf{u}}_1 = \frac{\mathbf{b}}{\|\mathbf{b}\|},$$
$$\tilde{\mathbf{v}}_1 = \frac{\tilde{J}^T\mathbf{b}}{\|\tilde{J}^T\mathbf{b}\|}.$$

Although $\tilde{\mathbf{u}}_1 = \mathbf{u}_1$, the following vectors $\tilde{\mathbf{u}}_k$ are not necessarily equal to $\mathbf{u}_k$ due to the usage of $\tilde{J}$. The k-the column of the identity (4.2) is computed as follows,

$$J\,\tilde{\mathbf{v}}_k = \beta_{1,k}\tilde{\mathbf{u}}_1 + \beta_{2,k}\tilde{\mathbf{u}}_2 + \cdots + \beta_{k+1,k}\tilde{\mathbf{u}}_{k+1}, \tag{4.9}$$

where the product $J\,\tilde{\mathbf{v}}_k$ is computed using (2.2) or a similar scheme. Note that the product $J\,\tilde{\mathbf{v}}_k$ may be computed using $\tilde{J}$, but we use a finite difference approximation that, in general, may be more accurate and in exact arithmetic, the value of the stepsize can be small enough to achieve a required accuracy. This expansion will provide us $\tilde{\mathbf{u}}_{k+1}$. Our implementation use the modified Gram-Schmidt orthogonalization Sauer 2011. Equation (4.9) will produce the following recurrence,

$$J\,\tilde{V}_k = \tilde{U}_{k+1}\,\tilde{B}_k, \tag{4.10}$$

where $\tilde{B}_k$ is no longer a bidiagonal matrix but a Hessenberg matrix of size $(k+1)\times k$. Also $\tilde{V}_k \neq V_k$. The $(k+1)$-th column of the analogous decomposition of equation (4.3) will be computed as,

$$J^T\,\tilde{\mathbf{u}}_{k+1} \approx \tilde{J}^T\tilde{\mathbf{u}}_{k+1} = \alpha_{1,k+1}\tilde{\mathbf{v}}_1 + \alpha_{2,k+1}\tilde{\mathbf{v}}_2 + \cdots + \alpha_{k+1,k+1}\tilde{\mathbf{v}}_{k+1}, \tag{4.11}$$

and will produce the following recurrence $\tilde{J}^T\tilde{U}_{k+1} = \tilde{V}_{k+1}\overline{B}_{k+1}$, where $\overline{B}_{k+1}$ will be a upper triangular $(k+1)\times(k+1)$ matrix. Note that $\tilde{U}_k \neq U_k$. This second expansion provides $\tilde{\mathbf{v}}_{k+1}$, which is also obtained using the modified Gram-Schmidt orthogonalization, in our implementation. Thus, as long as $\alpha_{k+1,k+1} \neq 0$, we will be able to obtain a new vector $\tilde{\mathbf{v}}_{k+1}$ for each *inner* iteration.

Now, for each iteration k, we will restrict the subspace to the columns space of $\tilde{V}_k$, which means $\tilde{\Delta}\mathbf{x}_k = \tilde{V}_k\,\tilde{\mathbf{c}}_k$. Therefore, we have,

$$\begin{aligned}\mathbf{b} - J\,\tilde{\Delta}\mathbf{x}_k &= \|\mathbf{b}\|\tilde{\mathbf{u}}_1 - J\,\tilde{V}_k\,\tilde{\mathbf{c}}_k,\\ &= \|\mathbf{b}\|\,\tilde{U}_{k+1}\,\mathbf{e}_1^{(k+1)} - \tilde{U}_{k+1}\,\tilde{B}_k\,\tilde{\mathbf{c}}_k,\\ &= \tilde{U}_{k+1}\left(\|\mathbf{b}\|\,\mathbf{e}_1^{(k+1)} - \tilde{B}_k\,\tilde{\mathbf{c}}_k\right).\end{aligned}$$

Thus, equation (4.1) is reduced to the following least-square problem,

$$\begin{aligned}\min_{\tilde{\Delta}\mathbf{x}_k\in\text{range}(\tilde{V}_k)}\|\mathbf{b} - J\,\tilde{\Delta}\mathbf{x}_k\|^2 &= \min_{\tilde{\mathbf{c}}_k\in\mathbb{R}^k}\left\|\tilde{U}_{k+1}\left(\|\mathbf{b}\|\mathbf{e}_1^{(k+1)} - \tilde{B}_k\tilde{\mathbf{c}}_k\right)\right\|^2,\\ &= \min_{\tilde{\mathbf{c}}_k\in\mathbb{R}^k}\left\|\|\mathbf{b}\|\,\mathbf{e}_1^{(k+1)} - \tilde{B}_k\,\tilde{\mathbf{c}}_k\right\|^2,\end{aligned} \tag{4.12}$$

which, like in equation (4.6), is also a smaller least-square problem with respect to (4.1). Equation (4.12) allows us to obtain the approximated solution $\tilde{\Delta}\mathbf{x}_k = \tilde{V}_k\tilde{\mathbf{c}}_k$. Note that the difference between $\tilde{B}_{k-1}$ and the following matrix $\tilde{B}_k$ is a new column at the end of the matrix and $k-1$ zeroes for the first $k-1$ coefficients of the last row. This small variation from the iteration $k-1$ to k is used in other methods, like GMRes Saad 2003, to perform an efficient QR decomposition. Our implementation also use these fact to solve (4.12) faster. Although GMRes usually uses a plane rotation, our implementation uses a Modified Gram-Schmidt procedure for simplicity. For a regularized least-square problem, we propose to use the same idea used by LSQR, i.e. solve (4.8).

Finally but of no lesser importance, we note that nsLSQR requires the storage of the dense matrices $\tilde{V}_k$, $\tilde{U}_k$ and $\tilde{B}_k$, whose memory requirements increase at each iteration. Since the dimension of $\tilde{V}_k$, $\tilde{U}_k$ and $\tilde{B}_k$ depends on the number of iterations k, more iteration increase its size and thus, the memory required for its storage. A restart strategy Saad 2003; Fong and M. Saunders 2011 is a common approach in Krylov-subspace methods to avoid the increase of memory produced at each iteration. Giving two parameters t_{in} and t_{out}, the method iterates as usual until t_{in} iterations has been performed. Then, the current computed solution is used as a initial guess and the method is executed from the beginning, dropping the current matrices $\tilde{V}_k$, $\tilde{U}_k$ and $\tilde{B}_k$. This procedure is repeated t_{out} times. Using this approach, the larger matrix stored in the method has size of $m \times t_{\text{in}}$. If $t_{\text{in}} < n$, then the method will use less memory compared to a non-restarted execution.

One question that may arise is the benefits of using nsLSQR instead of a direct QR decomposition over the linear least-square problem, considering that nsLSQR also have dense matrices that need to be stored. The answer to this is that, for QR, a dense $m \times n$ matrix need to be computed. This is forbidden for large scale problems. In contrast, although nsLSQR requires dense matrices, its memory usage increase over each iteration. Also, nsLSQR computes an approximated solution at each iteration and restart. Depending on the problems, the solution can be found early in the iteration process, saving computations and memory.

Algorithm 2 shows an outline of the algorithm described. Since a regularized least-square is solved using a bigger unregularized problem, the Algorithm 2 does not consider a value λ. Although the presented algorithm is general, it gives a general idea of the structure of the algorithm. For now, the Algorithm 2 does not consider the case of a breakdown, i.e. that $\alpha_{k+1,k+1} = 0$ and $\beta_{k+1,k} = 0$. More details will be given later in this chapter.

Algorithm 2 General nsLSQR algorithm.

Require: A function $\mathbf{F}$, a vector $\mathbf{x}_i$ 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 η, a number of iterations t_{inner} and t_{outer} to control the restarts, an initial guess $\mathbf{x}_0$ which usually is set to be the zero vector

Ensure: An approximated solution $\Delta\mathbf{x}$.

1: **for** i from 1 to t_{outer} **do**
2: $\quad\mathbf{r} := \mathbf{b} - J\mathbf{x}_0$ using a finite-difference approximation.
3: $\quad\tilde{\mathbf{u}}_1 := \frac{\mathbf{r}}{\|\mathbf{r}\|}$
4: $\quad\tilde{\mathbf{v}}_1 := \frac{\tilde{J}^T\mathbf{r}}{\|\tilde{J}^T\mathbf{r}\|}$
5: $\quad$**for** k from 1 to t_{inner} **do**
6: $\quad\quad$Compute $\tilde{\mathbf{u}}_{k+1}$ from (4.9) using a modified Gram-Schmidt procedure.
7: $\quad\quad$Compute $\tilde{\mathbf{v}}_{k+1}$ from (4.11) using a modified Gram-Schmidt procedure.
8: $\quad\quad$Solve the least-square problem $\min_{\tilde{\mathbf{c}}_k \in \mathbb{R}^k} \left\| \|\mathbf{r}\| \, \mathbf{e}_1^{(k+1)} - \tilde{B}_k \, \tilde{\mathbf{c}}_k \right\|^2$, for $\tilde{\mathbf{c}}_k$.
9: $\quad\quad\Delta\mathbf{x}_k := \tilde{V}_k \, \tilde{\mathbf{c}}_k$.
10: $\quad\quad$**if** The solution $\Delta\mathbf{x}_k$ fulfill a convergence criteria using the tolerance η **then**
11: $\quad\quad\quad$Returns $\Delta\mathbf{x}_k$.
12: $\quad\quad$**end if**
13: $\quad$**end for**
14: $\quad\mathbf{x}_0 := \mathbf{x}_0 + \Delta\mathbf{x}$
15: **end for**
16: Returns $\Delta\mathbf{x}_k$.

4.3 Relation of nsLSQR and GMRes

There is an important relation between nsLSQR and GMRes, which is not evident only with the development made in section 4.2. This relation can be obtained by the use of equations (4.10) and (4.11). Remark that the identity (4.11) is given by,

$$\tilde{J}^T \tilde{U}_{k+1} = \tilde{V}_{k+1} \overline{B}_{k+1}. \tag{4.13}$$

If we multiply (4.10) by $\tilde{J}^T$, and then use the identity provided by (4.11), we obtain,

$$\tilde{J}^T J \tilde{V}_k = \tilde{J}^T \tilde{U}_{k+1} \, \tilde{B}_k = \tilde{V}_{k+1} \, \overline{B}_{k+1} \, \tilde{B}_k. \tag{4.14}$$

Since $\overline{B}_{k+1}$ is an upper Hessenberg matrix and $\tilde{B}_k$ is upper triangular, the product $\overline{B}_{k+1} \, \tilde{B}_k$ results in an upper Hessenberg matrix. This is indeed the partial Hessenberg

decomposition for the matrix $\tilde{J}^T J$. Moreover, it will be the same decomposition we will obtain if we solve the following linear system of equations using GMRes Saad 2003,

$$\tilde{J}^T J \Delta \mathbf{x} = \tilde{J}^T \mathbf{b}, \tag{4.15}$$

where the orthonormalized Krylov subspace will be $\tilde{V}_k$ and the partial Hessenberg matrix will be $H_k = \overline{B}_{k+1} \tilde{B}_k$. Note that 4.15 is similar to the normal equations of (4.1) but instead of using the matrix J we are using its approximation $\tilde{J}$. This shows that nsLSQR, at the k-th iteration, is solving (4.1) restricted to the k-th Krylov subspace generated by $\tilde{J}^T J$, which is,

$$\mathcal{K}_k = \left\{ \tilde{J}^T \mathbf{b}, \tilde{J}^T J \, \tilde{J}^T \mathbf{b}, \ldots, \left(\tilde{J}^T J \right)^{k-1} \, \tilde{J}^T \mathbf{b} \right\}.$$

This shows that better approximations of $\tilde{J}$ will make nsLSQR to use a Krylov subspace closer to the one generated by $J^T J$, improving its convergence.

4.4 Breakdown in nsLSQR

Considering the equivalence between nsLSQR and GMRes, described in section 4.3, we have at least two possible cases where the iteration in nsLSQR can breakdown. These are

(i) When the solution of the linear system (4.15) is found, but it may not necessarily be equal to the solution that minimizes (4.1) since the linear system solved is not the normal equation of the least-square problem.

(ii) When the matrix $\tilde{J}^T J$ is singular.

A breakdown also occurs if $\beta_{k+1,k} = 0$. 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. For this reason, is not considered in the previous points. So if a breakdown is found, we cannot compute either of the following vectors $\tilde{\mathbf{v}}_{k+1}$, and therefore, the iterations cannot continue. This means that the coefficient $\alpha_{k+1,k+1}$ from equation (4.11) is zero. However, note that this breakdown is not related to the problem itself but the method used to solve the equation. To continue the process to compute an approximated solution, the method needs to extend the basis using a new linearly independent vector. For simplicity, our implementation use a random orthogonalized vector.

Also, note that the use of the regularizer $\lambda > 0$ modifies equation (4.15) as follows,

$$\left(\tilde{J}^T J + \lambda \, I \right) \Delta \mathbf{x} = \tilde{J}^T \mathbf{b}, \tag{4.16}$$

which is the equivalent version of the approximated 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. This will shift the eigenvalues of the matrix. Also, recall 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.

4.5 Convergence and Least-Square Residual Analysis of nsLSQR

In section 4.2, the general outline of nsLSQR method was presented. The differences between nsLSQR and LSQR were outlined, and the main mathematical equations and procedures were described. However, an important desired property for any numerical solver is the capability of ensuring convergence. This section describe the convergence properties of nsLSQR. In this section we perform two theoretical analyses. Firstly, we provide a convergence analysis for the least-square residual obtained by nsLSQR; and secondly, we show that the least-square residual obtained by nsLSQR is lower or equal compared ton the one obtained by GMRes when is used to solve the equation (4.15). This imply that even if the approximation matrix $\tilde{J}$ is accurate enough to make the linear system (4.15) a reasonable approximation of the normal equations, using nsLSQR converge faster than using GMRes.

Before we begin with the analysis, we need to build an explicit representation of the solution obtained by nsLSQR. Recall that nsLSQR minimizes the least square residual shown in equation (4.12). For completeness, we repeat here a simplified version,

$$\min_{\tilde{\Delta}\mathbf{x}_k=\tilde{V}_k\,\tilde{\mathbf{c}}_k} \|\mathbf{b} - J\,\tilde{\Delta}\mathbf{x}_k\|^2 = \min_{\tilde{\mathbf{c}}_k\in\mathbb{R}^k} \left\| \|\mathbf{b}\|\,\mathbf{e}_1 - \tilde{B}_k\,\tilde{\mathbf{c}}_k \right\|^2.$$

To solve this smaller least-square problem, we consider its reduced QR decomposition of the matrix,

$$\tilde{B}_k = \widehat{Q}_k\,\widehat{R}_k,$$

where $\widehat{Q}_k \in \mathbb{R}^{(k+1)\times k}$ and $\widehat{R}_k \in \mathbb{R}^{k\times k}$. For simplicity, in this section we will omit the super-index $(k+1)$ for the canonical vector $\mathbf{e}_1$. Now, we can obtain the least-square solution, which will be denoted by $\overline{\tilde{\mathbf{c}}_k^{(1)}}$, using the normal equations as follows,

$$\begin{aligned}
\tilde{B}_k^T\tilde{B}_k\tilde{\mathbf{c}}_k^{(1)} &= \|\mathbf{b}\|\tilde{B}_k^T\mathbf{e}_1,\\
\widehat{R}_k^T\widehat{Q}_k^T\widehat{Q}_k\widehat{R}_k\tilde{\mathbf{c}}_k^{(1)} &= \|\mathbf{b}\|\widehat{R}_k^T\widehat{Q}_k^T\mathbf{e}_1.
\end{aligned}$$

If R_k is non-singular, which is the case for a regularized problem, then $\left(R_k^{-1}\right)^T = \left(R_k^T\right)^{-1}$ exist and considering that $\widehat{Q}_k^T\widehat{Q}_k = I$, then

$$\begin{aligned}
\widehat{R}_k^T\widehat{Q}_k^T\widehat{Q}_k\widehat{R}_k\tilde{\mathbf{c}}_k^{(1)} &= \|\mathbf{b}\|\widehat{R}_k^T\widehat{Q}_k^T\mathbf{e}_1,\\
\widehat{R}_k\tilde{\mathbf{c}}_k^{(1)} &= \|\mathbf{b}\|\widehat{R}_k^T\widehat{Q}_k^T\mathbf{e}_1,\\
\tilde{\mathbf{c}}_k^{(1)} &= \|\mathbf{b}\|\widehat{R}_k^{-1}\widehat{Q}_k^T\mathbf{e}_1. \qquad (4.17)
\end{aligned}$$

Here, the super-index "(1)" denotes the solution found by nsLSQR. Later, in the analysis, the super-index "(2)" in the solution will denote the solution obtained by GMRes. This preliminary result gives us the necessary components to continue the analysis.

4.5.1 nsLSQR decreases monotonically the least-square residual

The analysis of the least-square residual can be obtained by direct computation of it. In particular, we evaluate the least-square residual at the least-square solution obtained in equation (4.17). We first take into account equation (4.10), which shows $J\,\tilde{V}_k = \tilde{U}_{k+1}\,\tilde{B}_k$, and then, we re-arrange the terms conveniently as follows,

$$\begin{aligned}
\left\| \|\mathbf{b}\|\,\mathbf{e}_1 - \tilde{B}_k\,\tilde{\mathbf{c}}_k \right\|^2 &= \left\| \|\mathbf{b}\|\,\mathbf{e}_1 - \widehat{Q}_k\,\widehat{R}_k \tilde{\mathbf{c}}_k^{(1)} \right\|^2, \\
&= \left\| \|\mathbf{b}\|\,\mathbf{e}_1 - \widehat{Q}_k\,\widehat{R}_k \left(\|\mathbf{b}\|\,\widehat{R}_k^{-1}\widehat{Q}_k^T\mathbf{e}_1 \right) \right\|^2, \\
&= \left\| \|\mathbf{b}\|\,\mathbf{e}_1 - \|\mathbf{b}\|\,\widehat{Q}_k\,\widehat{R}_k\,\widehat{R}_k^{-1}\widehat{Q}_k^T\mathbf{e}_1 \right\|^2, \\
&= \|\mathbf{b}\|^2 \left\| \mathbf{e}_1 - \widehat{Q}_k\widehat{Q}_k^T\mathbf{e}_1 \right\|^2.
\end{aligned}$$

Note that $\widehat{Q}_k$ is of size $(k+1)\times k$, thus $\widehat{Q}_k\,\widehat{Q}_k^T \neq I_{k+1}$. However, we could obtain the norm-2 by computing the inner product,

$$\begin{aligned}
\left\| \mathbf{e}_1 - \widehat{Q}_k\widehat{Q}_k^T\mathbf{e}_1 \right\|^2 &= \langle \mathbf{e}_1 - \widehat{Q}_k\widehat{Q}_k^T\mathbf{e}_1, \mathbf{e}_1 - \widehat{Q}_k\widehat{Q}_k^T\mathbf{e}_1 \rangle, \\
&= \left(\mathbf{e}_1 - \widehat{Q}_k\widehat{Q}_k^T\mathbf{e}_i \right)^T \left(\mathbf{e}_1 - \widehat{Q}_k\widehat{Q}_k^T\mathbf{e}_1 \right), \\
&= \left(\mathbf{e}_1^T - \mathbf{e}_1^T\widehat{Q}_k\widehat{Q}_k^T \right) \left(\mathbf{e}_1 - \widehat{Q}_k\widehat{Q}_k^T\mathbf{e}_1 \right), \\
&= \|\mathbf{e}_1\|^2 - 2\mathbf{e}_1^T\widehat{Q}_k\widehat{Q}_k^T\mathbf{e}_1 + \mathbf{e}_1^T\widehat{Q}_k\widehat{Q}_k^T\widehat{Q}_k\widehat{Q}_k^T\mathbf{e}_1, \\
&= 1 - 2\left\| \widehat{Q}_k^T\mathbf{e}_1 \right\|^2 + \mathbf{e}_1^T\widehat{Q}_k\widehat{Q}_k^T\mathbf{e}_1, \\
&= 1 - 2\left\| \widehat{Q}_k^T\mathbf{e}_1 \right\|^2 + \left\| \widehat{Q}_k^T\mathbf{e}_1 \right\|^2, \\
&= 1 - \left\| \widehat{Q}_k^T\mathbf{e}_1 \right\|^2 \geq 0.
\end{aligned}$$

Therefore, the residual of nsLSQR is given by,

$$\left\| \mathbf{b} - J\tilde{V}_k\tilde{\mathbf{c}}_k^{(1)} \right\|^2 = \|\mathbf{b}\|^2 \left\| \mathbf{e}_1 - \widehat{Q}_k\widehat{Q}_k^T\mathbf{e}_1 \right\|^2 = \|\mathbf{b}\|^2 \left(1 - \left\| \widehat{Q}_k^T\mathbf{e}_1 \right\|^2 \right). \tag{4.18}$$

Note that the vector $\widehat{Q}_k^T\mathbf{e}_1$ equals the first row of $\widehat{Q}_k$. This row has a new coefficient from the iteration k to $k+1$. So, we have that $\left\| \widehat{Q}_k^T\mathbf{e}_1 \right\|^2 \leq \left\| \widehat{Q}_{k+1}^T\mathbf{e}_1 \right\|^2$. Thus,

$$\|\mathbf{b}\|^2 \left(1 - \left\| \widehat{Q}_{k+1}^T\mathbf{e}_1 \right\|^2 \right) \leq \|\mathbf{b}\|^2 \left(1 - \left\| \widehat{Q}_k^T\mathbf{e}_1 \right\|^2 \right),$$

and so,

$$\left\| \mathbf{b} - J\tilde{V}_{k+1}\tilde{\mathbf{c}}_{k+1}^{(1)} \right\|^2 \leq \left\| \mathbf{b} - J\tilde{V}_k\tilde{\mathbf{c}}_k^{(1)} \right\|^2.$$

Therefore, the residual of nsLSQR decreases monotonically up to the n-th iteration. This means,

$$\left\|\mathbf{b} - JV_n\tilde{\mathbf{c}}_n^{(1)}\right\|^2 \leq \left\|\mathbf{b} - J\tilde{V}_k\tilde{\mathbf{c}}_k^{(1)}\right\|^2, \quad \forall\, k \leq n.$$

Notice that this lower bound for the residual is expected since it is a least-square problem, but the key result is that the least-square residual decreases monotonically. This is also valid for a regularized least-square problem, since it handled as a bigger unregularized least-square.

4.5.2 Comparison of least-square residuals between nsLSQR and GMRes

To perform the comparison between the least-square residual obtained by nsLSQR and GMRes, we will follow a similar approach as the analysis performed before. However, the algebraic manipulation will be different. Initially, we will compute the least-square residual obtained at iteration k by nsLSQR. This will be denoted as Case 1. Later, we will compute the least-square residual for GMRes. This will be denoted as Case 2. In this analysis, we will use the reduced QR decomposition of the matrix $\tilde{B}_k$ and its full QR decomposition. This means we use the following identity:

$$\tilde{B}_k = Q_k\, R_k = \left[\widehat{Q}_k \,|\, \mathbf{q}_{k+1}\right] \begin{bmatrix} \widehat{R}_k \\ \mathbf{0} \end{bmatrix},$$

where $Q_k \in \mathbb{R}^{(k+1)\times(k+1)}$ and $R_k \in \mathbb{R}^{(k+1)\times k}$.

4.5.3 Case 1

For this first case, we again evaluate directly the least-square residual considering the solution obtained by nsLSQR shown in equation (4.17), i.e.

$$\tilde{\mathbf{c}}_k^{(1)} = \widehat{R}_k^{-1}\, \widehat{Q}_k^T\, \|\mathbf{b}\|\, \mathbf{e}_1.$$

Thus, the computation is as follows,

$$\begin{aligned}
\left\|\mathbf{b} - J\tilde{\Delta}\mathbf{x}^{(1)}\right\|^2 &= \left\|\|\mathbf{b}\|\mathbf{e}_1 - \tilde{B}_k\,\tilde{\mathbf{c}}_k^{(1)}\right\|^2, \\
&= \left\|\|\mathbf{b}\|\mathbf{e}_1 - \tilde{B}_k\,\widehat{R}_k^{-1}\,\widehat{Q}_k^T\;\|\mathbf{b}\|\,\mathbf{e}_1\right\|^2, \\
&= \left\|\|\mathbf{b}\|\mathbf{e}_1 - Q_k\,R_k\,\widehat{R}_k^{-1}\,\widehat{Q}_k^T\;\|\mathbf{b}\|\,\mathbf{e}_1\right\|^2, \\
&= \left\|\|\mathbf{b}\|\mathbf{e}_1 - Q_k\begin{bmatrix}\widehat{R}_k\\ \mathbf{0}\end{bmatrix}\widehat{R}_k^{-1}\,\widehat{Q}_k^T\;\|\mathbf{b}\|\,\mathbf{e}_1\right\|^2, \\
&= \left\|\|\mathbf{b}\|\mathbf{e}_1 - Q_k\begin{bmatrix}\widehat{R}_k\widehat{R}_k^{-1}\,\widehat{Q}_k^T\;\|\mathbf{b}\|\,\mathbf{e}_1\\ \mathbf{0}\end{bmatrix}\right\|^2, \\
&= \left\|\|\mathbf{b}\|\mathbf{e}_1 - Q_k\begin{bmatrix}\widehat{Q}_k^T\;\|\mathbf{b}\|\,\mathbf{e}_1\\ \mathbf{0}\end{bmatrix}\right\|^2, \\
&= \left\|Q_k\left(Q_k^T\|\mathbf{b}\|\mathbf{e}_1 - \begin{bmatrix}\widehat{Q}_k^T\;\|\mathbf{b}\|\,\mathbf{e}_1\\ 0\end{bmatrix}\right)\right\|^2, \\
&= \left\|Q_k\left(\begin{bmatrix}\widehat{Q}_k^T\\ \mathbf{q}_{k+1}^T\end{bmatrix}\|\mathbf{b}\|\mathbf{e}_1 - \begin{bmatrix}\widehat{Q}_k^T\;\|\mathbf{b}\|\,\mathbf{e}_1\\ \mathbf{0}\end{bmatrix}\right)\right\|^2, \\
&= \left\|\begin{bmatrix}\widehat{Q}_k^T\|\mathbf{b}\|\mathbf{e}_1\\ \mathbf{q}_{k+1}^T\|\mathbf{b}\|\mathbf{e}_1\end{bmatrix} - \begin{bmatrix}\widehat{Q}_k^T\;\|\mathbf{b}\|\,\mathbf{e}_1\\ \mathbf{0}\end{bmatrix}\right\|^2, \\
&= \left\|\begin{bmatrix}\widehat{Q}_k^T\,(\|\mathbf{b}\| - \|\mathbf{b}\|)\,\mathbf{e}_1\\ \mathbf{q}_{k+1}^T\|\mathbf{b}\|\mathbf{e}_1\end{bmatrix}\right\|^2, \\
&= \|\mathbf{b}\|^2\,(\mathbf{q}_{k+1}^T\mathbf{e}_1)^2. \qquad (4.19)
\end{aligned}$$

Therefore, the least-square residual obtained by nsLSQR can also be represented as $\left\|\mathbf{b} - J\tilde{\Delta}\mathbf{x}^{(1)}\right\|^2 = \|\mathbf{b}\|^2\,(\mathbf{q}_{k+1}^T\mathbf{e}_1)^2$.

4.5.4 Case 2

For this case, we need to obtain first the solution computed by GMRes for equation (4.15), at each iteration k. Fortunately, we have the partial Hessenberg decomposition of $\tilde{J}^T J$, which is shown in equation (4.14). In particular, we observe that at the k-th iteration, GMRes minimizes the following residual:

$$\left\|\tilde{J}^T\mathbf{b} - \tilde{J}^T J\tilde{\Delta}\mathbf{x}^{(2)}\right\|^2,$$

and, since we can represent $\tilde{\Delta}\mathbf{x}_k = \tilde{V}_k\,\tilde{\mathbf{c}}_k^{(2)}$, where $\tilde{\mathbf{c}}_k^{(2)}$ are the coefficients obtained by GMRes, we have the following equivalent residual,

$$\left\|\tilde{J}^T\mathbf{b} - \tilde{J}^T J\tilde{\Delta}\mathbf{x}^{(2)}\right\|^2 = \left\|\left\|\tilde{J}^T\mathbf{b}\right\|\mathbf{e}_1 - \underbrace{\overline{B}_{k+1}\,\tilde{B}_k}_{H_k}\,\tilde{\mathbf{c}}_k^{(2)}\right\|^2,$$

where we use the partial Hessenberg decomposition $\tilde{J}^T J \tilde{V}_k = \tilde{J}^T \tilde{U}_{k+1}\, \tilde{B}_k = \tilde{V}_{k+1}\, \overline{B}_{k+1}\, \tilde{B}_k$. The value of $\tilde{\mathbf{c}}_k^{(2)}$, computed as the least-square solution, is given by,

$$\tilde{\mathbf{c}}_k^{(2)} = \left\| \tilde{J}^T \mathbf{b} \right\| \left(\overline{B}_{k+1}\, \tilde{B}_k \right)^{\dagger} \mathbf{e}_1,$$

where the matrix $\left(\overline{B}_{k+1}\, \tilde{B}_k \right)^{\dagger}$ denotes the Moore-Penrose pseudo-inverse of $\overline{B}_{k+1}\, \tilde{B}_k$. Considering that $\tilde{\mathbf{c}}_k^{(2)}$ is the k-th solution found by GMRes and recalling the full QR Decomposition of $\tilde{B}_k$, then we have the following expression for the residual (4.1),

$$\begin{aligned}
\left\| \mathbf{b} - J\tilde{\Delta}\mathbf{x}^{(2)} \right\|^2 &= \left\| \mathbf{b} - J\tilde{V}_k\, \tilde{\mathbf{c}}_k^{(2)} \right\|^2, \\
&= \left\| \mathbf{b} - \tilde{U}_{k+1}\, \tilde{B}_k\, \tilde{\mathbf{c}}_k^{(2)} \right\|^2, \\
&= \left\| \|\mathbf{b}\| \mathbf{e}_1 - \tilde{B}_k\, \tilde{\mathbf{c}}_k^{(2)} \right\|^2, \\
&= \left\| \|\mathbf{b}\| \mathbf{e}_1 - Q_k R_k\, \tilde{\mathbf{c}}_k^{(2)} \right\|^2, \\
&= \left\| \|\mathbf{b}\| Q\, Q^T \mathbf{e}_1 - Q_k \begin{bmatrix} \widehat{R}_k \\ \mathbf{0}^T \end{bmatrix} \tilde{\mathbf{c}}_k^{(2)} \right\|^2, \\
&= \left\| Q \left(\|\mathbf{b}\| Q^T \mathbf{e}_1 - \begin{bmatrix} \widehat{R}_k \tilde{\mathbf{c}}_k^{(2)} \\ 0 \end{bmatrix} \right) \right\|^2, \\
&= \left\| \|\mathbf{b}\| \begin{bmatrix} \widehat{Q}_k^T \\ \mathbf{q}_{k+1}^T \end{bmatrix} \mathbf{e}_1 - \begin{bmatrix} \widehat{R}_k \tilde{\mathbf{c}}_k^{(2)} \\ 0 \end{bmatrix} \right\|^2, \\
&= \left\| \|\mathbf{b}\| \begin{bmatrix} \widehat{Q}_k^T \mathbf{e}_1 \\ \mathbf{q}_{k+1}^T \mathbf{e}_1 \end{bmatrix} - \begin{bmatrix} \widehat{R}_k \tilde{\mathbf{c}}_k^{(2)} \\ 0 \end{bmatrix} \right\|^2, \\
&= \left\| \begin{bmatrix} \|\mathbf{b}\| \widehat{Q}_k^T \mathbf{e}_1 - \widehat{R}_k \tilde{\mathbf{c}}_k^{(2)} \\ \|\mathbf{b}\| \mathbf{q}_{k+1}^T \mathbf{e}_1 \end{bmatrix} \right\|^2, \\
&= \|\mathbf{b}\|^2 \left(\mathbf{q}_{k+1}^T \mathbf{e}_1 \right)^2 + \left\| \|\mathbf{b}\| \widehat{Q}_k^T \mathbf{e}_1 - \widehat{R}_k \tilde{\mathbf{c}}_k^{(2)} \right\|^2, \\
&= \|\mathbf{b}\|^2 \left(\mathbf{q}_{k+1}^T \mathbf{e}_1 \right)^2 + \left\| \|\mathbf{b}\| \widehat{Q}_k^T \mathbf{e}_1 - \left\| \tilde{J}^T \mathbf{b} \right\| \widehat{R}_k \left(\overline{B}_{k+1}\, \tilde{B}_k \right)^{\dagger} \mathbf{e}_1 \right\|^2.
\end{aligned}$$

Therefore, the least-square residual obtained by GMRes is

$$\left\| \mathbf{b} - J\tilde{\Delta}\mathbf{x}^{(2)} \right\|^2 = \|\mathbf{b}\|^2 \left(\mathbf{q}_{k+1}^T \mathbf{e}_1 \right)^2 + \left\| \|\mathbf{b}\| \widehat{Q}_k^T \mathbf{e}_1 - \left\| \tilde{J}^T \mathbf{b} \right\| \widehat{R}_k \left(\overline{B}_{k+1}\, \tilde{B}_k \right)^{\dagger} \mathbf{e}_1 \right\|^2. \tag{4.20}$$

4.5.5 Direct comparison of the least-square residual obtained by nsLSQR and GMRes

Equations (4.19) and (4.20) show the explicit least-square residual for nsLSQR and GMRes. In particular, we obtain the following quotient,

$$\begin{aligned}\frac{\left\|\mathbf{b} - J\tilde{\Delta}\mathbf{x}_k^{(2)}\right\|^2}{\left\|\mathbf{b} - J\tilde{\Delta}\mathbf{x}_k^{(1)}\right\|^2} &= \frac{\|\mathbf{b}\|^2 \left(\mathbf{q}_{k+1}^T\mathbf{e}_1\right)^2 + \left\|\|\mathbf{b}\|\widehat{Q}_k^T\mathbf{e}_1 - \widehat{R}_k\tilde{\mathbf{c}}_k^{(2)}\right\|^2}{\|\mathbf{b}\|^2 \left(\mathbf{q}_{k+1}^T\mathbf{e}_1\right)^2} \\ &= 1 + \frac{\left\|\|\mathbf{b}\|\widehat{Q}_k^T\mathbf{e}_1 - \widehat{R}_k\tilde{\mathbf{c}}_k^{(2)}\right\|^2}{\|\mathbf{b}\|^2 \left(\mathbf{q}_{k+1}^T\mathbf{e}_1\right)^2} \\ &= 1 + \rho_k \\ &\geq 1\end{aligned}$$

Re-arranging, we obtain,

$$\left\|\mathbf{b} - J\tilde{\Delta}\mathbf{x}_k^{(2)}\right\|^2 = (\rho_k + 1)\left\|\mathbf{b} - J\tilde{\Delta}\mathbf{x}_k^{(1)}\right\|^2.$$

Therefore, we can conclude that the residual obtained by nsLSQR is less than or equal to the residual computed by GMRes.

4.6 Implementation notes in nsLSQR

From Algorithm 2, it can be seen that nsLSQR does not have special procedures or routines that requires an special implementation. Having "external" procedures to compute an approximation for $J(\mathbf{x}_i)\mathbf{v}$ and $J^T(\mathbf{x}_i)\mathbf{w}$ are the main requirements of nsLSQR. However, two important topics need to be described here for completeness:

- How to efficiently solve the inner least-square problem found in nsLSQR, i.e. how to efficiently perform the line 8 of Algorithm 2.
- What are the stopping criteria used in nsLSQR.

4.6.1 Efficient solution for the inner least-square problem

Line 8 of Algorithm 2 requires the solution of a $k+1 \times k$ linear least-square problem, given by (4.12). If we compute the QR decomposition Sauer 2011 of $\tilde{B}_k = Q_k R_k$ with Q_k of size $k+1 \times k$ and R_k of size $k \times k$, we have that the solution of the least square problem is given by the solution of the following linear system of equation:

$$R_k\,\tilde{\mathbf{c}}_k = \|\mathbf{b}\|Q_k^T\mathbf{e}_1,$$

as shown in (4.17). Since R_k is an upper triangular matrix, its solution can be computed efficiently using a *backward substitution* approach Sauer 2011. Note also

that the product $Q_k^T \mathbf{e}_1$ is the first row of Q_k, so the matrix-vector product does not need to be computed.

To obtain an efficient QR decomposition for the current matrix $\tilde{B}_k$, let suppose that we have the QR decomposition for the previous matrix, i.e. $\tilde{B}_{k-1} = Q_{k-1}R_{k-1}$. From (4.9) and the equality $J\tilde{V}_k = \tilde{U}_{k+1}\tilde{B}_k$ we find that the structure of the matrix $\tilde{B}_k$ is the following:

$$\tilde{B}_k = \begin{bmatrix} \tilde{B}_{k-1} & \mathbf{b}_k \\ \mathbf{0}_{k-1}^T & \end{bmatrix},$$

where

$$\mathbf{b}_k = \begin{bmatrix} \beta_{1,k} \\ \beta_{2,k} \\ \vdots \\ \beta_{k,k} \\ \beta_{k+1,k} \end{bmatrix} \in \mathbb{R}^{k+1},$$

where the coefficients $\beta_{i,k}$ are obtained from (4.9). That is, the submatrix from the first to the k-th row and from the first to the $k-1$-th column of $\tilde{B}_k$ is $\tilde{B}_{k-1}$ and the last row from the first to the $k-1$-th column are zero. Therefore, the new part from $\tilde{B}_{k-1}$ to $\tilde{B}_k$ is the last column of $\tilde{B}_k$, i.e. $\mathbf{b}_k$. Therefore, the QR decomposition of $\tilde{B}_k$ has the following structure

$$\tilde{B}_k = \begin{bmatrix} \tilde{B}_{k-1} & \mathbf{b}_k \\ \mathbf{0}_{k-1}^T & \end{bmatrix} = \begin{bmatrix} Q_{k-1}R_{k-1} & \mathbf{b}_k \\ \mathbf{0}_{k-1}^T & \end{bmatrix} = \begin{bmatrix} Q_{k-1} & \mathbf{q}_k \\ \mathbf{0}_{k-1}^T & \end{bmatrix} \begin{bmatrix} R_{k-1} & \mathbf{r}_k \\ \mathbf{0}_{k-1}^T & \end{bmatrix}.$$

Hence we only need to compute $\mathbf{r}_k$ and $\mathbf{q}_k$, that is, the new column of Q_k and R_k. Note that the first $k-1$ columns of Q_k are the matrix Q_{k-1} and a rows of zeroes, while the first $k-1$ columns of R_k are the matrix R_{k-1} and a rows of zeroes. For simplicity, our implementation use the modified Gram-Schmidt method but in general any orthogonalization method can be used. Let $r_{i,k}$ be the i-th coefficient of $\mathbf{r}_k$. Then $\mathbf{r}_k$ and $\mathbf{q}_k$ are found by

$$\mathbf{b}_k = \sum_{i=1}^{k} r_{i,k}\mathbf{q}_i,$$

where $\mathbf{q}_i$, for $i \in \{1, 2, \ldots, k-1\}$ is the vector composed by the i-th column of Q_{k-1} and a zero as the last coefficient.

This approach requires the previous QR decomposition. This decomposition for the first matrix, $\tilde{B}_1$ is easy to compute, since

$$\tilde{B}_1 = \begin{bmatrix} \beta_{1,1} \\ \beta_{2,1} \end{bmatrix},$$

therefore, its QR decomposition is given by

$$Q_1 = \begin{bmatrix} \frac{\beta_{1,1}}{\sqrt{\beta_{1,1}^2+\beta_{2,1}^2}} \\ \frac{\beta_{2,1}}{\sqrt{\beta_{1,1}^2+\beta_{2,1}^2}} \end{bmatrix},$$
$$R_1 = \begin{bmatrix} \sqrt{\beta_{1,1}^2 + \beta_{2,1}^2} \end{bmatrix}.$$

The Algorithm 3 shows how to compute the new QR decomposition, given the previous one and the new column of $\tilde{B}_k$, i.e. $\mathbf{b}_k$. The notation $A[a:b, c:d]$ denotes the submatrix of A from the row a to the row b and from the column c to the column d. This notation is introduced to denote slicing over matrices easily.

Algorithm 3 Compute the new QR decomposition

Require: The new column $\mathbf{b}_k$ of $\tilde{B}_k$, the previous QR decomposition Q_{k-1} and R_{k-1}, the current iteration k.
Ensure: The new QR decomposition Q_k and R_k.
1: Build a matrix Q_k of size $k+1 \times k$ with all its coefficients equal to zero.
2: Build a matrix R_k of size $k \times k$ with all its coefficients equal to zero.
3: $Q_k[1:k\,,\,1:k-1] := Q_{k-1}$.
4: $R_k[1:k-1\,,\,1:k-1] := R_{k-1}$.
5: $\mathbf{y} := \mathbf{b}_k$.
6: **for** i from 1 to $k-1$ **do**
7: $\quad r_{i,k} := \langle \mathbf{y}, \mathbf{q}_i \rangle$, where $\mathbf{q}_i$ denotes the i-th column of Q_k.
8: $\quad \mathbf{y} := \mathbf{y} - r_{i,k}\mathbf{q}_i$.
9: **end for**
10: $r_{k,k} := \|\mathbf{y}\|$.
11: $\mathbf{q}_k = \frac{\mathbf{y}}{r_{k,k}}$.
12: Return Q_k and R_k.

Now that the update of the QR decomposition is available, the Algorithm 4 shows how to solve the inner least-square (4.12). It requires, as input, the QR decomposition of $\tilde{B}_{k-1}$, the vector $\mathbf{b}_k$ and the scalar $\|\mathbf{b}\|$. It returns the solution $\tilde{\mathbf{c}}_k$.

Algorithm 4 Algorithm for efficient solution of inner least-square in nsLSQR.

Require: The new column $\mathbf{b}_k$ of $\tilde{B}_k$, the previous QR decomposition Q_{k-1} and R_{k-1}, the scalar $\|\mathbf{b}\|$ and the current iteration k.
Ensure: The new QR decomposition Q_k and R_k, and the solution $\tilde{\mathbf{c}}_k$.
1: Use Algorithm 3 to compute Q_k and R_k from $\mathbf{b}_k$, Q_{k-1}, R_{k-1} and the current iteration k.
2: Obtain the first row of Q_k, denoted as $\mathbf{q}$.
3: Solve $R_k\,\tilde{\mathbf{c}}_k = \|\mathbf{b}\|\mathbf{q}$ using backward substitution.
4: Return Q_k, R_k and $\tilde{\mathbf{c}}_k$.

4.6.2 Stopping criteria of nsLSQR

For general linear least-square problem, the stopping criteria are usually the same and well-know. When solving (4.1), an intuitive option will be to measure the following residual

$$\|\mathbf{r}_k\| = \|\mathbf{b} - J\Delta\mathbf{x}_k\|^2,$$

at every iteration of the least-square solver, where $\Delta\mathbf{x}_k$ correspond to the k-th solution found in the algorithm. However, note that the solution of (4.1) does not need to be zero at the minimum. The value of the residual at the optimum value is completely problem-dependent. In exact arithmetic, we know that the problem (4.1) is equivalent to its normal equations, which is a linear system of equations. Therefore, the residual

$$\left\|\tilde{\mathbf{r}}_k\right\| = \left\|J^T\left(\mathbf{b} - J\Delta\mathbf{x}_k\right)\right\|, \tag{4.21}$$

is zero at its solution. The residual given by (4.21) is the standard stopping criteria for least-square problems, although usually is not computed directly but using alternative and more stable expressions.

Unfortunately, in nsLSQR such residual cannot be computed easily. Note that in nsLSQR we do not use J^T but an approximation procedure or matrix, denoted as $\tilde{J}$. However, the precision of the approximation is not previously known so the computation of $\tilde{\mathbf{r}}_k$ is not feasible since replacing J^T by the approximation does not guarantee that the computed residual is close to the original residual. A matrix-free method can be used, but its high computational cost also makes this alternative unfeasible since the residual is computed at each iteration. A different stopping criteria must be used in nsLSQR to be computationally feasible.

The following stopping criteria are used in our implementation of nsLSQR:

- Although we do know that the residual $\|\mathbf{b} - J\Delta\mathbf{x}_k\|^2$ does not need to be zero, we use it as an stopping criteria. This is mostly when we are solving a problem that is actually zero-residual or when the user know the actual optimal residual or a close value. For this criteria, a tolerance η_t is required.

- A second stopping criteria is when the computed solution, $\Delta\mathbf{x}_k$, does not vary significantly from iteration to iteration. That is, given a tolerance η_r and an integer $r > 0$, we test the following expression,

  $$\frac{\|\Delta\mathbf{x}_k - \Delta\mathbf{x}_{k-1}\|}{\|\Delta\mathbf{x}_k\|} < \eta_r,$$

  at each iteration. If the inequality holds for r continuously iterations, nsLSQR stops and returns the current computed solution.

- When the computed solution is close to the actual solution, the progress from iteration to iteration is reduced significantly. This can be seen as a saturation or stagnation of the residual. Then, the last stopping criteria used in our implementation is to stop the method when the saturation has been achieved.

To measure this saturation, we require two parameters: a tolerance η_s and an integer $s > 0$. The method compute a linear regression of the last s residuals $\|\mathbf{b} - J\Delta\mathbf{x}_i\|^2$, for $i \in \{k-s, k-s+1, \ldots, k-1, k\}$. Then, the slope of this linear regression is compared against the tolerance η_s. If the slope is less than the tolerance, then the method stop and returns the current solution.

4.6.3 Final algorithm of nsLSQR

For completeness, the Algorithm 5 is presented. This algorithm contains all the key elements described in this chapter, included the implementation details and also the stopping criteria. It is mainly based on the Algorithm 2 but with the implementation details described.

Algorithm 5 nsLSQR algorithm

Require: A function $\mathbf{F}$, a vector $\mathbf{x}_i$ to evaluate the Jacobian matrix, a stepsize $\epsilon > 0$ to be used in the finite-difference approximation, an approximation matrix $\tilde{J}$, a vector $\mathbf{b}$, tolerance η_t, η_r and η_s, integers r and s, a number of iterations t_{in} and t_{out} to control the inner and outer restarts, an initial guess $\mathbf{x}_0$ which usually is set to be the zero vector

Ensure: An approximated solution $\Delta\mathbf{x}$.

for i from 1 to t_{out} **do**
  $\mathbf{r} := \mathbf{b} - J\mathbf{x}_0$ using a finite-difference approximation.
  $\tilde{\mathbf{u}}_1 := \frac{\mathbf{r}}{\|\mathbf{r}\|}$
  $\tilde{\mathbf{v}}_1 := \frac{\tilde{J}^T\mathbf{r}}{\|\tilde{J}^T\mathbf{r}\|}$
  for k from 1 to t_{in} **do**
    Compute $\tilde{\mathbf{u}}_{k+1}$ and $\mathbf{b}_k$ from (4.9) using a modified Gram-Schmidt.
    Compute $\tilde{\mathbf{v}}_{k+1}$ from (4.11) using a modified Gram-Schmidt.
    if k is 1 **then**
      Build the initial QR decomposition, Q_1 and R_1
      Compute $\tilde{\mathbf{c}}_1 = \|\mathbf{b}\| R_1^{-1} Q_1^T \mathbf{e}_1^{(2)}$ by direct computation.
    else
      Solve the inner least-square using the Algorithm 4 with input $\mathbf{b}_k$, Q_{k-1}, R_{k-1}, the scalar $\|\mathbf{b}\|$ and the current iteration k. The vector $\tilde{\mathbf{c}}_k$ is obtained.
    end if
    $\Delta\mathbf{x}_k := \tilde{V}_k\, \tilde{\mathbf{c}}_k$.
    if Some of the stopping criteria is meet for η_t, η_r and r, or η_s and s **then**
      Returns $\Delta\mathbf{x}_k$.
    end if
  end for
  $\mathbf{x}_0 := \mathbf{x}_0 + \Delta\mathbf{x}$
end for
Returns $\Delta\mathbf{x}_k$.

4.7 Memory usage and complexity of nsLSQR

The proposed nsLSQR method was already described and its main algorithm was presented in previous section. The missing description is the total memory required by nsLSQR and its computational complexity.

4.7.1 Memory usage

The nsLSQR method, in contrast with LSQR, requires some dense matrices to be stored. In LSQR, due to the bidiagonal property over its matrices, the storage is very low. Since in nsLSQR the bidiagonal property is loss, the matrices are dense. To have control over the memory usage of nsLSQR, two parameters were introduced in the nsLSQR algorithms: t_{out} and t_{in}. The t_{in} parameter controls how far the basis produced by $\tilde{V}_k$ will be extended. Once t_{in} iterations has been executed, the computed matrices are discarded. This process is repeated t_{out} times. This process is called *restarting* and is a classical approach to avoid an excessive usage of memory of some Krylov subspace methods Saad 2003. When $t_{\text{out}} = 1$ and $t_{\text{in}} = n$, a full iteration is performed and the solution is guarantee to be obtained, in exact arithmetic. The reason of this is that $\tilde{V}_n$ is a basis for $\mathbb{R}^n$ and the solution must reside in that space. However, this configuration is unfeasible for large-scale problem since the matrix $\tilde{U}_n$ is of size $m \times n$, which is the same dimension for a full Jacobian matrix and under our assumptions, that storage is prohibited. For this reason, a different configuration is required for large-scale problems. However, convergence is not ensured.

In either case, for general values of t_{out} and t_{in}, and considering only the larger structures in nsLSQR, the memory requirements are mainly based on the following components:

- The matrix $\tilde{U}$ which is of size $m \times t_{\text{in}}$. Since the inner iteration is at most t_{in}, then a bigger matrix $\tilde{U}$ is not required.
- The matrix $\tilde{V}$ of size $n \times t_{\text{in}}$.
- The matrix $\tilde{B}$ of size $t_{\text{in}} + 1 \times t_{\text{in}}$.
- The matrices for the QR decomposition. The matrix Q is of size $t_{\text{in}} + 1 \times t_{\text{in}}$ while the matrix R is of size $t_{\text{in}} \times t_{\text{in}}$.
- The storage of the approximation matrix. Let denote the memory usage of the approximation matrix as ξmn, where $\xi < 1$ represent the compression or reduction in the memory usage, with respect to a full Jacobian matrix.

Note that the memory requirements for $\tilde{B}$, Q and R is given by,

$$(t_{\text{in}} + 1)t_{\text{in}} + (t_{\text{in}} + 1)t_{\text{in}} + t_{\text{in}}^2 = 3t_{\text{in}}^2 + 2t_{\text{in}}.$$

Then, considering all the previous matrices, the complexity for memory usage in nsLSQR is given by

$$\mathcal{O}(mt_{\text{in}} + nt_{\text{in}} + 3t_{\text{in}}^2 + 2t_{\text{in}}^2 + \xi mn).$$

When solving a large-scale problem and the nsLSQR method is going to be used, these parameters and configuration must be selected carefully to have enough memory for all the components.

4.7.2 Computational complexity

To compute the complexity of nsLSQR, let f be the number of elemental operations required to evaluate $\mathbf{F}$. To make the expression more general, let suppose that h is the number of elemental operations required to perform the product $\tilde{J}^T\mathbf{w}$. Then, we have the following items to consider for the computational complexity:

- The computation of $J\Delta\mathbf{x}$.
- The computation of $\tilde{\mathbf{u}}_1$.
- The computation of $\tilde{\mathbf{v}}_1$.
- The usage of the modified Gram-Schmidt to compute $\tilde{\mathbf{u}}_{k+1}$ and $\tilde{\mathbf{v}}_{k+1}$.
- The solution of the reduced inner least-square problem.
- Compute the current solution $\Delta\mathbf{x}_k = \tilde{V}_k\,\tilde{\mathbf{c}}_k$.
- The update of the solution for the next restart.

The computation of $J\Delta\mathbf{x}$ is done using a second-order finite-difference approximation,

$$J\Delta\mathbf{x} \approx \frac{\mathbf{F}(\mathbf{x}_i + \varepsilon\Delta\mathbf{x}) - \mathbf{F}(\mathbf{x}_i - \varepsilon\Delta\mathbf{x})}{2\varepsilon}.$$

This requires $2(m+n+f)+3$ elemental operations. The computation of $\tilde{\mathbf{u}}_1$ requires $2m$ elemental operations, since a norm computation and then a division over each component of the vector is needed. For $\tilde{\mathbf{v}}_1$, $h+2n$ elemental operations are required. For the last two items, the computation of $\Delta\mathbf{x}_k = \tilde{V}_k\,\tilde{\mathbf{c}}_k$ requires nk elemental operations, while the update for the next solution requires n elemental operations.

Then, we need the elemental operation for the usage of the Gram-Schmidt to compute $\tilde{\mathbf{u}}_{k+1}$ and $\tilde{\mathbf{v}}_{k+1}$. Let us begin first with the $\tilde{\mathbf{u}}_{k+1}$ vector computation. The expression used to compute $\tilde{\mathbf{u}}_{k+1}$ is given by (4.9), which is

$$J\tilde{\mathbf{v}}_k = \sum_{i=1}^{k} \beta_{i,k}\tilde{\mathbf{u}}_i + \beta_{k+1,k}\tilde{\mathbf{u}}_{k+1}.$$

Computing $J\tilde{\mathbf{v}}_k$ requires $2(m+n+f)+3$ elemental operations, as mentioned previously. Since we are using the modified Gram-Schmidt method, we set $\mathbf{y}$ as $J\tilde{\mathbf{v}}_k$. The first value $\beta_{1,k}$ is computed as $\langle \mathbf{y}, \tilde{\mathbf{u}}_1 \rangle$, which requires m elemental operations. Then, $\mathbf{y}$ is updated as $\mathbf{y} := \mathbf{y} - \beta_{1,k}\tilde{\mathbf{u}}_1$, which requires $2m$ elemental operations. The second values are computed as $\beta_{2,k} = \langle \mathbf{y}, \tilde{\mathbf{u}}_2 \rangle$ and the update is computed as $\mathbf{y} := \mathbf{y} - \beta_{2,k}\tilde{\mathbf{u}}_2$. This also requires $2m + m = 3m$ elemental operations. This process is repeated k times, giving a total of $3km$ elemental operations to compute the values of $\beta_{i,k}$ for $i \in \{1, 2, \ldots, k\}$ and the updates. Then, the computation of $\beta_{k+1,k} = \|\mathbf{y}\|$ requires m elemental operations. Finally, the computation of the new vector $\tilde{\mathbf{u}}_{k+1}$ is given by

$$\tilde{\mathbf{u}}_{k+1} = \frac{\mathbf{y}}{\beta_{k+1,k}},$$

which requires m elemental operations. Therefore, the total count of elemental operations to compute all the values of $\beta_{i,k}$ and the new vector $\tilde{\mathbf{u}}_{k+1}$ is

$$2(m+n+f)+3+3mk+2m = 2m+2n+2f+3+3mk+2m = m(3k+4)+2n+2f+3.$$

The process to compute $\tilde{\mathbf{v}}_{k+1}$ is similar, but instead of $2(m+n+f)+3$ elemental operations initially, we require h due to the use of the approximation matrix. This gives a total of $h + 3mk + 2m = h + m(3k+2)$ elemental operations.

For the inner least-square problem, we observe the Algorithm 4. We require the computational complexity of the computation of the new QR decomposition from the previous one and the cost of solving the linear system of equations by backward substitution. We can use a similar argument for the modified Gram-Schmidt method than the used for $\tilde{\mathbf{u}}_{k+1}$ and $\tilde{\mathbf{v}}_{k+1}$ and observing the Algorithm 3. Since the current matrix $\tilde{B}_k$ is of size $k+1 \times k$, we have that the computation of the new QR decomposition requires $3(k+1)(k-1) + 2(k+1) = 3(k^2-1) + 2k + 1$ elemental operations. Combined with the number of elemental operations requires to solve a linear problem by means of a backward substitution Sauer 2011, which is k^2 elemental operations, we have a total of $3(k^2-1) + 2k + 1 + k^2 = 4k^2 + 2k - 1$ elemental operations.

Now all the previous elemental operations need to be considered for t_{out} and t_{in}. In particular, we have

- Since the line 2 of Algorithm 5 is executed t_{out} times, we have a total of $t_{\text{out}}(3m + 2n + 2f + 3)$, considering the difference with vectors.
- The vector $\tilde{\mathbf{u}}_1$ is created t_{out} times, so we have a total of $2t_{\text{out}}m$ elemental operations.
- The vector $\tilde{\mathbf{v}}_1$ is created t_{out} times, so we have a total of $t_{\text{out}}(h + 2n)$ elemental operations.
- The current solution $\Delta\mathbf{x}_k$ is computed at each t_{in} iteration and its complexity depends on k. Also this is executed t_{out} times. Therefore, its number of

elemental operations is given by

$$t_{\text{out}} \sum_{k=1}^{t_{\text{in}}} nk = t_{\text{out}} n \frac{t_{\text{in}}(t_{\text{in}}+1)}{2} = \frac{n t_{\text{out}} t_{\text{in}}(t_{\text{in}}+1)}{2}.$$

- The update in line 19 of Algorithm 5 is repeated t_{out} times, giving a total count of nt_{out} elemental operations.
- The computation of $\tilde{\mathbf{u}}_{k+1}$ is realized for $k \in \{1, 2, \ldots, t_{\text{in}}\}$. This process is executed t_{out} times. Its number of elemental operations is given by

$$\begin{aligned} t_{\text{out}} \sum_{k=1}^{t_{\text{in}}} (3mk + 4m + 2n + 2f + 3) &= 3mt_{\text{out}} \sum_{k=1}^{t_{\text{in}}} k + t_{\text{out}} t_{\text{in}} (4m + 2n + 2f + 3), \\ &= 3mt_{\text{out}} \frac{t_{\text{in}}(t_{\text{in}}+1)}{2} + \\ &\quad t_{\text{out}} t_{\text{in}} (4m + 2n + 2f + 3). \end{aligned}$$

- A similar idea follows for the computation of $\tilde{\mathbf{v}}_{k+1}$. Its number of elemental operations is given by

$$\begin{aligned} t_{\text{out}} \sum_{k=1}^{t_{\text{in}}} (3km + h + 2) &= t_{\text{out}} t_{\text{in}}(h+2) + 3mt_{\text{out}} \sum_{k=1}^{t_{\text{in}}} k, \\ &= t_{\text{out}} t_{\text{in}}(h+2) + 3mt_{\text{out}} \frac{t_{\text{in}}(t_{\text{in}}+1)}{2}. \end{aligned}$$

- Finally, the inner least-square is solved for k from 1 to t_{in}. This process repeated t_{out} times. Its number of operations is given by

$$\begin{aligned} t_{\text{out}} \sum_{k=1}^{t_{\text{in}}} \left(4k^2 + 2k - 1\right) &= t_{\text{out}} \left(4 \sum_{k=1}^{t_{\text{in}}} k^2 + 2 \sum_{k=1}^{t_{\text{in}}} k - \sum_{k=1}^{t_{\text{in}}} 1 \right), \\ &= t_{\text{out}} \left(4 \frac{t_{\text{in}}(t_{\text{in}}+1)(2t_{\text{in}}+1)}{6} + 2 \frac{t_{\text{in}}(t_{\text{in}}+1)}{2} - t_{\text{in}} \right), \\ &= t_{\text{out}} \left(\frac{4}{3} t_{\text{in}}^3 + 3t_{\text{in}}^2 + \frac{2}{3} t_{\text{in}} \right). \end{aligned}$$

The final expression, which will be shown directly to avoid a large expression, is given by,

$$\mathcal{O}\left(t_{\text{out}} \left(2f(t_{\text{in}}+1) + h(1+t_{\text{in}}) + 3mt_{\text{in}}^2 + \frac{4}{3} t_{\text{in}}^3 \right) \right).$$

This expression was obtained by summing all the previous expression, i.e. the computational cost of $\tilde{\mathbf{u}}_1$ and $\tilde{\mathbf{v}}_1$, the computation of the current solution $\Delta \mathbf{x}_k$, the computation of $\tilde{\mathbf{u}}_{k+1}$ and $\tilde{\mathbf{v}}_{k+1}$, and the solution of the inner-least square problem.

Chapter 5

Quantization

In chapter 4 the nsLSQR method was explained. As shown, it can solve a linear least-square problem only using an approximation for the product $J^T(\mathbf{x}_i)\,\mathbf{w}$. In this chapter, an approximation strategy will be presented, with the aim of using it as an approximation matrix for nsLSQR. This approximation proposal is based on a quantized approximation of the matrix. This quantized approximation builds a restricted-memory approximation of the Jacobian matrix, using quantization to reduce the memory usage of the coefficients of the matrix. Later, its transpose is used to approximate $J^T(\mathbf{x}_i)\,\mathbf{w}$. This restricted-memory approximation is represented as an explicitly stored matrix whose coefficients use an user-specified number of bits. This number of bits will define the parameters required by our quantization procedure. The usage of a small number of bits will produce a matrix whose storage requirements are considerably less than a full double precision matrix. However, using an small number of bits may affect the performance of the matrix in the transpose of the matrix-vector product, i.e. increase its error. This implies that our approximation cannot be used with classical algorithms, for instance, LSQR. For this purpose, the nsLSQR was constructed.

Following the terminology used in Sayood 2006, our method falls in the *Scalar Uniform Midterm Quantization* category, which means that individual elements in the structure will be quantized. More specific, our method quantize each of the coefficients of a vector. Then, a matrix is quantized column-by-column in a systematic procedure.

5.1 Quantization approximation

5.1.1 Scalar quantization of a vector

Let $\mathbf{p} \in \mathbb{R}^m$ be a vector whose coefficients are given by p_i, for $i \in \mathcal{I}_m = \{1, 2, \ldots, m\}$. Let P be the maximum coefficient of $\mathbf{p}$ in absolute value, i.e. $P = \max_{i \in \mathcal{I}_m} |p_i|$. We have that each element in $\mathbf{p}$ is given by the following expression

$$p_i = 2Px_i - P = P(2x_i - 1), \quad x_i \in [0, 1].$$

Let suppose that we want to approximate (quantize) the vector $\mathbf{p}$ using b bits. This imply that we just have 2^b possible values to store in memory. From the 2^b available numbers to use, we need one to represent the zero value. Although this is not mandatory, it is desired to be able to represent accurately sparse matrices. If zero is not representable in our representation, sparse matrices will be poorly approximated by the quantized method. To get an even distribution of values among positives and negatives values, we need $2^{b-1}-1$ values to represent positive numbers and $2^{b-1}-1$ for negative numbers. Since $2^{b-1}-1+2^{b-1}-1+1=2^b-1$, then one value will be discarded from the total of available numbers, using b bits. Using the following change of variables

$$y_i = (2^b-2)x_i = 2(2^{b-1}-1)x_i \Leftrightarrow x_i = \frac{y_i}{2(2^{b-1}-1)}, \quad y_i \in [0, 2(2^{b-1}-1)],$$

we have that

$$p_i = \frac{2P}{2(2^{b-1}-1)}y_i - P = \frac{P}{2^{b-1}-1}y_i - P = \frac{P}{2^{b-1}-1}\left(y_i - \left(2^{b-1}-1\right)\right).$$

Setting

$$s = 2^{b-1}-1$$

and

$$d = \frac{P}{s},$$

we have that

$$p_i = d\left(y_i - s\right), \tag{5.1}$$

for $y_i \in [0, 2(2^{b-1}-1)] = [0, 2^b-2]$.

Equation (5.1) is valid for all $i \in I_m$ since d and s does not depend of i and y_i is the independent variable to match p_i. Note also that y_i is defined to range from 0 to $2^b-2 = 2(2^{b-1}-1)$. The quantization proposal is to approximate p_i rounding y_i to its nearest integer. We will denote such rounded value as $\tilde{y}_i$. The definition of $y_i \in [0, 2^b-2]$ ensure that $\tilde{y}_i \in [0, 1, \ldots, 2^b-2]$. Therefore, the value of $\tilde{y}_i$ is representable using b bits.

Finally, the proposed quantization approximation is given by

$$p_i \approx \tilde{p}_i = d\left(\tilde{y}_i - s\right), \tag{5.2}$$

where $\tilde{p}_i$ denotes the approximation of p_i. Note that the value of y_i is given by

$$p_i = d(y_i - s) \Leftrightarrow y_i = \frac{p_i}{d} + s.$$

The Algorithm 6 describe the quantization procedure defined previously. The algorithm shows a procedure to approximate a vector $\mathbf{p}$ and return the scaling factor d, the integer vector $\tilde{\mathbf{y}}$ and the shift s.

Algorithm 6 Algorithm to approximate a vector using scalar quantization

Require: A vector $\mathbf{p}$ of size m, a number of bits b

Ensure: A scalar d, a shift s and a vector $\tilde{\mathbf{y}}$, where $\mathbf{p} \approx d(\tilde{\mathbf{y}} - s\mathbf{1}_m)$

Compute the max value $P = \arg\max_{i \in \mathcal{I}_m} |p_i|$.
Set $s = 2^{b-1} - 1$.
Set $d = \frac{P}{s}$
Compute $\mathbf{y} = \frac{1}{d}\mathbf{p} + s\mathbf{1}_m$
Compute $\tilde{\mathbf{y}}$ rounding $\mathbf{y}$ to the nearest integer for each of its coefficients.
Return d, $\tilde{\mathbf{y}}$ and s.

The error of the approximation depends on the error produced by rounding to the nearest integer and scaled for d, since

$$\begin{aligned} |e_i| &= |p_i - d(\tilde{y}_i - s)|\,, \\ &= |d(y_i - s) - d(\tilde{y}_i - s)|\,, \\ &= |d(y_i - s - \tilde{y}_i) + s|\,, \\ &= |d(y_i - \tilde{y}_i)|\,, \\ &= d\,|y_i - \tilde{y}_i|\,, \\ &\leq \frac{d}{2}. \end{aligned}$$

Since $d = \frac{P}{s} = \frac{P}{2^{b-1}-1}$, this shows that the error is reduced increasing b (equivalent to having more quantization values) and having an small P. The latter is not relevant since it depends on the input vector to quantize.

5.1.2 Initial approach for matrix approximation

The idea of our proposal is to approximate a matrix $A \in \mathbb{R}^{m \times n}$ producing a matrix $\tilde{A}_b \in \mathbb{R}^{m \times n}$ where each of its coefficients use b bits. The matrix $\tilde{A}_b$ will be represented as

$$\tilde{A}_b = T_b\, D_b, \tag{5.3}$$

where T_b is a matrix of size $m \times n$ and D_b is a diagonal matrix of size $n \times n$. The idea of this description is that easily represent the quantization approximation for vectors given by the Algorithm 6, where the vector to quantize are the columns of the matrix A. Let $\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n$ be the columns of A. Then using the Algorithm 6 gives

$$\begin{aligned} \mathbf{a}_1 &\approx \tilde{\mathbf{a}}_1 = d_1(\tilde{\mathbf{y}}_1 - s\mathbf{1}_m), \\ \mathbf{a}_2 &\approx \tilde{\mathbf{a}}_2 = d_2(\tilde{\mathbf{y}}_2 - s\mathbf{1}_m), \\ &\vdots \quad\quad \vdots \\ \mathbf{a}_n &\approx \tilde{\mathbf{a}}_n = d_n(\tilde{\mathbf{y}}_n - s\mathbf{1}_m). \end{aligned}$$

Note that since all the columns of A are quantized using b bits, s is equal in each approximation. The scaling factor, d_i, is different for each column since it depends on the value of its maximum coefficient in absolute value. Then, A is approximated as

$$A \approx \tilde{A}_b = \left(M_b - s\mathbf{1}_m\mathbf{1}_n^T\right) D_b, \tag{5.4}$$

where

$$D_b = \begin{bmatrix} d_1 & 0 & \dots & 0 \\ 0 & d_2 & \dots & 0 \\ \vdots & \vdots & \dots & \vdots \\ 0 & 0 & \dots & d_n \end{bmatrix},$$

and the k-th column of M_b is $\tilde{\mathbf{y}}_k$. The effect of $s\mathbf{1}_m\mathbf{1}_n^T$ is to subtract s for all the coefficients of the matrix. Setting $T_b = \left(M_b - s\mathbf{1}_m\mathbf{1}_n^T\right)$ match (5.4) with (5.3).

It is important to note that our main idea to develop this approximation method is to reduce the memory requirements to store a matrix, in particular, the Jacobian matrix. The described approach achieve this considering that

- The diagonal matrix D does not need to be stored explicitly. It may be stored as a vector.

- The shift matrix $s\mathbf{1}_m\mathbf{1}_n^T$ only requires the storage of the shift s.

- The matrix M_b contains positive integers only, since its columns the are rounded values of positive numbers. Moreover, those integers are in the range $[0, 2^b - 2]$ so only b bits are needed to store each element.

A dense storage of A will require a total of $64mn$ bits of memory if we consider that each element of A is a 64 bit float (double precision) which is usually the case in most of linear algebra toolkits or current computer systems. The matrix M_b requires bmn. If $b < 64$, there is a significant reduce of memory needed to store the approximation matrix. The Algorithm 7 shows the quantization approximation process.

Algorithm 7 Algorithm to compute a quantization approximation of a given matrix

Require: A matrix A of size $m \times n$, a positive integer $b \geq 2$ of bits to use
Ensure: A matrix M_b, a diagonal matrix D_b and a shift s.

1: Create a matrix M_b of size $m \times n$ with zeros, where each element use b bits.
2: Create a diagonal matrix D_b.
3: **for** i from 1 to n **do**
4: Set $\mathbf{v}$ as the i-th column of the matrix A.
5: Use the Algorithm 6 with b bits over $\mathbf{v}$ and get the scaling factor d_i, the approximation vector $\tilde{\mathbf{v}}$ and the shift s_i.
6: Set the i-th column of M_b as $\tilde{\mathbf{v}}$.
7: Set $s = s_i$.
8: Set $D_{i,i} = d_i$.
9: **end for**
10: Return M_b, D_b and s.

5.1.3 Matrix approximation using quantization

The error between a scalar and its quantized approximation is bounded by

$$|e_i| \leq \frac{P}{2^b - 2}.$$

Since b is a parameter of our method, we can increase its value to reduce the error. If P is small, then the error is also reduced but this value does not depend of our method but the vector that is desired to approximate. This idea inspire the following method of matrix approximation, which is the method used in our implementation of the nonlinear solver. Suppose that using the matrix approximation method described previously to compute an approximation for A using b_1 bits we build a matrix $M_{b_1}^{(1)}$ with a shift coefficient s_1 and a diagonal matrix $D_{b_1}^{(1)}$ such that

$$A \approx \tilde{A}_{b_1}^{(1)} = \left(M_{b_1}^{(1)} - s_1 \mathbf{1}_m \mathbf{1}_n^T\right) D_1^{(1)}.$$

A remark here is that the notation of superscript denotes a counting, not a power or exponent.

If $\|A - \tilde{A}_{b_1}^{(1)}\| < \|A\|$ holds for some norm, we may expect that $\tilde{A}_{b_1}^{(1)} + \tilde{A}_{b_2}^{(2)}$ is a better approximation of A, where $\tilde{A}_{b_2}^{(2)}$ is the approximation matrix produced by the proposed quantization method over $A - \tilde{A}_{b_1}^{(1)}$ using b_2 bits. That is, the matrix $\tilde{A}_{b_2}^{(2)}$ is obtained using the Algorithm 7 to compute a quantized approximation of $A - \tilde{A}_{b_1}^{(1)}$. So, given a number L of matrix terms, which we call *layers*, we propose to approximate A as

$$A \approx \sum_{i=1}^{L} \left(M_{b_i}^{(i)} - s_i \mathbf{1}_m \mathbf{1}_n^T\right) D_{b_i}^{(i)}, \tag{5.5}$$

where the i-th components $M_{b_i}^{(i)}$, s_i and $D_{b_i}^{(i)}$ are computed using the Algorithm 7 over the accumulated error matrix

$$A - \sum_{j=1}^{i-1} \left(M_{b_j}^{(j)} - s_j \mathbf{1}_m \mathbf{1}_n^T \right) D_{b_j}^{(j)}.$$

Note that using this proposal for quantization, the bits used for each layer does not need to be the same. That is, we may have a set of L different bits $\{b_1, b_2, \ldots, b_L\}$ and to approximate each layer using a different precision.

The idea of using layers or terms of matrix approximation is based on the idea that the approximation method proposed effectively decrease the norm of the error matrix. The proof of this statement will be given in the section 5.2.

The Algorithm 8 shows a general description of the proposed quantization approximation procedure. Also, to reduce the computational cost of the approximation, an additional parameter η is used by the algorithm as an stopping criteria if the relative error obtained using less than L layers is less than the tolerance. That is, when building the j-th layer, the method test the following relative error

$$\frac{\left\| A - \sum_{i=1}^{j} \left(M_{b_i}^{(i)} - s_i \mathbf{1}_m \mathbf{1}_n^T \right) D_{b_i}^{(i)}, \right\|_F}{\|A\|_F} < \eta, \tag{5.6}$$

where $\|\cdot\|_F$ denotes the Frobenius norm. If (5.6) holds at the j-th layer or iteration of the procedure, then the algorithms stop and the current result is returned.

Algorithm 8 Algorithm to approximate a matrix using a quantization approach

Require: A matrix A of size $m \times n$, a number L of layers, a list $b = (b_1, b_2, \ldots, b_L)$ with the number of bits to use in each layer, a desired minimum tolerance η.

Ensure: A list matrices $M_{b_i}^{(i)}$, each one using b_i bits, a list with diagonal matrices $D_{b_i}^{(i)}$ and a list with shifts s_i. The length of each list is at most L and is less than L if the tolerance is meet before L layers.

1: Set M as an empty list.
2: Set D as an empty list.
3: Set s as an empty list.
4: **for** k from 1 to L **do**
5: Set a matrix $B = A - \sum_{i=1}^{k-1}(M_i - s_i \mathbf{1}_m \mathbf{1}_n^T) D_i$, where M_i is the i-th element in the list M, D_i is the i-th element in the list D and s_i is the i-th element in the list s.
6: Use the Algorithm 7 over the matrix B using b_k bits to get an approximation matrix $\tilde{M}$, a diagonal matrix $\tilde{D}$ and a shift $\tilde{s}$.
7: Add $\tilde{M}_i$ at the end of the list M.
8: Add $\tilde{D}$ at the end of the list D.
9: Add $\tilde{s}$ at the end of the list s.
10: Test (5.6). If the tolerance is reached with the current number of layers, end the for loop. Otherwise, continue to the next iteration.
11: **end for**
12: Return M, D and s.

5.1.4 Quantization for Jacobian matrix approximation

The Algorithm 8 shows how to approximate any matrix given a set of parameters, but recall that our aim is to couple the approximation in the Levenberg-Marquardt algorithm, for solving (3.3). As already described in chapter 3, the matrix involved in the required product is the Jacobian matrix. So, in this section we will show how our method approximate the Jacobian matrix. As explained, the approximation procedure for a matrix is based on a scalar quantization over each column vector of the matrix. Since the input of our Levenberg-Marquardt solver is a general function $\mathbf{F}$ and considering that building explicitly the Jacobian matrix is forbidden, we can obtain a column of the Jacobian matrix using a finite-difference like approximation. To compute the k-th column of the Jacobian matrix we use the k-th canonical vector, $\mathbf{e}_k^{(n)}$, since the Jacobian matrix is of size $m \times n$. Computing a single column for the Jacobian matrix at a time is enough to compute the quantization matrix successfully. The Algorithm 9 shows a description of a procedure to quantize the Jacobian matrix. The Algorithm 9 is similar to the Algorithm 8 but some additional performance element have been considered. In section 5.3 more detailed for an implementation will be given. The process is slightly different but follows the same idea of the explained quantization process.

Algorithm 9 Algorithm to approximate the Jacobian matrix using a quantization approach

Require: The function $\mathbf{F}$ whose Jacobian will be quantized, dimension m and n, the vector $\mathbf{x}_i$ to evaluate the Jacobian matrix, a number L of layers, a list $\mathbf{b} = (b_1, b_2, \ldots, b_L)$ with the number of bits to use in each layer, a desired minimum tolerance η.

Ensure: A list M of matrices $M_{b_i}^{(i)}$, each one using b_i bits, a list D with diagonal matrices $D_{b_i}^{(i)}$ and a list S with shifts s_i.

1: Set M as a list with L matrices of size $m \times n$.
2: Set D as a list with L diagonal matrices of size $n \times n$.
3: Set S as an empty list.
4: **for** k from 1 to n **do**
5: Build the canonical vector $\mathbf{e}_k^{(n)}$.
6: Compute $\mathbf{p} := J(\mathbf{x}_i)\mathbf{e}_k^{(n)}$ using a finite difference approximation (2.2).
7: **for** j from 1 to L **do**
8: Using the Algorithm 6 compute a value d, a vector $\mathbf{y}$ and the shift s using b_j bits.
9: Set $S_k := s$.
10: Set $D_j[k,k] := d$, where $D_j[k,k]$ represent the k-th diagonal element of the j-th matrix in the list D.
11: Set the k-th column of M_j to $\mathbf{y}$.
12: Compute $\mathbf{p} := \mathbf{p} - d(\mathbf{y} - s\mathbf{1})$.
13: **end for**
14: **end for**
15: Drop unnecessary layers if the tolerance η_q is reached with less than L layers.
16: Return M, D and s.

The main modification of Algorithm 9 with respect to Algorithm 8 is that in the former we obtain a column of the matrix. Then, we compute the L layers for that column. In the latter algorithm, at each iteration a layer is computed entirely. This modification is driven by the need to avoid unnecessary evaluation of the function $\mathbf{F}$. This is important if $\mathbf{F}$ has a high computational cost per evaluation. The rest of both algorithm follows the same idea.

Note that since we are computing the matrix column-by-column, the Frobenius norm of the error matrix can be updated within every iteration, avoiding its computation at the final stage of the algorithm. Later, in section 5.3, a more detailed description of a possible implementation will be given. It is important to consider that our implementation is based on Algorithm 9.

5.1.5 Compute the matrix-vector product of the transpose of the Jacobian matrix

From (5.5) it is clear that an approximation for the transpose of a matrix A is given by

$$A^T \approx \left(\sum_{i=1}^{L} \left(M_{b_i}^{(i)} - s_i \mathbf{1}_m \mathbf{1}_n^T \right) D_{b_i} \right)^T = \sum_{i=1}^{L} D_{b_i} \left(\left(M_{b_i}^{(i)} \right)^T - s_i \mathbf{1}_n \mathbf{1}_m^T \right). \tag{5.7}$$

So in order to approximate the product $A^T\mathbf{w}$, which is our main objective for this proposal, we need to compute

$$A^T\mathbf{w} \approx \sum_{i=1}^{L} D_{b_i} \left(\left(M_{b_i}^{(i)} \right)^T \mathbf{w} - s_i \langle \mathbf{1}_m, \mathbf{w} \rangle \mathbf{1}_n \right). \tag{5.8}$$

5.2 Proof of norm decreasing of quantization approximation

The idea of using layers to approximate a matrix using quantization is based on the idea of that each layer decrease the error between the source matrix and its quantized approximation. The following theorem shows that the error, using the squared 2-norm, decrease with each quantization over a vector. Since the Frobenius norm is defined as the sum of the squared 2-norm of each column, it is clear that if the vector norm is reduced, then the Frobenius norm also decrease.

Theorem 5.2.1. *Let $\mathbf{p} \in \mathbb{R}^m$ be a vector and $\tilde{\mathbf{p}}$ be the approximation produced by the Algorithm 6. Then $\|\mathbf{p} - \tilde{\mathbf{p}}\|_2^2 < \|\mathbf{p}\|_2^2$ for all $b \geq 2$.*

Proof. If $p_i = 0$ for all i, then

$$p_i = 0 = d(y_i - s) \Leftrightarrow y_i = s.$$

Algorithm 6 produce a vector $\tilde{\mathbf{p}}$ whose coefficient are integer values from 0 to $2^b - 2$. Since $y_i = s = 2^{b-1} - 1$ is an integer and $2^{b-1} - 1 < 2^b - 2 = 2(2^{b-1} - 1)$, it is representable using b bits in our method. Then $\tilde{y}_i = y_i$ for all i and $\mathbf{p} = \tilde{\mathbf{p}} = \mathbf{0}$.

Suppose that $\mathbf{p} \neq \mathbf{0}$. Since $P = \max_{1 \leq i \leq m} |p_i|$ then there exist and index k such that $p_k = -P$ or $p_k = P$. If $p_k = -P$, then

$$p_k = -P = d(y_i - s) = \frac{P}{s}(y_k - s) \Leftrightarrow y_k = 0.$$

Since $y_k = 0$ is an integer then $\tilde{y}_k = 0$. Using $b \geq 2$ bits allows to represent at least from 0 to $2^2 - 2 = 2$. Then 0 is a valid value in the representation by our algorithm so $p_k - \tilde{p}_k = 0$. If $p_k = P$ then

$$p_k = P = \frac{P}{s}(y_k - s) \Leftrightarrow y_k = 2s = 2^b - 2.$$

Since $y_k = 2^b - 2$ is an integer, then $\tilde{y}_k = y_k = 2^b - 2$. Using $b \geq 2$ bits allows to represent values from 0 to $2^b - 2$. Since $2^b - 2$ is in the available range, then $p_k - \tilde{p}_k = 0$. This shows that $\tilde{\mathbf{p}}_k$ has at least one element from $\mathbf{p}$ so the difference $\mathbf{p} - \tilde{\mathbf{p}}$ contains at least one zero element. We need to prove that $(p_i - \tilde{p}_i)^2 \leq p_i^2$ for all i such that $|p_i| \neq P$.

Note that $(p_i - \tilde{p}_i)^2 = p_i^2 - 2p_i\tilde{p}_i + \tilde{p}_i^2 = p_i^2 - (2p_i\tilde{p}_i - \tilde{p}_i^2)$. We need to prove that $2p_i\tilde{p}_i - \tilde{p}_i^2 \geq 0$.

Let $p_i = d(y_i - s)$ and $\tilde{p}_i = d(\tilde{y}_i - s)$. Then

$$\begin{aligned} 2p_i\tilde{p}_i - \tilde{p}_i^2 &= 2d^2(y_i - s)(\tilde{y}_i - s) - d^2(\tilde{y}_i - s)^2, \\ &= 2d^2(y_i\tilde{y}_i - s(y_i + \tilde{y}_i) + s^2) - d^2(\tilde{y}_i^2 - 2s\tilde{y}_i + s^2), \\ &= d^2(2y_i\tilde{y}_i - 2sy_i - 2s\tilde{y}_i + 2s^2 - \tilde{y}_i^2 + 2s\tilde{y}_i - s^2), \\ &= d^2(2y_i\tilde{y}_i - 2sy_i + s^2 - \tilde{y}_i^2). \end{aligned}$$

Since $d^2 > 0$, we require that $c_i := 2y_i\tilde{y}_i - 2sy_i + s^2 - \tilde{y}_i^2 \geq 0$. Note that the quantized value is $y_i \geq 0$. Let $y_i = z_i + \delta_i$, where $z_i \in \mathbb{N}_0$ and $0 \leq \delta_i < 1$. Then we have that

$$\tilde{y}_i = \begin{cases} z_i, & \delta_i \leq 0.5, \\ z_i + 1, & \delta_i > 0.5. \end{cases}$$

We define $\mathcal{L} := \{i \in \{1, 2, \ldots, m\} \,|\, \delta_i \leq 0.5\}$ and $\mathcal{U} := \{i \in \{1, 2, \ldots, m\} \,|\, \delta_i > 0.5\}$. We will study both cases.

- Let suppose that $i \in \mathcal{L}$. Then $y_i = z_i + \delta_i$ and $\tilde{y}_i = z_i$. Using this in c_i we have

$$\begin{aligned} c_i &= 2(z_i + \delta_i)z_i - 2sy_i + s^2 - z_i^2, \\ &= 2z_i^2 + 2z_i\delta_i - 2sy_i + s^2 - z_i^2, \\ &= z_i^2 + 2z_i\delta_i - 2sy_i + s^2, \\ &= z_i^2 + 2z_i\delta_i - 2sy_i + s^2 + \delta_i^2 - \delta_i^2, \\ &= \left(z_i^2 + 2z_i\delta_i + \delta_i^2\right) - 2syi + s^2 - \delta_i^2, \\ &= (z_i + \delta_i)^2 - 2syi + s^2 - \delta_i^2, \\ &= y_i^2 - 2syi + s^2 - \delta_i^2, \\ &= (y_i - s)^2 - \delta_i^2. \end{aligned}$$

  In order to have $c_i \geq 0$ we require that $(y_i - s)^2 \geq \delta_i^2$. We need to study the following cases

  - Let suppose that $y_i \in \mathbb{N}_0$. Then $\delta_i = 0$ and $y_i = z_i$. Note that $y_i \in \{0, 1, \ldots, 2^b - 2\} = \{0, 1, \ldots, 2s\}$. Then $(y_i - s)^2 \geq \delta_i^2 = 0$ with equality only if $y_i = s$ which is a possible value for y_i.
  - Let suppose that $y_i \notin \mathbb{N}_0$ so $\delta_i \neq 0$. Since $i \in \mathcal{L}$ then $0 < \delta_i \leq 0.5$. We will study the possible cases for different values of z_i.

* If $z = s$, then $(y_i - s)^2 = (z_i + \delta_i - s)^2 = (s + \delta_i - s)^2 = \delta_i^2 \leq \delta_i^2$ which is true.
* If $z_i > s$, then

$$\begin{aligned} z_i - s &> 0, \\ z_i - s + \delta_i &\delta_i, \\ (z_i + \delta_i - s)^2 &> \delta_i^2. \end{aligned}$$

Then the inequality holds.

* If $z_i < s$, then $z_i \leq s - 1$ since both z_i and s are positive integer. Then $y_i - s \leq \delta_i - 1$. Since $0 < \delta_i \leq 0.5$ then $\delta_i \leq 1 - \delta_i \leq s - y_i$. Then

$$\begin{aligned} (s - y_i)^2 &\leq \delta_i^2, \\ (y_i - s)^2 &\leq \delta_i^2. \end{aligned}$$

Then the inequality holds.

Then $c_i \geq 0$ for all $i \in \mathcal{L}$ and $(p_i - \tilde{p}_i)^2 \leq p_i^2$ for $i \in \mathcal{L}$.

- Let suppose that $i \in \mathcal{U}$. Then $y_i = z_i + \delta_i$ and $\tilde{y}_i = z_i + 1$. Replacing this in c_i we have

$$\begin{aligned} c_i &= 2(z_i + \delta_i)(z_i + 1) - 2sy_i + s^2 - (z_i + 1)^2, \\ &= 2(z_i^2 + z_i + \delta_i z_i + \delta_i) - 2sy_i + s^2 - (z_i^2 + 2z_i + 1), \\ &= 2z_i^2 + 2z_i + 2\delta_i z_i + 2\delta_i - 2sy_i + s^2 - z_i^2 - 2z_i - 1, \\ &= z_i^2 + 2\delta_i z_i + 2\delta_i - 2sy_i + s^2 - 1 + \delta_i^2 - \delta_i^2, \\ &= (z_i^2 + 2z_i\delta_i + \delta_i^2) - 2sy_i + s^2 + 2\delta_i - 1 - \delta_i^2, \\ &= (z_i + \delta_i)^2 - 2sy_i + s^2 + 2\delta_i - 1 - \delta_i^2, \\ &= (y_i^2 - 2sy_i + s^2) + 2\delta_i - 1 - \delta_i^2, \\ &= (y_i - s)^2 - (1 - 2\delta_i + \delta_i^2), \\ &= (y_i - s)^2 - (1 - \delta_i)^2, \\ &= (y_i - s + 1 - \delta_i)(y_i - s - 1 + \delta_i), \\ &= (z_i + \delta_i - s + 1 - \delta_i)(z_i + \delta_i - s - 1 + \delta_i), \\ &= (z_i - s + 1)(z_i - s + 2\delta_i - 1). \end{aligned}$$

Since $i \in \mathcal{U}$, then $0.5 < \delta_i < 1$. Also note that $z_i \in \mathbb{N}_0$. Then we need to study the following cases:

– If $z_i = s$, then

$$(z_i - s + 1)(z_i - s + 2\delta_i - 1) = 2\delta_i - 1.$$

Note that

$$\frac{1}{2} < \delta_i < 1 \Leftrightarrow 0 < 2\delta_i - 1 < 1.$$

Then $c_i > 0$ and the inequality holds.

- If $z_i > s$, then $z_i - s > 0$. Then clearly $c_i > 0$ and the inequality holds.
- If $z_i < s$ or $z_i \leq s - 1$, then $z_i - s + 1 \leq 0$. Also

$$z_i - s + 2\delta_i - 1 \leq -1 + 2\delta_i - 1,$$
$$z_i - s + 2\delta_i - 1 = 2(\delta_i - 1).$$

  Since $\delta_i < 1$ then $z_i - s + 2\delta_i - 1 < 0$. Then $z_i - s + 1 \leq 0$ and $z_i - s + 2\delta_i - 1 < 0$ so $c_i \geq 0$ and the inequality holds.

Then $c_i \geq 0$ for all $i \in \mathcal{U}$.

Since $c_i \geq 0$ for both $\mathcal{L}$ and $\mathcal{U}$, then $c_i \geq 0$ for all $i \in \{1, 2, \ldots, m\}$ and the theorem is proved. □

The theorem states that the difference between the vector and its approximation strictly decrease its norm with respect to the original vector. Since there is always a value P in the approximation vector, each layer of the quantization has at least one equal value so each layer produce a error vector with at least one zero element.

5.3 Implementation details of the quantization approximation

All the algorithms described in section 5.1.1 where presented with an easy description. It is important to remark that their description do not reflect an optimal or a practical implementation. The main idea of these algorithms is to describe the method and their keys components, not a particular implementation of it. Regardless of this, their actual implementation has a major role in the overall solver since a poor implementation will impact in the memory usage and computation time of the proposal. In particular, our method it is based on the idea of storing matrices whose coefficients may use less than 8 bits, which is usually the minimum unity in most modern computers. In this chapter, a proposal for an implementation of a quantized matrix is given. The focus on this proposal is the possibility of storing matrices whose coefficients may use less than 8 bits and also to be suitable for a parallel implementation. The former is a major component, required to achieve a significant reduce of memory usage while the latter is useful when parallel or distributed resources are available.

5.3.1 Storage of integer matrices

The minimal unity of of a memory chunk in modern computers is 1 byte or equivalently, 8 bits. This means that all structures that the computer read, writes and interpret are composed at least by 8 bits or multiples of it. So a quantized matrix M using 2 bits per element will use 8 bits per element if it is stored directly without a proper data structure, specifically constructed to manage this issue, losing $\frac{3}{4}$ of

memory. To implement our proposal, it is necessary to build a specific data structure for this purpose. In this section we will explain briefly how our implementation manage this issue.

Note that matrices M produced by Algorithm 7 using b bits contains positive integers only, ranging from 0 to $2^b - 2$. Since all the coefficients are positive integers, there exist a binary representation of each one so we have that a matrix $M_{b_i}^{(i)}$ may be written as

$$M_{b_i}^{(i)} = \sum_{k=0}^{b_i-1} 2^k \, M_{b_i,k}^{(i)}, \tag{5.9}$$

where $M_{b_i,k}^{(i)}$ denotes the k-th binary coefficient of the binary representation of all coefficients from $M_{b_i}^{(i)}$. The number of bits b_i defines the length of the sum (5.9). Our proposal to achieve a true compression of the approximation matrix is to represent the matrix M_{b_i} in (5.4) as a sum of binary matrices, as shown in (5.9). The second subindex k denotes the position in the binary representation of coefficients. As said previously, the minimal unity of storage is 8 bits, so even that $M_{b_i,i}^{(k)}$ is a binary matrix, each of its elements need 8 bits for storage. To achieve the full compression of our method, we propose the following approach to store each matrix $M_{b_i,k}^{(i)}$.

Note that by Euclidean division, we have that there exist an integer p and r such that

$$n = 8p + r.$$

Our proposal is to store the matrix $M_{b_i,k}^{(i)}$, whose dimension is $m \times n$, as a matrix $\tilde{M}_{b_i,k}^{(i)}$ of dimension $m \times p$ if $r = 0$ or dimension $m \times (p+1)$ if $r > 0$. For simplicity, we will consider that $r = 0$ for all of our examples and explanations, since for $r > 0$ we can fill the matrix with zeros until $r = 0$. The (i, j) element of $\tilde{M}_{b_i,k}^{(i)}$ contains a positive integer whose binary representation is given by the number from position $(i, 8j)$ to $(i, 8j + 7)$ of $M_{b_i,k}^{(i)}$, similar of what is done for the function `packbits` from *Numpy* library[1]. Using this packing and unpacking approach, the matrix is compressed and decompressed in run time when it is needed. Equation (5.10) shows an example of this compression approach. The left matrix contains, for example, seven rows of the matrix $M_{b_i,k}^{(i)}$ and eight columns. This matrix is compressed in the right vector, which contains integers whose representations are given for columns in the left side matrix.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0 \\ 2 \\ 9 \\ 240 \\ 52 \\ 241 \\ 1 \end{bmatrix}. \tag{5.10}$$

[1] `https://numpy.org/doc/stable/reference/generated/numpy.packbits.html?highlight=packbits#numpy.packbits`

For example, taking the fifth row of the left matrix in (5.10), we have that

$$0 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 32 + 16 + 4 = 52,$$

where 52 is the fifth coefficient in the right-hand side. The previous procedure is equivalent to multiply a $m \times 8$ matrix by the following vector,

$$\mathbf{c}_{\text{bin}} = \begin{bmatrix} 2^7 \\ 2^6 \\ 2^5 \\ 2^4 \\ 2^3 \\ 2^2 \\ 2^1 \\ 2^0 \end{bmatrix} = \begin{bmatrix} 128 \\ 64 \\ 32 \\ 16 \\ 8 \\ 4 \\ 2 \\ 1 \end{bmatrix}. \tag{5.11}$$

To compute the required memory that our data structure needs, recall that from a matrix $M_{b_i}^{(i)}$ of size $m \times n$ with positive integer coefficients, we produce b_i binary matrices, as in (5.9). Each of these binary matrices will be compressed to a matrix of size $m \times p$, where $n = 8p$, whose coefficients are positive 8 bits integers only. Since each of these matrices are 8 bits integers, then the memory required to store one of these matrices is given by $8mp = 8m\frac{n}{8} = mn$ bits. Since we have b_i of these matrices, the total amount of memory required to store the matrix $M_{b_i}^{(i)}$ using these representation and procedure is $b_i mn$, which is equivalent to the memory requirements that a single matrix of b_i bits per coefficients. So, our proposed method of compression and data structure achieves the desired memory usage.

5.3.2 Implementation details for compression, decompression and usage of a quantized matrix

Note that the described scheme is for storage only. This is a data structure used to be able to store matrices using less than 8 bits per coefficients, but in the actual computation or usage of the matrix, it will be decompressed to retrieve its binary representation. Also, note that this procedure transform matrices of size $m \times 8$ into a 8 bit integer vector of size m. The choice of packing from 8 columns of the binary matrix is arbitrary and a bigger number of columns may be used. However, our experiments shows that using combination of less than 8 bits achieves small errors in the approximation matrices. This results will be shown in the chapter 7.

The Algorithm 10 describe the compression procedure of a $m \times 8$ matrix, in a simple manner. This implementation is simple to understand but requires an intermediate matrix to store partial results. The Algorithm 11 shows a better and more memory-efficient algorithm. By observing Algorithm 10 and 11, it is clear that our proposal for quantization *compress* each row of the $m \times 8$ matrix. We call this a *row-wise* compression. A *column-wise* compression it is also possible. In either case,

our proposed algorithm is suitable for parallelization since each of the rows can be compressed in parallel.

Algorithm 10 Algorithm to compress (*pack*) a portion of a matrix

Require: An integer matrix A of size $m \times 8$ to compress using b bits. It is required that each integer in A fits using b bits.

Ensure: Returns a $m \times b$ matrix with 8-bit positive integers.

Let M be a matrix a zero-matrix of size $m \times b$.
Let B be an intermediate matrix of size $m \times 8$, used to store intermediate results for packing.
Let $\mathbf{c}_{\text{bin}}$ be the vector defined in (5.11).
for k from 1 to b **do**
  for i from 1 to m **do**
    for j from 1 to 8 **do**
      Let $A_{i,j} = \sum_{d=0}^{b-1} c_d 2^d$ be the binary representation of $A_{i,j}$. Set $B_{i,j} := c_{k-1}$.
    end for
  end for
  Set the i-th column of M equal to $B\mathbf{c}_{\text{bin}}$.
end for
Return M.

Algorithm 11 Algorithm to compress (*pack*) a portion of a matrix without an inner matrix

Require: An integer matrix A of size $m \times 8$ to compress using b bits. It is required that each integer in A fits using b bits.

Ensure: Returns a $m \times b$ matrix with 8-bit positive integers.

Let M be a matrix a zero-matrix of size $m \times b$.
Let $\mathbf{c}_{\text{bin}}$ be the vector defined in (5.11).
for i from 1 to m **do**
  for j from 1 to 8 **do**
    Let $A_{i,j} = \sum_{d=0}^{b-1} c_d 2^d$ be the binary representation of $A_{i,j}$.
    for k from 1 to b **do**
      $M_{i,b} := M_{i,b} + 2^{8-j} c_{k-1}$.
    end for
  end for
end for
Return M.

Algorithm 12 combines the Algorithm 9 with the Algorithm 11 to compress a Jacobian matrix using our quantization approximation and the proposed data structure and compression approach. Note that the algorithm stores the compressed matrix, since this structure is what makes possible the reduction in memory usage, with respect to the original matrix.

Algorithm 12 Algorithm to compress a Jacobian matrix using a quantization approach

Require: The function $\mathbf{F}$ whose Jacobian will be quantized, dimension m and n, the vector $\mathbf{x}_i$ to evaluate the Jacobian matrix, a number L of layers, a list $\mathbf{b} = (b_1, b_2, \ldots, b_L)$ with the number of bits to use in each layer, a desired minimum tolerance η.

Ensure: A list $M = \{M_2, M_2, \ldots, M_L\}$, where M_i is a list containing p matrices of size $m \times b$, a list D with diagonal matrices $D_{b_i}^{(i)}$ and a list S with shifts s_i.

Create L empty lists M_i, for $i \in \{1, 2, \ldots, L\}$.
Set D as a list with L diagonal matrices of size $n \times n$.
Set S as an empty list.
Set $i := 1$.
Set A_i as a zero-matrix of size $m \times 8$, for $i \in \{1, 2, \ldots, L\}$.
for k from 1 to n **do**
    Build the canonical vector $\mathbf{e}_k^{(n)}$.
    Compute $\mathbf{p} := J(\mathbf{x}_i)\mathbf{e}_k^{(n)}$ using a finite difference approximation (2.2).
    for j from 1 to L **do**
        Using the Algorithm 6 compute a value d, a vector $\mathbf{y}$ and the shift s using b_j bits.
        Set $S_k := s$.
        Set $D_j[k, k] := d$, where $D_j[k, k]$ represent the k-th diagonal element of the j-th matrix in the list D.
        Set the i-th column of A_j to $\mathbf{y}$.
        Compute $\mathbf{p} := \mathbf{p} - d(\mathbf{y} - s\mathbf{1})$.
    end for
    $i := i + 1$.
    if $i > 8$ **then**
        $i := 1$.
        for j from 1 to L **do**
            Compress the matrix A_j using Algorithm 11 store its result in $\tilde{A}_j$.
            Add $\tilde{A}_j$ to M_j.
        end for
    end if
end for
Let M be a list containing all M_i, for $i \in \{1, 2, \ldots, L\}$.
Drop unnecessary layers if the tolerance η_q is reached with less than L layers.
Return M, D and S.

Last but of no lesser importance, the Algorithm 14 shows the decompression procedure, used to compute the matrix transpose-vector product between the quantized matrix and an arbitrary vector. This is the only application that requires a decompression of the data. It use the Algorithm 13 to decompress and perform small products, for each of the $m \times b$ matrices. Since the required matrix is compressed,

the Algorithm 14 performs both operations at the same time: the decompression and the transpose-vector product. Since we are multiplying an approximation of the transpose of the Jacobian matrix, the input vector dimensions should be m. For a parallel implementation of Algorithm 14, the for-loop from 1 to p can be executed in parallel, since each *pack* can be decompressed in parallel. The Algorithm 13 is difficult to understand at first glance, but its main idea is that the algorithm traverses the matrix of size $m \times b$, obtain its binary representation and use its binaries coefficients to perform the product with the corresponding coefficient in the input vector.

Algorithm 13 Algorithm to decompress and multiply a packed matrix

Require: A compressed matrix M of size $m \times b$, a vector $\mathbf{v}$ of size m.

Ensure: Returns a vector $\mathbf{y}$ of size 8 with the result of the product for the transpose of the uncompressed matrix $m \times 8$ and $\mathbf{v}$.

1: $\mathbf{a} := \mathbf{0}_8$.
2: **for** i from 1 to m **do**
3: **for** j from 1 to b **do**
4: Let $A_{i,j} = \sum_{r=0}^{7} c_r 2^{7-r}$ be its binary representation.
5: **for** k from 1 to 8 **do**
6: $a_k := a_k + c_{k-1}\, v_i$, where a_i and v_i denotes the i-th component of $\mathbf{a}$ and $\mathbf{v}$, respectively.
7: **end for**
8: **end for**
9: **end for**
10: Return $\mathbf{a}$.

Algorithm 14 Algorithm to perform the transpose of a quantized matrix and a vector

Require: The list M, D and S returned from the Algorithm 12, a vector $\mathbf{v}$ of size m.

Ensure: Returns a vector $\mathbf{y}$ of size n.

1: Let $\mathbf{y} := \mathbf{0}_n$.
2: $p := \frac{n}{8}$.
3: **for** k from 1 to L **do**
4: $\mathbf{a} := \mathbf{0}_n$.
5: **for** t from 1 to p **do**
6: Set A to be the t-th matrix in the M_k list. A is of size $m \times b$.
7: Let $\mathbf{b}$ be the output vector of Algorithm 13 with A and $\mathbf{v}$.
8: Set from the $8(t-1)+1$ to the $8t$ coefficient of $\mathbf{a}$ equal to $\mathbf{b}$.
9: **end for**
10: $\mathbf{y} := \mathbf{y} + \mathbf{a}$.
11: **end for**
12: Return $\mathbf{y}$.

5.4 Total memory usage and computational complexity

In this final section of the Quantization chapter, we will show the total memory usage of the proposed quantized approximation, along with its computational complexity. Let suppose that we quantize a Jacobian matrix using L layers and the bits for each layer is given by the set $\{b_1, b_2, \ldots, b_L\}$. Also, let f be the computational complexity of a single evaluation of $\mathbf{F}$.

5.4.1 Memory usage

To compute to total memory usage of our proposed quantized approximation, we need to consider the following components per layer:

- The storage of the scaling diagonal matrix D_i, for $i \in \{1, 2, \ldots, L\}$. It can be efficiently stored using an n-size vector.
- A single shift s_i is required for a specific layer.
- Recall from (4.12) that the $m \times n$ matrix M_i is compressed as b_i matrices of size $m \times p$ of 8 bit integers, where $n = 8p$ for p a positive integer. Therefore, the compressed used memory for a single matrix M_i is given by $b_i(8mp) = b_i mn$.

Therefore, the memory used by a single layer is given by $b_i mn + n + 1$. For the L layers, the total memory usage is given by

$$\sum_{i=1}^{L} (b_i mn + n + 1) = mn \sum_{i=1}^{L} b_i + nL + L = \mathcal{O}\left(mn \sum_{i=1}^{L} b_i \right). \tag{5.12}$$

Expression (5.12) was obtained by summing the storage requirement of D_i, the shift value and the $m \times n$ integer matrix, compressed using b_i bits, for each layer. The expression (5.12) shows that the required memory of our quantized approximation is slightly superior to the theoretical expected memory $b_i mn$ per layer, but contains the overhead produced by the storage of the shift and the diagonal matrix D_i, which is part of our representation for compression. Since usually L will not be too large (in the experiments of chapter 7, $L = 4$ at most), the term nL and L are small compared to the dominant $mn \sum_{i=1}^{L} b_i$.

5.4.2 Computational complexity

Here, we remark two computational complexities that are relevant in this approximation. These are:

- The complexity of building our quantized approximation for the Jacobian matrix $J(\mathbf{x}_i)$. This consider the compression procedure.

- The complexity of performing the product of the transpose of the quantized matrix and a vector. This operation consider the decompression procedure before performing the actual product.

For the building of the quantized approximation and its later compression, we look at the Algorithm 12. The compression procedures has the following computational operations that needs to be considered:

- At each outer iteration a column of the Jacobian is computed through the product $J(x_i)\mathbf{e}_k^{(n)}$. This is computed using a finite-difference approximation. In our implementation, a second-order approximation was used, which is given by the following expression

$$J(x_i)\mathbf{e}_k^{(n)} \approx \frac{\mathbf{F}(\mathbf{x}_i + \varepsilon\mathbf{e}_k^{(n)}) - \mathbf{F}(\mathbf{x}_i - \varepsilon\mathbf{e}_k^{(n)})}{2\varepsilon}.$$

  This approximation consist in the following operations:

  - The computation of $\mathbf{x}_i + \varepsilon\mathbf{e}_k^{(n)}$ and $\mathbf{x}_i - \varepsilon\mathbf{e}_k^{(n)}$ requires $n+1$ elemental operations for each vector operation. Since the one of the vector is the k-th canonical vector, the actual sum or subtraction is efficiently performed summing or subtracting the value of ε directly in the k-th position of $\mathbf{x}_i$. This gives a total of $2n+2$ elemental operations.
  - Two function evaluations of $\mathbf{F}$. This has a computational cost of $2f$ elemental operations.
  - The subtraction of $\mathbf{F}(\mathbf{x}_i + \varepsilon\mathbf{e}_k^{(n)}) - \mathbf{F}(\mathbf{x}_i - \varepsilon\mathbf{e}_k^{(n)})$, which is a subtraction of an m-dimensional vector. This requires m elemental operations.
  - Finally, the division by 2ε, which requires $m+1$ elemental operations.

  Hence, the total count of elemental operations used to retrieve a single column of the Jacobian matrix is given by $2n+2+2f+m+m+1 = 2(m+n+f)+3$. If this procedure is repeated for all columns of the Jacobian matrix, the final number of elemental operations is given by $2n(m+n+f)+3n$.

- Line 10 of Algorithm 12 requires the usage of the Algorithm 6 to the quantized column, the shift and its scaling factor. Without giving too much detail of Algorithm 6, its more expensive operations are:

  - Found the maximum value in the input vector requires m elemental operations.
  - Compute the vector $\mathbf{y}$ requires $2m+1$ elemental operations.
  - Rounding require proportionally m elemental operations.

  Therefore, the elemental operations required by the Line 10 of Algorithm 6 is approximately $4m+1$. Also the Line 14 requires $3m$ elemental operations. All

these lines are repeated L times per outer iteration. Therefore, the computational cost of the loop in Line 9 of Algorithm 12 is $L(4m+1+3m) = 7Lm+L$. Since this is repeated n times, once per column, the total number of elemental operations is $n(7Lm + L)$

- From Line 17 to 23 of Algorithm 12, the compression procedure is realized. Note that this procedure is executed at each "pack". Under the assumption that $n = 8p$, for p a positive integer, then the procedure is executed p times. At each time, the Algorithm 11 is executed L times. The complexity of the Algorithm 11 is given by the procedure of compressing an $m \times 8$ matrix to produce a $m \times b$ matrix. The general complexity of this is given by $8b_j m$, for $j \in \{1, 2, \ldots, L\}$. This procedure is repeated pL times, giving a final complexity of

$$p \sum_{j=1}^{L} 8b_j m = 8pm \sum_{j=1}^{L} b_j = mn \sum_{j=1}^{L} b_j.$$

Summarizing, the overall complexity, considering all previous section of the compression algorithm, is given by

$$mn \left(2 + 7L + \sum_{j=1}^{L} b_j \right) + 2n^2 + 2nf + n\left(3 + L\right),$$

or

$$\mathcal{O}\left(mn \left(2 + 7L + \sum_{j=1}^{L} b_j \right) + 2nf \right). \tag{5.13}$$

Equation (5.13) shows that the complexity of the compression procedure depends of the cost of the evaluation of $\mathbf{F}$. Also depends on the number of layers and bits used. This shows that the computational cost is higher than building a matrix but the increased computational cost of building the matrix saves memory usage of the matrix, which is a very important feature when the matrix does not fit in the available memory. An important remark here is that we do not consider the computational cost required to obtain the binary representation of a number. The reason for this is that in our implementation, a table is pre-computed whose entries contains the 8-bit representation from 0 to 255. The representation is of 8 bits since we are compressing in packs of 8 columns. If a bigger pack is desired, the pre-computed table also need to reflect this.

The next component that requires our attention is the computational complexity of performing a matrix-vector product. In particular, we focus on the product of our interest, which is the transpose of the quantized matrix and a vector. To get a complexity for this procedure, we need to look first at the Algorithm 13. It receives compressed matrix of size $m \times b$ and a vector of size m, and returns a vector of size 8 with the result of the transpose of the compressed portion of the matrix and

the input vector. Neglecting a product and the obtain of the binary representation, which can be obtained efficiently using a pre-computed table, the procedure requires approximately $8b_jm$ operations, for $j \in \{1, 2, \ldots, L\}$. Considering this and observing the Algorithm 14, we have that the major computational task has the following complexity:

- From Line 6 to 8 in Algorithm 14, the mayor computation is done by the Algorithm 13. This procedure requires $8b_jm$ operations. This procedure is repeated p times.
- Line 10 requires n elemental operations.

Previous task are repeated L times. Therefore, the computation complexity of performing a single transpose matrix-vector product is given by

$$nL + 8pm \sum_{j=1}^{L} b_j = nL + mn \sum_{j=1}^{L} b_j \tag{5.14}$$

Equation (5.14) shows that the computational complexity of performing a single product is $\mathcal{O}\left(mn \sum_{j=1}^{L} b_j\right)$. This shows that using our quantized approach, computing matrix-vector product is more computational demanding than realizing the same product using an explicitly stored matrix. This is the cost that is payed to be able to store a large-scale approximation matrix. From (5.14) also is clear that having more layers and using more bits to approximate the matrix increase the computational cost of performing a single product. The reason for this is the required task of decompression of the data, to be used as matrix to perform the desired product.

Chapter 6

Proposed nonlinear solver

At this moment, several elements have been explained. All these components play an important role in the principal focus of this thesis: the solution of large-scale nonlinear problems. As mentioned in section 3, the algorithm that will be used is the Levenberg-Marquardt method. Although in section 3 a brief description was given, in this section we will give further details and also define some important elements that need to be specified before making a practical implementation.

6.1 The Levenberg-Marquardt method

As briefly explained in chapter 3, the Levenberg-Marquardt method is an algorithm proposed by Marquardt Marquardt 1963 to solve nonlinear least-square problems. In our case, giving a function $\mathbf{F} : \mathbb{R}^n \to \mathbb{R}^m$ with $m \geq n$, we try to solve $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ by means of solving the following unconstrained optimization problem

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

where $g(\mathbf{x})$ is defined as

$$g(\mathbf{x}) = \frac{1}{2}\|\mathbf{F}(\mathbf{x})\|^2,$$

for convenience. In chapter 3 a general algorithm was given, the Algorithm 1. However, such explanation was very general and some important details were not specified. These details are

- The construction of the Jacobian matrix and its transpose.
- The solution of the linear system of equations (3.5) or its equivalent least-square problem given by (3.6).
- Criteria for acceptance of the current computed solution $\Delta\mathbf{x}_i$.
- Criteria for updating the damping factor λ_i.

At this point, it is clear what is the proposal of this thesis regarding the first two components: the construction of the Jacobian matrix never happens explicitly, but

- The algorithm is designed to never require the explicit Jacobian matrix, but only its action over a vector, i.e. $J(\mathbf{x}_i)\mathbf{v}$, for $\mathbf{v}$ arbitrary. This required product is computed in our method using a finite-difference approximation.

- The algorithm also does not require the transpose of the Jacobian matrix but only its action over a vector. Also, the matrix-vector product requirement can be an approximation of any precision, i.e. does not need to be a high precision approximation but a coarse one may also work. However, better approximation improves the performance of the overall method. This product is approximated by our quantized approximation, explained in chapter 5.

- Both previous items are combined to solve the inner least-square problem by means of the nsLSQR method, explained in chapter 4. The benefits and also the need of nsLSQR was already explained, but in summary it can be used with the quantized approximation to successfully solve the linear least-square problem, even if the approximation is not accurate.

The only missing elements to have a fully functional implementation of a Levenberg-Marquardt method are the stopping criteria, the acceptance criteria for the current solution and the update criteria for the damping factor. In this section we will briefly expose the stopping criteria used in our implementation, while in the following sections the criteria for updating the solution and damping factor will be explained in more detail.

Our implementation use three simple stopping criteria:

- Given a tolerance η, the method stops if the following expression holds,

$$\frac{\|\mathbf{F}(\mathbf{x}_i)\|}{\|\mathbf{F}(\mathbf{x}_0)\|}, \tag{6.1}$$

  where $\mathbf{x}_0$ denotes the initial guess and $\mathbf{x}_i$ the i-th computed solution.

- Given a tolerance η_d and an integer d, the method stops if the following expression holds

$$\frac{\|\mathbf{x}_{i+1} - \mathbf{x}_i\|}{\|\mathbf{x}_{i+1}\|} < \eta_d, \tag{6.2}$$

  for $i \in \{i - d, i - d + 1, \ldots, i\}$.

- As a safe guard method, if T iterations are executed, the method stops.

6.2 Computation of damping factor

An important component that has not been discussed yet is the value and updating criteria of the damping factor. As already explained in chapter 2 and 3, the damping factor plays two mayor roles in the Levenberg-Marquardt method:

- Makes the matrix $J^T(\mathbf{x}_i)J(\mathbf{x}_i)$ non-singular by adding a positive value.
- The magnitude of the value of the damping factor can be used to speed-up the convergence of the method when close to the solution or guarantee convergence when far from the solution. In other words, its value can be used to build a globally convergent method.

The damping factor is an important element in the performance of the Levenberg-Marquardt method and still is an active research field Fan and Pan 2009; Cui, Zhao, B. Xu, and Gao 2017. Although more robust method to compute the damping factor exist, our implementation use a more simple approach. This is due to that the key of our proposal is to propose a variant of a Levenberg-Marquardt method with novel inner components, i.e. the usage of quantization and the nsLSQR. The usage of more robust alternatives for computing the damping factor is an interesting path for a future work. In particular, our implementation use a mix of methods in Kelley 1999; Marquardt 1963; S. J. Wright and Holt 1985. Basically, the approach is to increase or reduce the damping value depending on the ratio between the actual reduction of the new solution and the predicted reduction of the linear model.

As mentioned 3, the original proposal made by Marquardt in Marquardt 1963 was a simple criteria to increase or decrease λ. In general, the original proposal of Marquardt is based on the solution of the linear system using a damping factor. Using a set of criteria, the damping factor and the computed step is accepted or rejected, requiring the computation of a new damping factor to obtain a new step. Given a current approximate solution $\mathbf{x}_i$, a new step $\Delta\mathbf{x}_i$ is computed using the damping factor λ_i. Let $\mathbf{x}_i = \mathbf{x}_{i-1} + \Delta\mathbf{x}_{i-1}$ be the approximate solution computed in the previous iteration using λ_{i-1} and $\nu > 1$. To compute λ_i, Marquardt proposed:

- Compute $\mathbf{a}$ as the solution of

$$\left(J^T(\mathbf{x}_i)J(\mathbf{x}_i) + \frac{\lambda_{i-1}}{\nu}I\right)\mathbf{a} = -J^T(\mathbf{x}_i)\mathbf{F}(\mathbf{x}_i).$$

  If $g(\mathbf{x}_i + \mathbf{a}) \leq g(\mathbf{x}_i)$, then the step is accepted. This means that $\lambda_i = \frac{\lambda_{i-1}}{\nu}$ and $\Delta\mathbf{x}_i = \mathbf{a}$. Using this, the new approximated solution is $\mathbf{x}_{i+1} = \mathbf{x}_i + \Delta\mathbf{x}_i$. If the condition is not meet, then test the next condition.

- Compute $\mathbf{b}$ as the solution of

$$\left(J^T(\mathbf{x}_i)J(\mathbf{x}_i) + \lambda_{i-1}I\right)\mathbf{b} = -J^T(\mathbf{x}_i)\mathbf{F}(\mathbf{x}_i).$$

  If $g(\mathbf{x}_i + \mathbf{a}) > g(\mathbf{x}_i)$ and $g(\mathbf{x}_i + \mathbf{b}) \leq g(\mathbf{x}_i)$, then the solution is accepted. Thus $\lambda_i = \lambda_{i-1}$ and $\Delta\mathbf{x}_i = \mathbf{b}$. The new solution is computed as $\mathbf{x}_{i+1} = \mathbf{x}_i + \Delta\mathbf{x}_i$.

- If the previous test also fails, then increase λ by successive multiplication by ν until reach an integer k such that for a vector $\mathbf{c}$ as the solution of

$$\left(J^T(\mathbf{x}_i)J(\mathbf{x}_i) + \lambda_{i-1}\nu^k I\right)\mathbf{c} = -J^T(\mathbf{x}_i)\mathbf{F}(\mathbf{x}_i),$$

  that satisfies $g(\mathbf{x}_i+\mathbf{c}) \leq g(\mathbf{x}_i)$. In such case, we have $\lambda_i = \lambda_{i-1}\nu^k$ and $\Delta\mathbf{x}_i = \mathbf{c}$. The new approximated solution is $\mathbf{x}_{i+1} = \mathbf{x}_i + \Delta\mathbf{x}_i$.

As can be seen, the approach proposed by Marquardt is very simple. The main idea is that to test if we can reduce the current damping factor as $\frac{\lambda_{i-1}}{\nu}$, trying to speed up the convergence. If that fails to meet the acceptance criteria, which in the case of the proposal of Marquardt its simply some reduction in the residual, we try to achieve some progress using the current damping factor. If that also fails, then we increase the current damping factor by a constant factor until acceptance. Theoretically, this can always be achieved since we know that the the step computed by the Levenberg-Marquardt method is always a descent direction for $g(\mathbf{x}_i)$.

Although the simplicity of Marquardt's approach, it is not used in practice. Based on Fletcher 1971; Moré 1978; Kelley 1999, a more sophisticated approach will be used in our method. The modification of the damping factor is based on how *well* the linear model approximate the true function. The usage of a linear model is the base of Newton-type methods. If the linear model differs greatly of the real function, there is no reason to guarantee that solving the linear model will lead to a significant improvement. So the damping factor is increased to make the algorithm closer to a descent gradient approach. On the contrary, if the linear model is close to the function, a method closer to Gauss-Newton is preferred to improve the convergence rate. The actual reduction, **ared**, is computed as

$$\text{ared} = \|\mathbf{F}(\mathbf{x}_i)\|_2^2 - \|\mathbf{F}(\mathbf{x}_i + \Delta\mathbf{x}_i)\|_2^2.$$

This is equivalent to the difference in the residual between the current solution and the computed solution. It is called "actual" reduction since we are using $\mathbf{F}$ to compute it, hence it represents the achieved reduction between the residuals. The predicted reduction, pred, is computed as

$$\text{pred} = \langle J(\mathbf{x}_i)\,\Delta\mathbf{x}_i, \mathbf{F}(\mathbf{x}_i)\rangle - \frac{1}{2}\left(\|J(\mathbf{x}_i)\,\Delta\mathbf{x}_i\|^2 + \lambda\|\Delta\mathbf{x}_i\|^2\right).$$

It is computed as the difference in the residual between the current solution and the residual obtained by using the linear model Kelley 1999, which is used to compute the next solution in the inner least-square problem.

Given parameters $0 < \omega_d < 1 < \omega_i$, $\mu_0 \leq \mu_l < \mu_h$ and considering that $\gamma =$ ared/pred, the step $\Delta\mathbf{x}_i$ computed using λ_i and the current solution $\mathbf{x}_i$, the used method for updating the damping factor is given by

- If $\gamma < \mu_0$, then $\lambda_{i+1} = \omega_i\lambda_i$ and $\mathbf{x}_{i+1} = \mathbf{x}_i$.
- If $\mu_0 \leq \gamma < \mu_l$, then $\lambda_{i+1} = \omega_i\lambda_i$ and $\mathbf{x}_{i+1} = \mathbf{x}_i + \Delta\mathbf{x}_i$.

- If $\mu_l \leq \gamma \leq \mu_h$, then $\lambda_{i+1} = \lambda_i$ and $\mathbf{x}_{i+1} = \mathbf{x}_i + \Delta\mathbf{x}_i$.
- If $\mu_h < \gamma$, then $\lambda_{i+1} = \omega_d \lambda_i$ and $\mathbf{x}_{i+1} = \mathbf{x}_i + \Delta\mathbf{x}_i$.

To avoid $\lambda_i = 0$, we require a minimum value $\lambda_{\min}$ since the linear system (3.5) could be singular for $\lambda = 0$. The general idea of the used method and its criteria of acceptance and rejection is,

- The value of ω_d is used to decrease the damping factor, while the scalar ω_i is used to increase its value. This process requires an initial damping factor.
- The values of μ_0, μ_l and μ_h defines a range of acceptance in the mismatch of pred and ared.
- If the ration γ is less than μ_0, which is the minimum value of mismatch tolerated, then the computed step is rejected and the current solution is maintained. Also, the damping factor is increased. The rejection is due to the high mismatch between the true function and its linear approximation. Since the model does not reflect accurately the nonlinear function, the damping factor is increased with the goal of making a step closer to a steepest descent method.
- If $\gamma > \mu_0$, then the step is accepted since μ_0 represents the minimum value for mismatch acceptance. However, depending on where γ is with respect to μ_l and μ_h, the damping factor is increase, maintained or decreased.

Using this approach, we do not worry about the residual behavior along with the iterations. We expect that if we are far from the solution, the linear model will be not perform very well and hence, a bigger damping factor will be required to ensure convergence.

6.3 lm-nsLSQR algorithm

All the components required to propose a completed algorithm were explained. In this final section, we will show the final algorithm that will describe our proposed nonlinear solver for large-scale overdetermined problems. We call the algorithm *lm-nsLSQR* (Levenberg-Marquardt nsLSQR).

For readability, the set of input parameters will be given here and not in the Algorithm 15 since the list is quite large. Also, to improve readability, will be grouped depending on the purpose of the parameter. Those are

- Parameters for the Levenberg-Marquardt method itself:
  - A function $\mathbf{F}$.
  - A vector $\mathbf{x}_0$ used as an initial guess.
  - An initial damping factor λ_0.

  - Scalar values used for increase or reduce the damping factor, i.e. values for ω_i and ω_d.
  - Scalar values for acceptance or rejection of solution and damping factor, i.e. values for μ_0, μ_l and μ_h.
  - A tolerance η to be used by the lm-nslSQR method as a stopping criteria.
  - A tolerance η_d and an integer value d, used for stopping the method when no significant progress is made.
  - A maximum number of iterations T for lm-nsLSQR.

- Quantized approximation parameters:
  - An integer L to be used as the number of layers for quantization.
  - A list $b = \{b_1, b_2, \ldots, b_L\}$ of L bits to be used per each layer.
  - A tolerance η_q, to be used in the quantized approximation as an accepted relative error in the approximation.

- nsLSQR parameters:
  - The number of iterations for t_{out} and t_{in}.
  - A tolerance η_t to be used for the residual.
  - An integer r and tolerance η_r used for to test the varying between iterations.
  - An integer s and tolerance η_s used to test the saturation of the residual.

- Finite-difference parameter: an scalar ε.

An important remark about the Algorithm 15 is that the quantized matrix does not need to be computed at every iteration. Since there is chance that the current step is not accepted, then the current solution also does not change. Therefore, the matrix $J(\mathbf{x}_i)$ remains the same and hence, there is no need to update the quantized matrix.

Algorithm 15 lm-nsLSQR Algorithm

Build the initial quantized matrix $\tilde{J}$ of $J(\mathbf{x}_0)$ with parameters L, b and η_q.
for i from 0 to T **do**
  Solve the linear least-square problem (3.6) using nsLSQR with parameters $\tilde{J}$ as the approximation matrix, current solution $\mathbf{x}_i$, regularization parameter λ_i, iterations t_{out} and t_{in}, tolerances η_t, η_r and η_s, integers r and s. Obtain a vector $\Delta\mathbf{x}_i$.
  Compute the ratio between of the predicted and actual reduction γ = ared/pred, using (6.2) and (6.2).
  if $\gamma < \mu_o$ **then**
    $\lambda_{i+1} := \omega_i \lambda_i$.
    $\mathbf{x}_{i+1} := \mathbf{x}_i$.
  else if $\mu_0 \leq \gamma < \mu_l$ **then**
    $\lambda_{i+1} := \omega_i \lambda_i$.
    $\mathbf{x}_{i+1} := \mathbf{x}_i + \Delta\mathbf{x}_i$.
  else if $\mu_0 \leq \gamma < \mu_l$ **then**
    $\lambda_{i+1} := \lambda_i$.
    $\mathbf{x}_{i+1} := \mathbf{x}_i + \Delta\mathbf{x}_i$.
  else
    $\lambda_{i+1} := \omega_d \lambda_i$.
    $\mathbf{x}_{i+1} := \mathbf{x}_i + \Delta\mathbf{x}_i$.
  end if
  if The step $\Delta\mathbf{x}_i$ was accepted **then**
    Build the quantized matrix $\tilde{J}$ of $J(\mathbf{x}_{i+1})$ with parameters L, b and η_q.
  end if
  if Equation 6.1 holds for η or Equation 6.2 holds for η_d and d **then**
    Return the solution $\mathbf{x}_{i+1}$
  end if
end for
Return the current solution $\mathbf{x}_{i+1}$.

Chapter 7

Numerical Experiments

In this section, we explain and show the results obtained by our numerical experiments. Different numerical experiments will be performed to evaluate different aspect of our proposed method. The following experiments will be performed:

- Section 7.1 analyzes the accuracy of the quantization approximation.
- Section 7.2 quantifies the induced error in the computation of the matrix-vector product with $\tilde{J}$.
- Section 7.3 evaluates the performance of the nsLSQR method when the number of bits used for the quantized matrix is fixed, for several matrices.
- Section 7.4 evaluates the performance of the nsLSQR method for different matrices using different number of bits combinations.
- Section 7.5 shows the difference between GMRes and nsLSQR.
- Section 7.6 analyzes the memory usage versus execution time of LSQR, using an explicitly stored matrix and a matrix-free approach, and nsLSQR with different quantized matrices.
- Finally, section 7.7 analyzes the performance of the proposed lm-nsLSQR method.

The set of functions used in our experiments are different in nature and structure, and were chosen to follow different patterns, density or statistical distribution. The set of matrices selected corresponds to a mixture between test cases found in the literature and test cases proposed. Notice that the Jacobian matrices are obtained from a finite difference approximation (2.2), given $\mathbf{F}(\mathbf{x})$. In particular, the set of linear and non-linear functions studied are the following:

(i) A linear function $\mathbf{F}(\mathbf{x}) = \mathbf{b} - A\,\mathbf{x}$, where the coefficients of A and $\mathbf{b}$ are normally distributed as $\mathcal{N}(0, 1)$. The size of the matrix A is $m \times n$. It is important to mention that this matrix is generated efficiently and deterministically each time it is needed, to avoid its storage. This test will be denoted as *Normal problem.*

(ii) A linear function $\mathbf{F}(\mathbf{x}) = \mathbf{b} - A\,\mathbf{x}$, where the coefficients of A and $\mathbf{b}$ are uniformly distributed in $[-1, 1]$. The size of the matrix A is $m \times n$. Same than for the Normal problem, the matrix is not stored but generated when required in a form of matrix-vector product. This test will be denoted as *Uniform problem*.

(iii) The *Quadratic Function* proposed by Toint Toint 1987. The size of the problem is given by $m = 6(n - 3)$. This test will be denoted as *Sparse problem I*.

(iv) The *Diagonal function pre-multiplied by a quasi-orthogonal matrix*, which is used in Cruz and Raydan 2003. The size of the problem is given by $m = n$. This problem will be denoted as *Sparse problem II*.

(v) A modification of the *Trigonometric function*, based on Moré et al. work Moré 1978. It is defined by m functions and n input variables, where just $m \geq n$ is required. The function is defined as follows,

$$f_k^{(1)}(\mathbf{x}) = n + k\,(1 - \cos(x_{i+1})) - \sin(x_{i+1}) - \sum_{j=1}^{n} \cos^{p+1}(x_j),$$

where $k = n\,p + i$ for $i < n$. This test will be denoted as *Dense problem I*.

(vi) A logarithmic function, defined by

$$f_k^{(2)}(\mathbf{x}) = x_{i+1}^{p+1}\,\log\left(\sum_{j=1}^{n} x_j^2 + 1\right) + x_{i+1},$$

where $k = n\,p + i$ for $i < n$. The value of m and n is arbitrary, as long as $m \geq n$ This test will be denoted as *Dense problem II*.

In all of experiments, unless explicitly specified other value, the vector used to evaluate the Jacobian matrix is a random uniformly distributed vector in $[-1, 1]$. Regarding the problem size, the value of m and n was chosen to require approximately 32 [GB] of memory for a fully stored matrix J. For clarity, we remark that such matrix is never constructed nor stored explicitly. If the matrix were to be stored explicitly, it will require 32 [GB] of memory. Specifically, the dimension used for the test cases are the following, unless otherwise stated:

- For the normal and uniform problem, the dimensions were $m = 80000$ and $n = 50000$.
- For the sparse problem I, the dimensions of the Jacobian matrix was $m = 155982$ and $n = 26000$.
- For the sparse problem II, the dimension of the Jacobian matrix was $m = n = 64000$.
- Finally, for the dense problem I and II, the dimension of the Jacobian matrix were $m = 80000$ and $n = 50000$.

7.1 Accuracy of the quantized approximation

This first experiment makes a direct comparison between the Jacobian matrix of the used test problems and its quantized approximation. The Jacobian matrix that is considered to be the reference matrix is computed by means of a finite-difference approximation. The quantized approximation is constructed for different set of layers and bits. In particular, the layers were from 1 to 4 and the number of bits were 2, 4, 6 and 8 for each layer. The computed error, for a matrix J and its quantized version $\tilde{J}_{b,L}$, where b represents the number of bits used in each layer and L is the number of layers used, is computed as,

$$\frac{\|J - \tilde{J}_{b,L}\|_F}{\|J\|_F}, \tag{7.1}$$

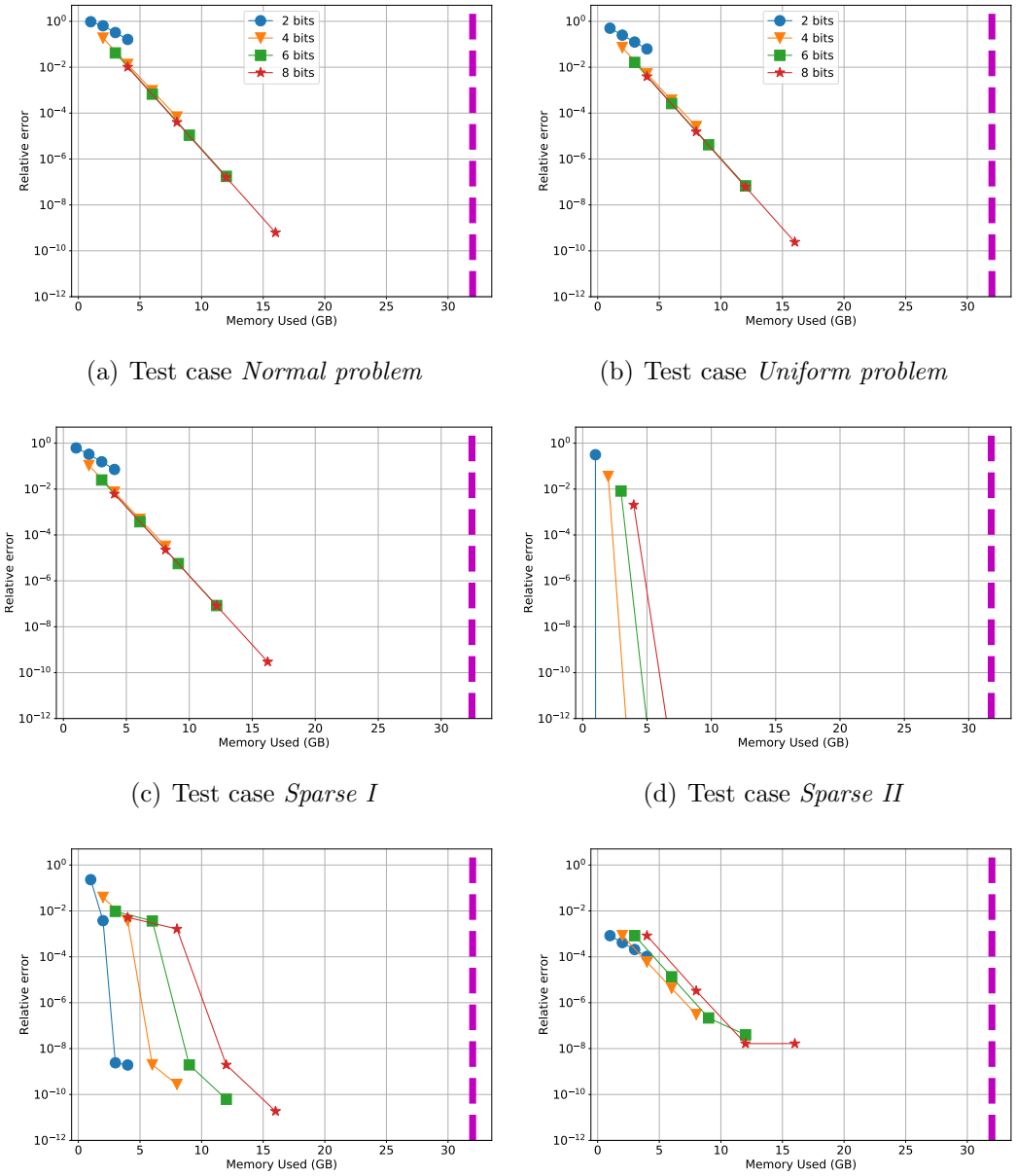
that is, the relative error using the Frobenius norm. For the normal and uniform problem, since the structure of the matrix has a stochastic construction, the experiment was realized 25 times, each one with a different normally and uniformly distributed matrix. Then, a mean error was computed.

The Figure 7.1 shows the relative error (7.1) as a function of the physical memory used to represent the quantized Jacobian matrix $\tilde{J}_{b,L}$ for each test case. For reference purposes, on the right side of the plots, a magenta dashed line is included to show the total memory that would have been required to store the matrix in double precision, which is approximately 32 [GB] for each test case. The standard deviation for the experiments was not included in the plots since it is imperceptible in the scale used. It goes from 10^{-4} for 1 layer to 10^{-12} for 4 layers.

A clear conclusion from 7.1 is that using two bits, although is more cheap in terms of memory usage, its performance is very poor. Its poor performance is expected, since using only two bits per element implies that we are able to represent just 4 values. Our approach discard one of these values, so we have just 3 numbers left. These are $-P$, 0 and P. In fact, using a 2 bit quantization approximation is more close to a sparsification procedure. The reduce of error compared to use 4 bits, which is just a few more bits per element is significantly. Some of the presented figures shows important information that need to be remarked:

- Figure 7.1(c) shows the relative error for the Sparse I problem. Note that although the Jacobian of this function is sparse, its performance is not so good as expected. The reason for this is that its sparsity is more regular in the row-wise direction than the column-wise direction. The quantized approximation proposed quantize in a column-wise direction, which is affected by the sparsity direction. Nevertheless, the relative error achieved is less than 10^{-8} using half of the memory required for full storage of the original matrix. However, it is clearly more convenient in this case to use a direct sparse representation of the problem. The purpose of this test case is to show that the proposed quantization can incrementally capture sparsity patterns.

- Figure 7.1(d) shows the relative error for the Sparse II problem. In this case, the quantized approximation captures the sparsity of the Jacobian matrix,



(a) Test case *Normal problem*

(b) Test case *Uniform problem*

(c) Test case *Sparse I*

(d) Test case *Sparse II*

(e) Test case *Dense I*

(f) Test case *Dense II*

Figure 7.1: Mean relative error (7.1) for each test case between the Jacobian matrix J and its quantized approximation $\tilde{J}_{b,L}$, where b denotes the number of bits used per layer and L the number of layers. The type of markers indicate the number of bits used, and the quantity of markers from left to right represent the number of layers used.

achieving a significant reduce in the error just by using two layers, regardless of the bits for each layer. This shows that some sparsity patterns can be

captured by our proposed method.

- Figure 7.1(e) shows the relative error for the test case *Dense matrix I*. The performance is very competitive only using three layers, regardless of the number of bits. The steep reduction of the relative error for the third layer is due to the structure of the Jacobian matrix of this test case. Although the matrix is dense, it has a fixed pattern in that most values are equal. This pattern is well approximated by the layered system proposed.

- Finally, the last test case of this section is shown in Figure 7.1(f). Test case *Dense matrix II* computes the Jacobian matrix of the linear combination of logarithmic functions. The outcome may seem slightly similar to the outcomes obtained for test case *Normal matrix* and *Sparse matrix*, but they differ in this case since a greater number of layers with few bits is better than few layers with more bits. For instance, three layers with 4 bits perform better than two layers with 6 bits, even when the memory requirements are equal. This means that in this test case the quantization approximates better J by increasing the number of layers rather than by increasing the number of bits of each layer.

As a general conclusion, the plots of Figure 7.1 shows that a relative error close to 10^{-6} is possible to achieve using $\frac{1}{3}$ of the memory required by the full double precision matrix.

7.2 Accuracy of the product using the quantized approximation

In the previous section, the quantized approximation was evaluated in its accuracy as an approximation matrix. In this subsection, we test the quantized matrix over its desired context, i.e. the accuracy of the transpose of the quantized matrix times a vector, compared to the vector $J^T(\mathbf{x}_i)\mathbf{w}$. To measure the error, 50 matrix transpose-vector products are computed for each test case. Then, the mean error is computed. Similar to Equation (7.1), we compute the error as follows:

$$\frac{\|J^T\,\mathbf{v}_j - \tilde{J}^T_{b_i,L}\,\mathbf{v}_j\|}{\|J^T\,\mathbf{v}_j\|}, \tag{7.2}$$

where $\mathbf{v}_j$, for $j \in \{1, 2, \ldots, 50\}$, are randomly generated vectors where their coefficients follow a Uniform$(0, 1)$ distribution.

The exact value of $J^T\,\mathbf{v}_j$, for the purpose of this experiment, is computed following Sanhueza et al. Sanhueza and Torres 2017 procedure. The expression used to compute the product $J^T\,\mathbf{v}_j$ is the following,

$$J^T(\mathbf{x}_i)\mathbf{w} \approx= \sum_{i=1}^{n} \langle J(\mathbf{x}_i)\mathbf{q}_i, \mathbf{w}\rangle \mathbf{q}_i, \tag{7.3}$$

where $\mathbf{q}_i$ for $i \in \{1, 2, \ldots, n\}$ is an orthonormal basis for $\mathbb{R}^n$. In our experiments, the orthonormal basis is the canonical basis. This approximation is matrix-free but demands a high computation effort since it requires several matrix-vector product of the Jacobian matrix, which is computed using a finite difference approach and hence, it requires several function evaluations. Notice this is exactly the procedure we want to avoid, but we need to use it here to obtain an accurate approximation of $J^T \mathbf{v}_j$ for comparison. In this numerical experiment, we use the same values of number of bits b and number of layers L already used in the previous section.

Same than for section 7.1, the purple dashed line represent the memory required for an explicit double-precision matrix storage. The Figure 7.2 shows the results for all test cases. The X-axis represent the memory used, while the Y-axis represents the relative error. As expected from the results in section 7.2, using more bits and/or layers increase the mean accuracy of the approximation, but increase the memory requirements. Also, like in section 7.2, the standard deviation for the experiments were not included in the plots since they are also imperceptible. It goes from 10^{-3} for 1 layer to 10^{-14} for 4 layers.

We observe that a relative error close to 10^{-5} is accomplished using less than a $\frac{1}{3}$ of the memory compared to the storage of the Jacobian matrix explicitly. Figures 7.2(e) and 7.2(f) show an stagnation of the error when using only one layer, but a great improvement is obtained as we increase the number of layers.

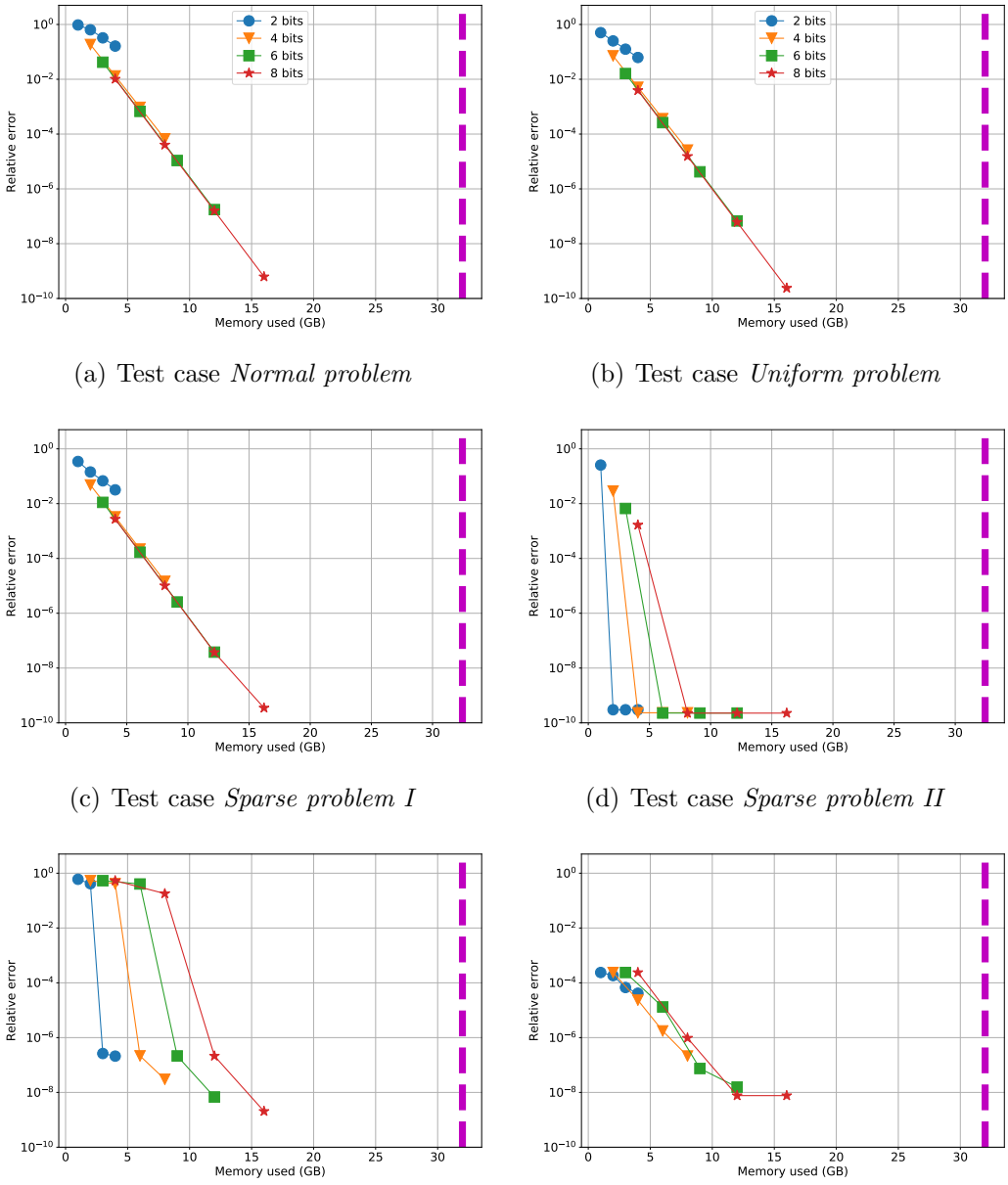
The conclusions from this experiments are the following,

- The accuracy of the approximation of the Jacobian matrix J by $\tilde{J}^T_{b,L}$ translates into the product matrix-vector.
- The usage of more than one layers may improve the accuracy drastically.
- A good approximation is achievable using significantly less memory than an explicit stored matrix.

7.3 nsLSQR using a fixed number of bits for the quantized matrix

In this section we will test the nsLSQR with the quantized approximation, which is the main contribution of this thesis. We know that the LSQR is a popular and well-known method. One of the main question that arise in our proposed nsLSQR method, combined with the quantized approximation, is how it performs compared to LSQR, since in nsLSQR we use an approximated matrix and the relation built in the method are different. For this reason, in this chapter and the afterward we will test the nsLSQR method against the LSQR to solve a least-square problem involving the defined functions in our test cases. This means finding $\mathbf{y}$ in the following minimization,

$$\min_{\mathbf{y}\in\mathbb{R}^n} \|\mathbf{F}(\mathbf{x}_i) - J(\mathbf{x}_i)\mathbf{y}\|^2 + \lambda\|\mathbf{y}\|^2,$$



(a) Test case *Normal problem*

(b) Test case *Uniform problem*

(c) Test case *Sparse problem I*

(d) Test case *Sparse problem II*

(e) Test case *Dense problem I*

(f) Test case *Dense problem II*

Figure 7.2: Relative error (7.2) for each test case between the Jacobian matrix J^T times $\mathbf{v}_j$ and its quantized approximation $\tilde{J}^T_{b,L}$ times $\mathbf{v}_j$, where b denotes the number of bits used per layer and L the number of layers. The type of markers indicate the number of bits used, and the quantity of markers from left to right represent the number of layers used.

for a uniformly distributed vector $\mathbf{x}_i$. Unlike the previous numerical experiment, in this case we only perform the computation once per test case. To quantify the

quality of the approximations of $\mathbf{y}$ found, we measure the following relative residual,

$$\frac{\|\mathbf{F}(\mathbf{x}_i) - J(\mathbf{x}_i)\,\mathbf{y}\|}{\|\mathbf{F}(\mathbf{x}_i)\|}, \tag{7.4}$$

where $\mathbf{y}$ represents the solution obtained in each test case.

To perform the comparison, we use the fact that when we have the exact value for $J^T\,\mathbf{v}$, nsLSQR is mathematically equivalent to LSQR. In particular, we will use the computationally expensive approximation of Sanhueza et al. Sanhueza and Torres 2017 to obtain the value of $J^T\,\mathbf{v}$ with nsLSQR to recover LSQR. In our numerical experiments, this is denoted as *Matrix-free LSQR*.

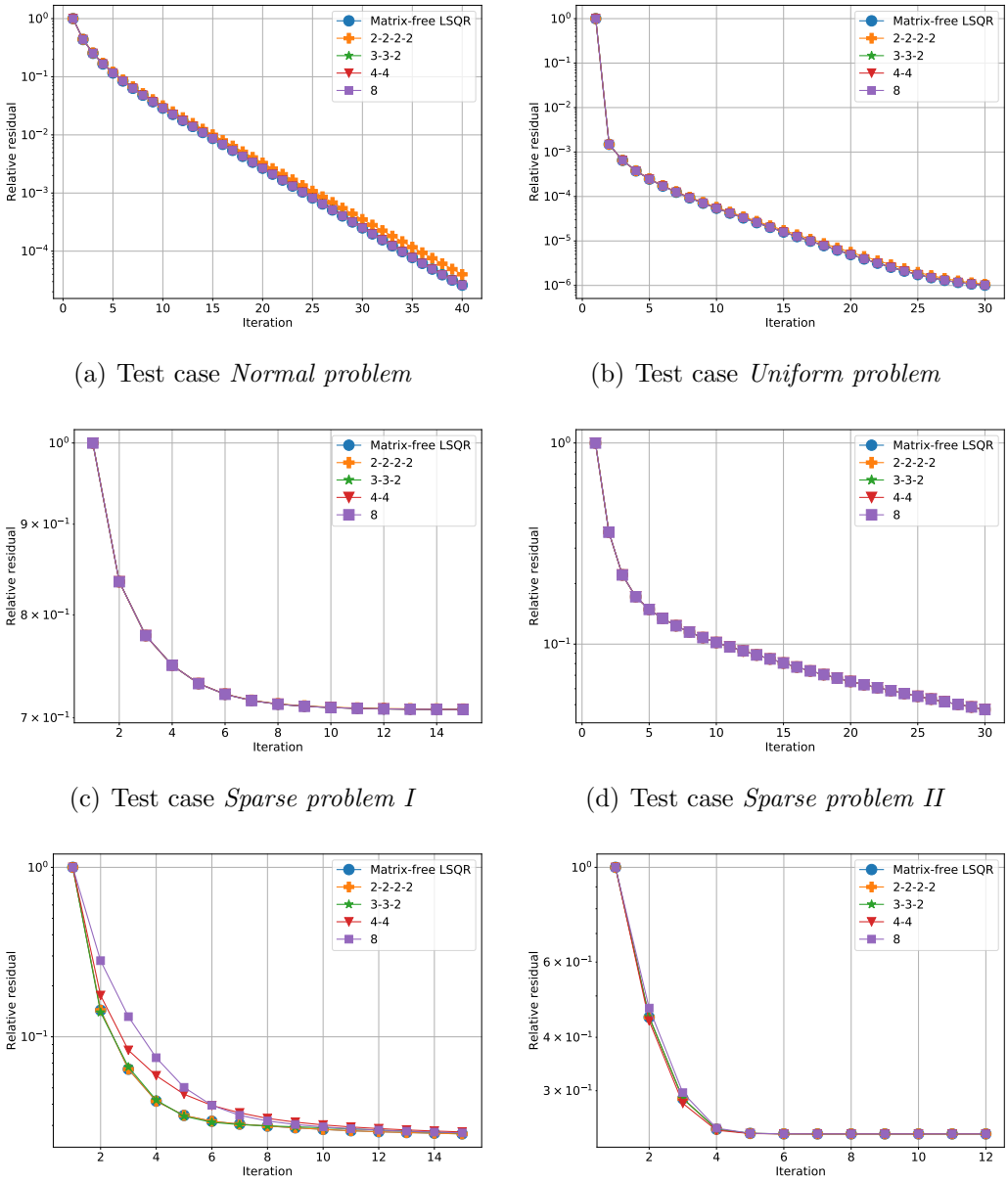
In this experiment, we will fix total number of bits that will be used in our quantized approximation. In particular, we will use 8 bits per coefficient with different combinations. This implies that the quantized matrix uses 8 times less memory than an explicit stored matrix using double precision. The combinations selected are the followings:

- Four layers of 2 bits, denoted as $2-2-2-2$ in the legends of Figure 7.3.
- Two layers of 3 bits and one layer of 2 bits, denoted as $3-3-2$.
- Two layers of 4 bits, denoted as $4-4$.
- One layer of 8 bits, denoted as 8.

For all the numerical experiments, we used a damping factor $\lambda = 10^{-5}$, for simplicity. The algorithm was executed with 50 restarts and 500 iterations per restart. The tolerances described in chapter 4.6 were set to 10^{-8}, and we also stop them after 50 iterations if no significant progress is made. Note that, theoretically, the algorithm decrease the residual monotonically in exact arithmetic, according to chapter 4.5. However, we observe numerically that orthogonality of the matrices $\tilde{V}_k$ and $\tilde{U}_k$ is lost, and this implies that the residual may increase after reaching saturation. At that point, the method is considered that has already converged and it is stopped.

Figure 7.3 shows the numerical result obtained for the 6 test cases, described at the beginning of this chapter. Notice that in Figure 7.3 we show a few numbers of iterations since saturation was quickly achieved. Using *Matrix-free LSQR*, the final relative residual of the normal equations range from 10^{-6} to 10^{-9} depending on the problem.

An important conclusion from Figure 7.3 is that even in the case when we only use 8 bits per element, the performance of a quantized matrix in nsLSQR is similar to the Matrix-free LSQR in all test cases. Moreover, in the case Figure 7.3(e), we observe a similar behavior in the relative residual for all the combinations of number of layers and number of bits per layer even though that combination with few layers did not show good results in the previous experiments, see Figure 7.1(e) and 7.2(e). Another important remark here is that nsLSQR, using the parameters described in



(a) Test case *Normal problem*

(b) Test case *Uniform problem*

(c) Test case *Sparse problem I*

(d) Test case *Sparse problem II*

(e) Test case *Dense problem I*

(f) Test case *Dense problem II*

Figure 7.3: Relative error (7.4) of the nsLSQR method for the different test cases, achieved at each iteration. The type of marker defines the specific configuration of bits to quantize the Jacobian matrix.

this section, use no more than 5GB, more than 6 times less than a fully stored matrix, which use approximately 32GB. Considering this and the results from Figure 7.3, we can conclude that nsLSQR may achieve similar residuals to LSQR but using considerable less memory than fully stored the Jacobian matrix.

7.4 nsLSQR using a variable number of bits for the quantized matrix

In the previous section, nsLSQR was tested using different combinations of bits summing a total of 8 bit per coefficient. One of the main conclusion from the previous experiment is that a good performance can be achieved using almost any combination of bits, giving a total of 8 bits although this is also problem dependent. However, one question that now arise is how much is affected nsLSQR by using a low-bit approximation? That is, if an small number of bits are used, how much decrease the performance of nsLSQR? The experiments in this section will try to answer such questions.

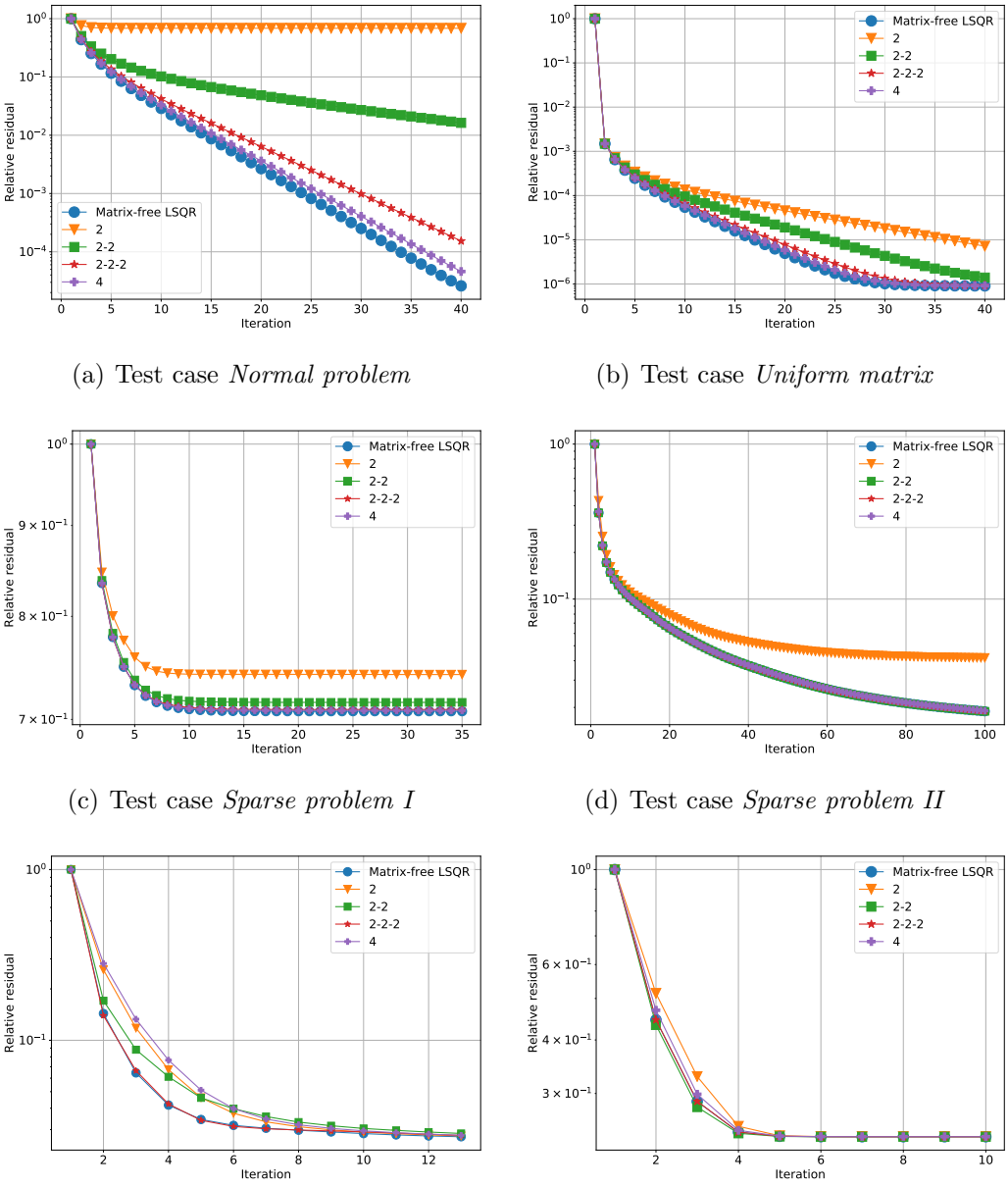
In this part we will test nsLSQR using quantized matrices that use a different number of bits per coefficients. In particular, the combinations selected are one to three layers of 2 bits, and one layer of 4 bits. In general, the aim is to use less than 8 bits for the quantized matrix. The setup of the experiment will be the same than for section (7.3) regarding the parameters, test cases and residual measure.

Figure 7.4 shows the numerical results obtained for all of the test cases. Same than for section 7.3, the experiments were executed for more iterations than the ones shown in Figure 7.4, but they were limited due to saturation of residual.

The main conclusion of this experiment is that the number of bits may impact considerably the performance of the nsLSQR method. For example, for Figure 7.4(e) and Figure 7.4(f), the impact of a low-bit approximation is not significant. Even for the Uniform problem, whose results are shown in the Figure 7.4(b), the results for the combination 2 and $2-2$ are relatively acceptable since its linear convergence makes the result closer to the other combinations. However, for the rest of the cases, the impact of the usage of a low-bit approximation is evident. In particular, we have that

- For the Sparse problem I and II, whose results are shown in Figure 7.4(c) and Figure 7.4(d), respectively, an early saturation is appreciated. This is reflected in the fact that for early iterations, the method decrease the residual at a rate similar to the other execution of nsLSQR. However at some point, the execution of nsLSQR that use a 2 bit matrix saturates and the reduction of the residual is extremely slow. Numerically, the residual is decreased at each iteration but at small quantities that are almost imperceptible. Even the combination $2-2$ achieve an early saturation for the Sparse problem I.

- For the Normal problem, represented in Figure 7.4(a), a similar results can be appreciated. Its result show how the convergence curve decrease faster when more bits are used for the quantized approximation.



(a) Test case *Normal problem*

(b) Test case *Uniform matrix*

(c) Test case *Sparse problem I*

(d) Test case *Sparse problem II*

(e) Test case *Dense problem I*

(f) Test case *Dense problem II*

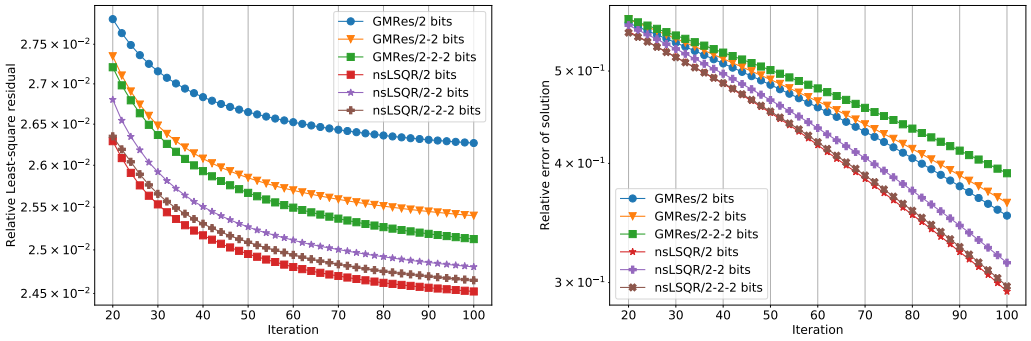
Figure 7.4: Relative error (7.4) of the nsLSQR method for the different test cases, achieved at each iteration. The type of marker defines a specific configuration of bits to quantize a matrix.

7.5 Comparison between nsLSQR and GMRes

In section 4.4, the relationship and differences between GMRes and nsLSQR were presented. In particular, we showed that both methods use the same Krylov subspace

and that the main difference is in the inner linear least-square problem solved. These numerical experiments complement the theoretical analysis included in chapter 4.3.

In nsLSQR, as a linear least-square solver, the problem solved is defined by (4.8), if we consider a non-zero regularization parameter. In general, to use GMRes to solve a least-square problem, we can solve its least-square problem. However, since we are using a quantized matrix for the transpose Jacobian matrix, we cannot use the normal equations directly. As shown in 4.4, the equivalent of the normal equations for the quantized matrix is given by (4.16). This would be what GMRes would solve. Therefore, although they use the same Krylov subspace to approximate a solution, the computed approximation does not need to be same, since the original problem and the inner least-square problem that need to be solved differs. To show this, we will use the Dense problem I.



(a) Least-square residual for nsLSQR and GMRes.

(b) Relative error between an approximated solution using LSQR and the approximated solution computed at each iteration using nsLSQR and GMRes.

Figure 7.5: Results obtained using GMRes and nsLSQR to solve the same test problem. Different markers represent a combination of a method and a specific number of bits for the quantized matrix. For visualization purpose, the plot begins at the 20th iteration and the markers are included every two iterations

Figure 7.5 shows the results obtained when using GMRes and nsLSQR to solve the same problem. We remark again that even when the problem is the same, the path to pursue the solution differs. If $\mathbf{y}_k$ is the current computed solution, then the Figure 7.5(a) shows the following relative residual

$$\frac{\|\mathbf{F}(\mathbf{x}_i) - J(\mathbf{x}_i)\,\mathbf{y}_k\|}{\|\mathbf{F}(\mathbf{x}_i)\|},$$

at each iteration, for GMRes and nsLSQR. Figure 7.5(b) shows the following result,

$$\frac{\|\Delta\mathbf{x} - \mathbf{y}_k\|}{\|\Delta\mathbf{x}\|},$$

where $\Delta\mathbf{x}$ is a computed solution using a matrix-free LSQR, and $\mathbf{y}_k$ is the k-th approximated solution of GMRes or nsLSQR. As shown in 7.5(a), the residual computed at each iteration is similar for each method. A further difference is noted in Figure 7.5(b), which shows how the difference between the computed solutions increases with each new iteration. Although the difference is small, Figure 7.5 shows that effectively there is a difference in the computation of the solution between both methods, as expected given the results obtained in section 4.5. In both experiments, the performance of the nsLSQR method is slightly better compared to GMRes. We note that in Figure 7.5(a) the configuration with a lower value of the relative least-square residual is effectively nsLSQR with a quantization of 2 bits. In Figure 7.5(b) we observe that the best outcomes come from using nsLSQR with one layer of 2 bits and 3 layers of 2 bits. In all cases, nsLSQR outperforms GMRes.

7.6 Memory usage of nsLSQR and LSQR

In this section, we will explain the experiment realized to show the difference in the memory requirements and execution time between LSQR and nsLSQR. This is indeed, an important question since we need to visualize what is gained and what is the cost of using nsLSQR instead of LSQR.

The result of this section must be connected with the outcomes of section 7.4, where we studied numerically the effect of using a quantized approximation with a different number of bits by changing the number of layers. As shown in section 7.1 and 7.2, we observed that if we use more bits per coefficient of the quantized matrix, we obtain a better approximation. However, using more bits will require more memory, as explained in section 5.4. Thus, there is a trade-off between using less-memory and getting an early saturation of the relative residual or faster convergence.

In this experiment, nsLSQR with different bit usage will be compared against LSQR. To be precise in the use of LSQR in this numerical experiment, we need to explicitly describe the two approaches we will use to solve a least-square problem. Since the context of the least-square solver is to be used in the Levenberg-Marquardt solver, there are some constrains. For instance, we do not have the Jacobian matrix but only its function $\mathbf{F}$. These two approaches are the following:

(i) Build and then explicitly store an approximation of the Jacobian matrix J. Since we do not have explicitly the Jacobian matrix, we will approximate it using a finite difference approximation for each column, which is easy to obtain using the canonical vectors. Since the LSQR algorithm requires a small amount of memory Paige and M. A. Saunders 1982c, the total memory usage of this approach is mainly determined by the memory required for the explicit storage of the approximated Jacobian matrix. This will be denoted as *Explicit Jacobian LSQR* in our experiment.

(ii) Use of a finite difference approximation to compute the Jacobian matrix-vector product, and use a matrix-free approximation for the product of the transpose

of the Jacobian matrix and a vector Sanhueza and Torres 2017. This product is computed as,

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

where $\mathbf{q}_i$ for $i \in \{1, 2, \ldots, n\}$ is an orthonormal basis for $\mathbb{R}^n$. In our case, the canonical basis is used. The memory usage of this approach is considerably less than the previous approach since the Jacobian does not need to be stored. However, this is a more computational intensive approach since it requires several matrix-vector products and function evaluations. This is what before we called *Matrix-free LSQR*. We have repeated the definition of *Matrix-free LSQR* for completeness.

It is important to mention that the quantized approximation of the Jacobian matrix requires less memory than the explicit Jacobian matrix, but the computational cost of a matrix transpose-vector product is higher using our compression and decompression method, as explained in section 5.3. To compare LSQR with nsLSQR, considering the two approaches presented, we solve the same problem and plot the computation time vs the memory requirements needed by each method. In this case, we solved the Dense problem I, i.e. the extended trigonometric function. The dimension of the Jacobian matrix here is 51200×32000, which requires approximately 13 [GB] for full storage. We use here a smaller matrix than before because we need to store it explicitly, which was not needed before. The parameters for nsLSQR consider five quantization matrices. The matrices use $2, 4, 8, 16$ and 32 bits per coefficient, respectively. All methods are iterated 50 times.

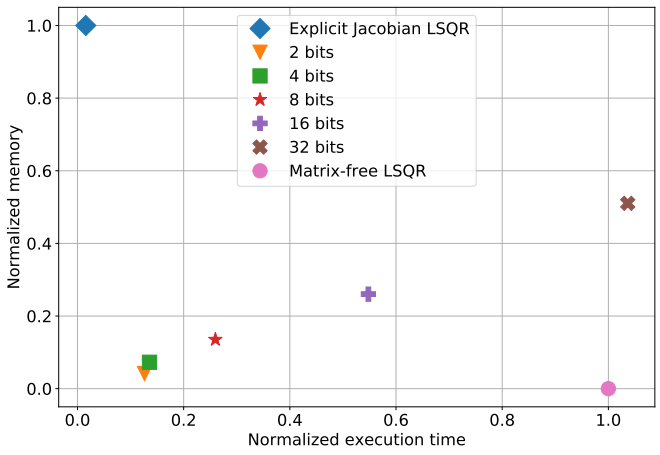


Figure 7.6: Theoretical memory usage and execution time of nsLSQR and two versions of LSQR. The memory used and the execution time required are normalized by the outcomes obtained by the two versions of LSQR used.

In Figure 7.6 we observe the outcome of this numerical experiment. As highlighted before, the *Explicit Jacobian LSQR* algorithm is the one that uses the great-

est amount of memory but, at the same time, it is the fastest. Its high memory requirement is due to the explicit storage used, and its low computational time is due to that its matrix-vector product is faster since the matrix is explicitly available. However, recall that in this numerical experiments we were able to use the *Explicit Jacobian LSQR* since the problem was relatively small. For larger problems, this is unfeasible. On the other hand, the *Matrix-free LSQR* is the algorithm that uses the least amount of memory, but it has one of the longest execution times. This is due to the high computational cost of computing the matrix-free transpose of the Jacobian matrix times a vector. Note also that the execution time using a quantized matrix of 32 bits is higher than a matrix-free approach. This is mostly due to the overhead of the implementation of the quantization method. In particular, since our quantized implementation has a maximum of 8 bits per coefficient, to compute a 32 bit quantized matrix we use four layers of 8 bits, which adds an additional complexity to the procedure. For all the configurations of nsLSQR, we clearly observe that the higher the number of bits used, the more time it needs. This is also mainly related to the overhead required in the quantization procedure. Although using fewer bits for quantization reduces considerably the execution time, results for chapter 7.4 show that the combination of bits used in the quantized matrix affects directly the performance of nsLSQR. For instance, using one layer of 2 bits may be faster but it may saturate the relative residual obtained by nsLSQR early, as show in section 7.4. Therefore, there is a trade-off between a possible early saturation of the relative residual and having a faster computation.

7.7 Performance of lm-nsLSQR method

This last section of experiments will focus in the missing part of our thesis: the nonlinear solver. As explained in chapter 3 and chapter 6, our method is based on the Levenberg-Marquardt method. The main issue of this method to be used in large-scale problem is the solution of the inner least-square. However, the effectiveness of the quantized approximation and nsLSQR was showed in previous experiments. Therefore, the only missing part is to combine all methods proposed in this thesis to build the nonlinear solver.

The lm-nsLSQR method was used over all of the test cases, even the linear ones. Recall that even that Levenberg-Marquardt is used for nonlinear problems, is capable of solving linear ones.

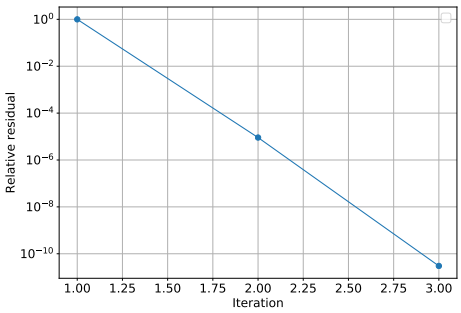
In this case, we have enough information from previous experiments to select a bit combination for the quantized matrix. In particular, in this experiments, the quantized matrix was constructed using three layers: two layers of 3 bits and one layer of 2 bits.

For the parameters for the nsLSQR method inside the lm-nsLSQR method, the method was executed for 20 restarts and 500 iteration per restart. The relative residual tolerance for the nsLSQr method was set to 10^{-8} and the tolerances for saturation and progress solution to 10^{-10}, to have some guarantee of the precision in the

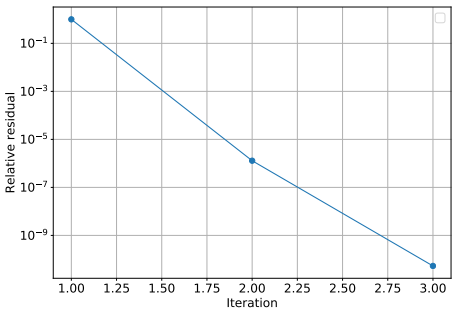
computed solution. nsLSQR was stopped if no progress in the solution was achieved for 30 iterations, and for the saturation stopping criteria, the last 100 residual were considered. The damping parameter is given by the Levenberg-Marquardt method. For the initial guess, the zero vector was always used.

In general, the Levenberg-Marquardt method or similar Newton-based methods converge faster and in less iteration if the initial guess is close to the solution. However, modifying accordingly the damping factor, the Levenberg-Marquardt should be capable of converging to a local minimum even if the initial guess is far from the solution. For this experiment, the initial guess was a vector whose coefficients where uniformly distributed in $[-1, 1]$ to use a general vector not related to the problem and thus, not necessarily close to the solution. The tolerance for the relative residual was set to 10^{-6}. For the progress stopping criteria, if the relative error of the last 100 computed solution is less than the tolerance 10^{-10}, the method also stops. The number of iterations was set to 10000, as a safe-guard method. As for the damping factor updating parameters, the following values were used: $\lambda_0 = 10^{-2}, \lambda_{\min} = 10^{-10}, \omega_d = 0.1, \omega_i = 10, \mu_0 = 10^{-4}, \mu_l = 0.25$ and $\mu_h = 0.75$. The values of μ_0, μ_l and μ_h comes from Moré 1978. The value of λ_0 and $\lambda_{\min}$ are arbitrary, although a better choice for an starting λ_0 may reduce the number of iterations. Finally, the values for $\omega_d = 0.1$ and $\omega_i = 10$ are taken from Marquardt 1963.

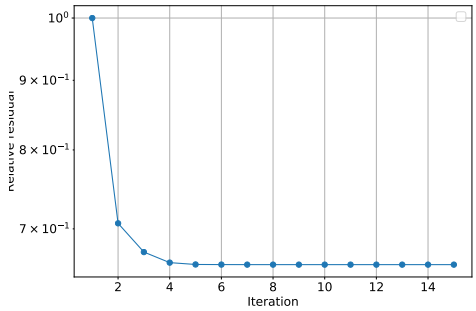
Figure 7.7 shows the result of the experiment. Each plot shows the residual of $\frac{\|\mathbf{F}(\mathbf{x}_i)\|}{\|\mathbf{F}(\mathbf{x}_0)\|}$ at each iteration. Note that plots in Figure 7.7(c) and Figure 7.7(d) does not converge to 0, since these problems are not zero-residual. However, it reach a local minimum of the residual function. The rest of problems are zero-residual and is expected to converge to zero, which is the case in our results, where the tolerance 10^{-6} is achieved for those problems.



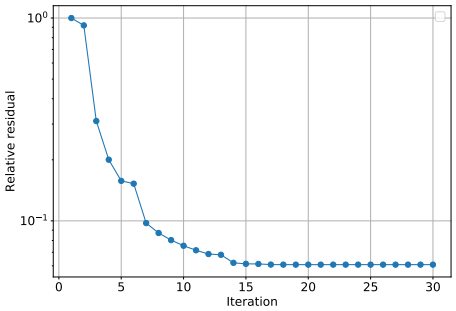
(a) Test case *Normal problem*



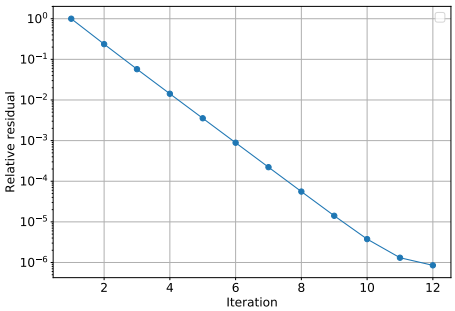
(b) Test case *Uniform matrix*



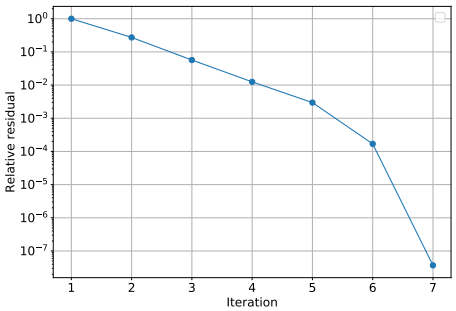
(c) Test case *Sparse problem I*



(d) Test case *Sparse problem II*



(e) Test case *Dense problem I*



(f) Test case *Dense problem II*

Figure 7.7: Normalized residual of the nonlinear system of equation, at each iteration of lm-nsLSQR, for the different test cases.

Chapter 8

Conclusion

The origin of this thesis proposal and topic was the solution of a large-scale overdetermined nonlinear system of equation. The work done in Sanhueza and Torres 2017 was a first step in the fulfillment of that objective. In that previous work, a fully-matrix free was developed and initially, the idea was to couple the matrix-free approximation with a Newton's method. However, although using a minimum amount of memory, its high computational time make the method impractical in a lot of scenarios, where the nonlinear function to be solved has a medium to high computational cost for a single evaluation. This thesis is an evolution of the previous work, and the main objective was achieved.

In the first sections, from section 1 and section 2, the problem and different method how to solve it were presented. Although different method exist, each one with different properties, many of them has requirements that are unavailable in a general overdetermined nonlinear system of equations, i.e. the availability of the nonlinear function only makes some components, like the gradient of the residual of the nonlinear function, hard to compute. This thesis proposed a nonlinear solver based on the Levenberg-Marquardt method, which was named as *lm-nsLSQR*. The Levenberg-Marquardt method use our proposed nsLSQR method, which was described in section 4, to solve its inner least-square problem. The main property of the proposed nsLSQR method is that does not require the exact product of the transpose of the linear matrix but only an approximation of it. To propose a even more robust method, a quantized approximation was proposed in section 5 that approximate the Jacobian matrix using a restricted-memory approach. This approximation use the Quantization technique to compute a low-memory approximation of the Jacobian matrix. This quantized approximation is used in nsLSQR to approximate the product of the transpose of the Jacobian matrix and a vector.

From section 1 to section 6, the aim is to present the mathematical development, equations and expression along with the algorithms required to visualize the proposed methods. In section 7, several numerical experiments were performed with the aim to validate the proposed components, i.e. validates the quantized approximate, the performance of nsLSQR as a linear least-square solver for large-scale problems, and the lm-nsLSQR method as a competitive solver for large-scale nonlinear overdetermined

problems. The first experiments performed tested the accuracy of the quantized matrix in two context, as an approximated matrix and in the product of its transpose and a vector, with respect to the original matrix. Our experiments showed that the quantized approximation can obtain a low error for both application, if parameter are selected optimally. Moreover, relative errors close to 10^{-6} are easy to obtain using around the $\frac{1}{3}$ of the memory required by a full double precision matrix. In some problems, achieving small errors is possible using even less memory.

Later, the performance of the nsLSQR using a quantized matrix was tested. Two experiments were performed. The first one compared LSQR against several execution of nsLSQR with different quantized matrices, all of them using a total of 8 bits per coefficient. The second one compared LSQR against nsLSQR using quantized matrices whose bit usage varies from 2 bits to 6 bits. The performed tests shows interesting results. The most important one was that nsLSQR, combined with the quantized approximation, can handle a large-scale least-square problem using considerably less memory than a common least-square solver with a fully stored matrix. nsLSQR quickly reduce the residual of the least-square problem, even when using quantized matrices of around 6 bit in total. Even more, in our all test cases, when using 8 bits for the quantized matrix, nsLSQR showed almost no difference compared with LSQR. This is a very important result, since a quantized matrix using 8 bits per coefficient requires 8 times less memory than a fully double-precision matrix. That is, if a matrix of a least-square problem requires 32GB of storage, the problem can be solved by nsLSQR using around 4GB, which probably will fit in almost all machines around there. However, an important results also obtained in the experiments of subsection 7.4, is that the convergence of nsLSQR can be harmed if the approximation matrix performs poorly. In particular, if the quantized matrix uses too few bits, the nsLSQR may converge extremely slow or saturate too early the residual. Although this early saturation and slow-convergence was not observed in all our test cases, it need to be considered before setting the parameters for the quantized matrix.

If the quantized matrix produce an approximation whose error is relatively small, one is tempted to use it directly to solve the normal equations of the least-square using a more classical or traditional linear solver. In section 4 we proved that the nsLSQR reduce the residual of the least-square monotonically and also that the residual computed at the k-th iteration of nsLSQR is a lower bound of the residual computed at the k-th iteration computed by GMRes using the approximated normal equations. This is an important claim and this was tested in the experiment showed in subsection 7.5. Although we only tested this using one of our test cases and the difference is small, the results also showed that effectively the residual obtained by nsLSQR is smaller than the one obtained using GMRes. Although it was mathematically proved, the experiment was realized to confirm numerically this find.

In subsection 7.6 an important comparison was tested. For a large-scale problem, we know that is unfeasible to store the matrix explicitly. However, this is also the faster approach to use, for instance, the LSQR method. On the other hard,

one option is to use a fully matrix-free approach, using a method similar like the used in Sanhueza and Torres 2017. However, it is known that such approximation demands a high computational time. In subsection 7.6, an experiment was performed to visualize where nsLSQR can fit between this two sides of LSQR. The obtained results showed that effectively LSQR using an explicitly stored matrix performs faster than nsLSQR. However, nsLSQR use considerably less memory. On the other hand, nsLSQR also use more memory than a fully matrix-free LSQR. However, depending on how many bits are used for the quantized matrix, in general nsLSQR performs much faster than a matrix-free version of LSQR. In general, using 8 bits, which was the number of bits used in experiment presented in subsection 7.3, performs considerably faster than a matrix-free LSQR and using much less memory. The experiment showed how nsLSQR fits between these two approaches for using LSQR, how there is a trade-of in memory and execution time in the quantized approximation and finally, how nsLSQR is an interesting option for solving large-scale problem, considering the memory required and the computational execution time demanded.

Last, but not least, the final experiment, presented in subsection 7.7, showed the performance of the lm-nsLSQR solver. Since the effectiveness of nsLSQR was already showed, is expected that the proposed Levenberg-Marquardt method can minimize at each iteration the nonlinear problems used as test cases.

To summarize, we successfully developed

- An effective approximation procedure to approximate the Jacobian matrix. In particular, this matrix con be used in nsLSQR without compromising the convergence of the least-square solver.
- A linear least-square solver capable of handling large-scale matrices using the proposed quantized approach.
- A nonlinear solver that combined with our quantized approximation and the nsLSQR can handle effectively large-scale nonlinear problems.

Thus, we successfully accomplished the goals of this thesis.

Bibliography

[1] W. Jouha, A. E. Oualkadi, P. Dherbécourt, E. Joubert, and M. Masmoudi, "Silicon carbide power mosfet model: An accurate parameter extraction method based on the levenberg–marquardt algorithm", *IEEE Transactions on Power Electronics*, vol. 33, no. 11, pp. 9130–9133, 2018.

[2] P. Hu, G.-Y. Cao, X.-J. Zhu, and J. Li, "Modeling of a proton exchange membrane fuel cell based on the hybrid particle swarm optimization with levenberg–marquardt neural network", *Simulation Modelling Practice and Theory*, vol. 18, no. 5, pp. 574–588, 2010, ISSN: 1569-190X. DOI: https://doi.org/10.1016/j.simpat.2010.01.001. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1569190X1000002X.

[3] S. Pouyanfar, S. Sadiq, Y. Yan, H. Tian, Y. Tao, M. P. Reyes, M.-L. Shyu, S.-C. Chen, and S. S. Iyengar, "A survey on deep learning: Algorithms, techniques, and applications", *ACM Comput. Surv.*, vol. 51, no. 5, Sep. 2018, ISSN: 0360-0300.

[4] Y.-C. Du and A. Stephanus, "Levenberg-marquardt neural network algorithm for degree of arteriovenous fistula stenosis classification using a dual optical photoplethysmography sensor", *Sensors*, vol. 18, no. 7, p. 2322, Jul. 2018, ISSN: 1424-8220. DOI: 10.3390/s18072322. [Online]. Available: http://dx.doi.org/10.3390/s18072322.

[5] C. T. Kelley, *Iterative Methods for Optimization*. Society for Industrial and Applied Mathematics, 1999. DOI: 10.1137/1.9781611970920. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611970920. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611970920.

[6] J. Nocedal and S. Wright, *Numerical Optimization*, ser. Springer Series in Operations Research and Financial Engineering. Springer New York, 2000.

[7] E. Chong and S. Zak, *An Introduction to Optimization*, ser. Wiley Series in Discrete Mathematics and Optimization. Wiley, 2013, ISBN: 9781118515150.

[8] C. T. Kelley, *Iterative Methods for Linear and Nonlinear Equations*. Society for Industrial and Applied Mathematics, 1995. DOI: 10.1137/1.9781611970944. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611970944. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611970944.

[9] D. W. Marquardt, "An algorithm for least-squares estimation of nonlinear parameters", *Journal of the Society for Industrial and Applied Mathematics*, vol. 11, no. 2, pp. 431–441, 1963. DOI: 10.1137/0111030. eprint: https://doi.org/10.1137/0111030. [Online]. Available: https://doi.org/10.1137/0111030.

[10] Y. Lin, D. O'Malley, and V. V. Vesselinov, "A computationally efficient parallel levenberg-marquardt algorithm for highly parameterized inverse model analyses", *Water Resources Research*, vol. 52, no. 9, pp. 6948–6977, 2016. DOI: 10.1002/2016WR019028. eprint: https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1002/2016WR019028. [Online]. Available: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1002/2016WR019028.

[11] S. Henn, "A levenberg–marquardt scheme for nonlinear image registration", *BIT Numerical Mathematics*, vol. 43, pp. 743–759, Nov. 2003. DOI: 10.1023/B:BITN.0000009940.58397.98.

[12] S. Finsterle and M. B. Kowalsky, "A truncated levenberg–marquardt algorithm for the calibration of highly parameterized nonlinear models", *Computers & Geosciences*, vol. 37, no. 6, pp. 731–738, 2011, 2009 Transport of Unsaturated Groundwater and Heat Symposium, ISSN: 0098-3004. DOI: https://doi.org/10.1016/j.cageo.2010.11.005. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0098300410003729.

[13] J. Bao, C. K. W. Yu, J. Wang, Y. Hu, and J.-C. Yao, "Modified inexact Levenberg–Marquardt methods for solving nonlinear least squares problems", *Computational Optimization and Applications*, vol. 74, no. 2, pp. 547–582, Nov. 2019. DOI: 10.1007/s10589-019-00111-. [Online]. Available: https://ideas.repec.org/a/spr/coopap/v74y2019i2d10.1007_s10589-019-00111-y.html.

[14] M. Raydan, "The barzilai and borwein gradient method for the large scale unconstrained minimization problem", *SIAM Journal on Optimization*, vol. 7, no. 1, pp. 26–33, 1997. DOI: 10.1137/S1052623494266365. eprint: https://doi.org/10.1137/S1052623494266365. [Online]. Available: https://doi.org/10.1137/S1052623494266365.

[15] Y.-H. Dai and H. Zhang, "Adaptive two-point stepsize gradient algorithm", *Numerical Algorithms*, vol. 27, pp. 377–385, Aug. 2001. DOI: 10.1023/A:1013844413130.

[16] F. Biglari and M. Solimanpur, "Scaling on the spectral gradient method", *Journal of Optimization Theory and Applications*, vol. 158, Aug. 2013. DOI: 10.1007/s10957-012-0265-5.

[17] L. Hongwei, Z. Liu, and X. Dong, "A new adaptive barzilai and borwein method for unconstrained optimization", *Optimization Letters*, pp. 1–29, May 2017. DOI: 10.1007/s11590-017-1150-9.

[18] H. Mohammad and M. Waziri, "Structured two-point stepsize gradient methods for nonlinear least squares", *Journal of Optimization Theory and Applications*, vol. 181, Apr. 2019. DOI: 10.1007/s10957-018-1434-y.

[19] W. L. Cruz and M. Raydan, "Nonmonotone spectral methods for large-scale nonlinear systems", *Optimization Methods and Software*, vol. 18, no. 5, pp. 583–599, 2003. DOI: 10.1080/10556780310001610493.

[20] W. L. Cruz, J. M. Martínez, and M. Raydan, "Spectral residual method without gradient information for solving large-scale nonlinear systems of equations", *Mathematics of Computation*, vol. 75, no. 255, pp. 1429–1448, 2006, ISSN: 00255718, 10886842.

[21] W. Cheng, Y. Xiao, and Q.-J. Hu, "A family of derivative-free conjugate gradient methods for large-scale nonlinear systems of equations", *Journal of Computational and Applied Mathematics*, vol. 224, no. 1, pp. 11–19, 2009, ISSN: 0377-0427. DOI: https://doi.org/10.1016/j.cam.2008.03.050.

[22] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd. USA: Society for Industrial and Applied Mathematics, 2003, ISBN: 0898715342.

[23] Y.-H. Dai and C.-X. Kou, "A nonlinear conjugate gradient algorithm with an optimal property and an improved wolfe line search", *SIAM Journal on Optimization*, vol. 23, no. 1, pp. 296–320, 2013. DOI: 10.1137/100813026. eprint: https://doi.org/10.1137/100813026.

[24] G. Yuan, T. Li, and W. Hu, "A conjugate gradient algorithm for large-scale nonlinear equations and image restoration problems", *Applied Numerical Mathematics*, vol. 147, pp. 129–141, 2020, ISSN: 0168-9274. DOI: https://doi.org/10.1016/j.apnum.2019.08.022. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0168927419302314.

[25] T. Sauer, *Numerical Analysis*, 2nd. USA: Addison-Wesley Publishing Company, 2011, ISBN: 0321783670.

[26] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C (2nd Ed.): The Art of Scientific Computing*. USA: Cambridge University Press, 1992, ISBN: 0521431085.

[27] D. A. Knoll and D. E. Keyes, "Jacobian-free newton-krylov methods: A survey of approaches and applications", *J. Comput. Phys.*, vol. 193, no. 2, pp. 357–397, Jan. 2004, ISSN: 0021-9991. DOI: 10.1016/j.jcp.2003.08.010. [Online]. Available: https://doi.org/10.1016/j.jcp.2003.08.010.

[28] W. J. Leong, M. A. Hassan, and M. W. Yusuf, "A matrix-free quasi-newton method for solving large-scale nonlinear systems", *Computers & Mathematics with Applications*, vol. 62, no. 5, pp. 2354–2363, 2011, ISSN: 0898-1221. DOI: https://doi.org/10.1016/j.camwa.2011.07.023.

[29] M. [Noor] and M. Waseem, "Some iterative methods for solving a system of nonlinear equations", *Computers & Mathematics with Applications*, vol. 57, no. 1, pp. 101–106, 2009, ISSN: 0898-1221. DOI: https://doi.org/10.1016/j.camwa.2008.10.067.

[30] A. Abubakar and P. Kumam, "An improved three-term derivative-free method for solving nonlinear equations", *Computational and Applied Mathematics*, vol. 37, Sep. 2018. DOI: 10.1007/s40314-018-0712-5.

[31] J. E. Dennis Jr. and J. J. Moré, "Quasi-newton methods, motivation and theory", *SIAM Review*, vol. 19, no. 1, pp. 46–89, 1977. DOI: 10.1137/1019005.

[32] H. Badem, A. Basturk, A. Caliskan, and M. E. Yuksel, "A new efficient training strategy for deep neural networks by hybridization of artificial bee colony and limited–memory bfgs optimization algorithms", *Neurocomputing*, vol. 266, pp. 506–526, 2017, ISSN: 0925-2312.

[33] X. Liu, S. Liu, J. Sha, J. Yu, Z. Xu, X. Chen, and H. Meng, "Limited-memory bfgs optimization of recurrent neural network language models for speech recognition", in *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2018, pp. 6114–6118.

[34] L. Bottou, F. E. Curtis, and J. Nocedal, "Optimization methods for large-scale machine learning", *SIAM Review*, vol. 60, no. 2, pp. 223–311, 2018. DOI: 10.1137/16M1080173.

[35] H. Mohammad and S. Santos, "A structured diagonal hessian approximation method with evaluation complexity analysis for nonlinear least squares", *Computational & Applied Mathematics*, Aug. 2018. DOI: 10.1007/s40314-018-0696-1.

[36] J. J. Moré, "The levenberg-marquardt algorithm: Implementation and theory", English, in *Numerical Analysis*, ser. Lecture Notes in Mathematics, G. Watson, Ed., vol. 630, Springer Berlin Heidelberg, 1978, pp. 105–116, ISBN: 978-3-540-08538-6. DOI: 10.1007/BFb0067700. [Online]. Available: http://dx.doi.org/10.1007/BFb0067700.

[37] M. I. Ibrahimy, R. Ahsan, and O. O. Khalifa, "Design and optimization of levenberg-marquardt based neural network classifier for emg signals to identify hand motions", *Measurement Science Review*, vol. 13, no. 3, 2013.

[38] J. S. Smith, B. Wu, and B. M. Wilamowski, "Neural network training with levenberg–marquardt and adaptable weight compression", *IEEE Transactions on Neural Networks and Learning Systems*, vol. 30, no. 2, pp. 580–587, 2019.

[39] J. Bilski, B. Kowalczyk, and K. Grzanek, "The parallel modification to the levenberg-marquardt algorithm", in *Artificial Intelligence and Soft Computing*, Cham: Springer International Publishing, 2018, pp. 15–24, ISBN: 978-3-319-91253-0.

[40] L. Chen, "A modified levenberg–marquardt method with line search for nonlinear equations", *Computational Optimization and Applications*, vol. 65, May 2016. DOI: 10.1007/s10589-016-9852-y.

[41] L. Chen, "A high-order modified levenberg–marquardt method for systems of nonlinear equations with fourth-order convergence", *Applied Mathematics and Computation*, vol. 285, pp. 79–93, 2016, ISSN: 0096-3003. DOI: https://doi.org/10.1016/j.amc.2016.03.031.

[42] B. Huang and C. Ma, "A shamanskii-like self-adaptive levenberg–marquardt method for nonlinear equations", *Computers & Mathematics with Applications*, vol. 77, no. 2, pp. 357–373, 2019, ISSN: 0898-1221. DOI: https://doi.org/10.1016/j.camwa.2018.09.039. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0898122118305510.

[43] J. Fan and J. Pan, "A note on the levenberg–marquardt parameter", *Applied Mathematics and Computation*, vol. 207, no. 2, pp. 351–359, 2009, ISSN: 0096-3003. DOI: https://doi.org/10.1016/j.amc.2008.10.056. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0096300308008199.

[44] M. Cui, Y. Zhao, B. Xu, and X.-w. Gao, "A new approach for determining damping factors in levenberg-marquardt algorithm for solving an inverse heat conduction problem", *International Journal of Heat and Mass Transfer*, vol. 107, pp. 747–754, 2017, ISSN: 0017-9310. DOI: https://doi.org/10.1016/j.ijheatmasstransfer.2016.11.101. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0017931016331568.

[45] R. Fletcher, "Modified marquardt subroutine for non-linear least squares.", Atomic Energy Research Establishment, Tech. Rep., Jan. 1971.

[46] A. Sanhueza and C. 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)*, 2017, pp. 1–9.

[47] A. Griewank and A. Walther, *Evaluating Derivatives*, Second. Society for Industrial and Applied Mathematics, 2008.

[48] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: A survey", *J. Mach. Learn. Res.*, vol. 18, no. 1, pp. 5595–5637, Jan. 2017, ISSN: 1532-4435.

[49] W. Xu, N. Zheng, and K. Hayami, "Jacobian-free implicit inner-iteration preconditioner for nonlinear least squares problems", *Journal of Scientific Computing*, vol. 68, Feb. 2016. DOI: 10.1007/s10915-016-0167-z.

[50] N. Safiran, J. Lotz, and U. Naumann, "Algorithmic differentiation of numerical methods: Second-order adjoint solvers for parameterized systems of nonlinear equations", *Procedia Computer Science*, vol. 80, pp. 2231–2235, Dec. 2016. DOI: 10.1016/j.procs.2016.05.388.

[51] X. Fu, S. Li, M. Fairbank, D. C. Wunsch, and E. Alonso, "Training recurrent neural networks with the levenberg–marquardt algorithm for optimal control of a grid-connected converter", *IEEE Transactions on Neural Networks and Learning Systems*, vol. 26, no. 9, pp. 1900–1912, 2015.

[52] M. Förster and U. Naumann, "Solving a least-squares problem with algorithmic differentiation and openmp", in *Proceedings of the 19th International Conference on Parallel Processing*, ser. Euro-Par'13, Aachen, Germany: Springer-Verlag, 2013, pp. 763–774, ISBN: 9783642400469.

[53] E. Carson and N. J. Higham, "Accelerating the solution of linear systems by iterative refinement in three precisions", *SIAM Journal on Scientific Computing*, vol. 40, no. 2, A817–A847, 2018. DOI: 10.1137/17M1140819. eprint: https://doi.org/10.1137/17M1140819. [Online]. Available: https://doi.org/10.1137/17M1140819.

[54] E. Bergou, S. Gratton, and L. N. Vicente, "Levenberg–marquardt methods based on probabilistic gradient models and inexact subproblem solution, with application to data assimilation", *SIAM/ASA Journal on Uncertainty Quantification*, vol. 4, no. 1, pp. 924–951, 2016. DOI: 10.1137/140974687. eprint: https://doi.org/10.1137/140974687. [Online]. Available: https://doi.org/10.1137/140974687.

[55] S. Bellavia, S. Gratton, and E. Riccietti, "A levenberg–marquardt method for large nonlinear least-squares problems with dynamic accuracy in functions and gradients", *Numerische Mathematik*, Jun. 2018. DOI: 10.1007/s00211-018-0977-z.

[56] D. Achlioptas and F. Mcsherry, "Fast computation of low-rank matrix approximations", *J. ACM*, vol. 54, no. 2, 9–es, Apr. 2007, ISSN: 0004-5411.

[57] K. Sayood, "9 - scalar quantization", in *Introduction to Data Compression*, ser. The Morgan Kaufmann Series in Multimedia Information and Systems, K. Sayood, Ed., Burlington: Morgan Kaufmann, 2006, pp. 227–271, ISBN: 978-0-12-620862-7. DOI: https://doi.org/10.1016/B978-012620862-7/50009-2. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780126208627500092.

[58] R. M. Gray and D. L. Neuhoff, "Quantization", *IEEE Transactions on Information Theory*, vol. 44, no. 6, pp. 2325–2383, Oct. 1998, ISSN: 1557-9654. DOI: 10.1109/18.720541.

[59] E. Fiesler, A. Choudry, and H. J. Caulfield, "Weight discretization paradigm for optical neural networks", in *Optical Interconnections and Networks*, H. Bartelt, Ed., International Society for Optics and Photonics, vol. 1281, SPIE, 1990, pp. 164–173. DOI: 10.1117/12.20700.

[60] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations", in *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds., Curran Associates, Inc., 2015, pp. 3123–3131.

[61] P. Wang and J. Cheng, "Fixed-point factorized networks", in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 3966–3974.

[62] J. Wu, C. Yu, S. Fu, C. Liu, S. Chien, and Y. Tsao, "Increasing compactness of deep learning based speech enhancement models with parameter pruning and quantization techniques", *IEEE Signal Processing Letters*, vol. 26, no. 12, pp. 1887–1891, 2019.

[63] Y. Yang, L. Deng, S. Wu, T. Yan, Y. Xie, and G. Li, "Training high-performance and large-scale deep neural networks with full 8-bit integers", *Neural Networks*, vol. 125, pp. 70–82, 2020, ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2019.12.027.

[64] C. C. Paige and M. A. Saunders, "Lsqr: An algorithm for sparse linear equations and sparse least squares", *ACM Trans. Math. Softw.*, vol. 8, no. 1, pp. 43–71, Mar. 1982, ISSN: 0098-3500. DOI: 10.1145/355984.355989. [Online]. Available: https://doi.org/10.1145/355984.355989.

[65] G. Golub and W. Kahan, "Calculating the singular values and pseudo-inverse of a matrix", *Journal of the Society for Industrial and Applied Mathematics Series B Numerical Analysis*, vol. 2, no. 2, pp. 205–224, 1965. DOI: 10.1137/0702016. eprint: https://doi.org/10.1137/0702016. [Online]. Available: https://doi.org/10.1137/0702016.

[66] C. C. Paige and M. A. Saunders, "Algorithm 583: Lsqr: Sparse linear equations and least squares problems", *ACM Trans. Math. Softw.*, vol. 8, no. 2, pp. 195–209, Jun. 1982, ISSN: 0098-3500. DOI: 10.1145/355993.356000.

[67] D. C.-L. Fong and M. Saunders, "Lsmr: An iterative algorithm for sparse least-squares problems", *SIAM Journal on Scientific Computing*, vol. 33, no. 5, pp. 2950–2971, 2011. DOI: 10.1137/10079687X. eprint: https://doi.org/10.1137/10079687X. [Online]. Available: https://doi.org/10.1137/10079687X.

[68] S. J. Wright and J. N. Holt, "An inexact levenberg-marquardt method for large sparse nonlinear least squares", *The Journal of the Australian Mathematical Society. Series B. Applied Mathematics*, vol. 26, no. 4, pp. 387–403, 1985. DOI: 10.1017/S0334270000004604.

[69] P. L. Toint, "On large scale nonlinear least squares calculations", *SIAM Journal on Scientific and Statistical Computing*, vol. 8, no. 3, pp. 416–435, 1987. DOI: 10.1137/0908042.

[70] C. C. Paige and M. A. Saunders, "Algorithm 583: Lsqr: Sparse linear equations and least squares problems", *ACM Trans. Math. Softw.*, vol. 8, no. 2, pp. 195–209, Jun. 1982, ISSN: 0098-3500. DOI: 10.1145/355993.356000.