universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

Analysis of Several Derivative-Free Methods for Local Optimization

verfasst von / submitted by

Stefan Scheer, BSc

angestrebter akademischer Grad / in partial fulfillment of the requirements for the degree of

Master of Science (MSc)

Wien, 2021 / Vienna 2021

Studienkennzahl lt. Studienblatt / degree program code as it appears on the student record sheet:	A 066 821
Studienrichtung lt. Studienblatt / degree program as it appears on the student record sheet:	Masterstudium Mathematik
Betreut von / Supervisor:	ao. Univ.-Prof. Dipl.-Ing. Dr. Hermann Schichl

This page intentionally left blank.

Dedicated to my daughter.

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my supervisor Professor Hermann Schichl. His superb introductory lecture to Higher Mathematics was the main reason why I chose this career path. I want to thank him for all the inspiration, insight and guidance he has given me during the course of my studies.

My biggest thanks go to my family Theresa and Paulina, whose love seems endless. Being with them fills me with joy and warmth. Theresa, thank you for always supporting me, for listening to all my troubles and for standing by me in hard times.

Next on the list to thank are my older brothers, as they can always be counted on. I also appreciate my fellow students, especially Arved Bartuska, with whom I discussed almost every topic covered on our master's program. There was always a fun and positive atmosphere in my basketball team, which has helped me a lot along the way.

Last but not least, I want to thank Doris and Johannes for regularly taking care of my child, my parents, and Herta and Friedrich for providing me a room to write during the pandemic.

Abstract

In this thesis, we examine a total of five algorithms for unconstrained local optimization. Problems where the objective function is in the nonlinear least squares sense are very common in applications such as data fitting and regression analysis. The presented Gauss-Newton and Levenberg-Marquardt algorithms are among the most popular solvers for these problems. They benefit from the special structure of the Hessian matrix of such objective functions, which allows them to omit the costly computation of second-order derivatives. However, it can still be very challenging or numerically expensive to obtain the required Jacobian matrix in each iterative step of the respective procedure. Therefore, we analyze three derivative-free methods which may be used instead. The finite difference analogues of the Gauss-Newton and Levenberg-Marquardt algorithms by Brown and Dennis use forward differences to approximate the Jacobians. We prove local convergence of the methods and show that quadratic convergence can be guaranteed for functions that are zero at the minimum. The DUD algorithm by Ralston and Jennrich is a secant method that makes efficient use of function evaluations. As with the other depicted procedures, a linear least square problem has to be solved in each iteration. But instead of invoking a method for normal equations, DUD employs a stepwise regression procedure that is based on the Gauss-Jordan algorithm. DUD's optional line search does not increase the algorithmic reliability, but can be activated to enable alternative solution paths.
The results of computer experiments show how the mentioned methods differ in terms of efficiency and reliability. We conclude that our derivative-free optimization algorithms are mostly superior to the two established gradient methods when it comes to solving low-dimensional problems with medium relative accuracy.

Zusammenfassung

In dieser Arbeit untersuchen wir insgesamt fünf Algorithmen zur unbeschränkten lokalen Optimierung. Probleme, bei denen die Zielfunktion das Normquadrat einer vektorwertigen Funktion ist, kommen sehr häufig in Anwendungen wie zum Beispiel der Datenanpassung und der Regressionsanalyse vor. Wir beschreiben zwei der populärsten Löser für diese Probleme, das Gauß-Newton-Verfahren und den Levenberg-Marquardt-Algorithmus. Diese Methoden profitieren von der speziellen Struktur der Hesse-Matrix solcher Zielfunktionen, die es ihnen erlaubt die aufwendige Berechnung von Ableitungen zweiter Ordnung zu unterlassen. Dennoch kann es sehr herausfordernd oder numerisch teuer sein, die erforderliche Jacobi-Matrix in jedem iterativen Schritt des jeweiligen Verfahrens zu erhalten. Daher analysieren wir drei ableitungsfreie Methoden, die stattdessen verwendet werden können. Die Finite-Differenzen-Analoga des Gauß-Newton-Verfahrens und des Levenberg-Marquardt-Algorithmus von Brown und Dennis verwenden Vorwärts-Differenzenquotienten als Approximation der Jacobi-Matrizen. Wir beweisen die lokale Konvergenz der Methoden und zeigen, dass für Funktionen, deren Wert am Minimum Null ist, quadratische Konvergenz garantiert werden kann. Der DUD-Algorithmus von Ralston und Jennrich ist ein Sekantenverfahren, das Funktionsauswertungen effizient nutzt. Wie bei den anderen vorgestellten Algorithmen muss in jeder Iteration ein lineares Kleinste-Quadrate-Problem gelöst werden. Aber anstelle der Methode der Normalengleichungen verwendet DUD ein Verfahren zur schrittweisen Regression, welches auf dem Gauß-Jordan-Algorithmus basiert. Das optionale Liniensuchverfahren von DUD erhöht zwar nicht die algorithmische Zuverlässigkeit, kann aber aktiviert werden, um alternative Lösungspfade zu ermöglichen.
Ergebnisse von Computerexperimenten zeigen, wie sich die angeführten Methoden in Bezug auf Effizienz und Zuverlässigkeit unterscheiden. Wir kommen zum Schluss, dass unsere ableitungsfreien Optimierungsverfahren zur Lösung niedrigdimensionaler Probleme mit mittlerer relativer Genauigkeit meist besser geeignet sind als die beiden etablierten Gradientenmethoden.

Contents

List of Tables

List of Figures

This page intentionally left blank.

1 Introduction

The main goal of this thesis is to analyze, to implement and to test several derivative-free optimization algorithms for finding the zeros of systems of nonlinear equations. We will also discuss the basic concepts of optimization, give an overview of common techniques and investigate least square problems.

1.1 Outline

This section continues with a **motivation** to emphasize the importance of optimization in today's world and to provide some context for this work.
We will introduce **basic definitions**, classify **optimization problems** and distinguish different **local optimization techniques** in Section 2.
In Section 3, we demonstrate how crucial **nonlinear least squares** problems are for this thesis and how they are applied to **data fitting** and **regression**.
The main part consists of Sections 4 and 5, where several unconstrained optimization **algorithms** are thoroughly discussed and benchmarked on a **test set**. We present the **numerical results** in various forms and analyze the performance of our methods.
Finally, we conclude our findings in Section 6.
Required but well known mathematical concepts can be looked up in Appendix A.
The **MATLAB source code** of the programs and test functions, implemented in MATLAB version R2016a by the author Stefan Scheer, can be found in Appendix C.

1.2 Motivation

Optimization is a tool for determining the "best" solution from all feasible solutions to certain mathematically defined problems, which are often models that represent the physical reality. It is widely used in science, engineering, economics, commerce, and industry and its area of application is still growing. So basically, optimization techniques are utilized in almost every discipline where numerical information is processed. Some examples where optimization problems occur are:

- Profit maximization, e.g., designing a portfolio of investments to maximize the expected return while maintaining a certain level of risk
- Optimal production
- Logistics
- Structural design, e.g., buildings and bridges
- Aero-engine and aero-frame design, e.g., computing the optimal shape of an aircraft component
- Chemical reactor design
- Protein design

- Robotics, e.g., finding the optimal trajectory for a robot arm
- Machine scheduling problems
- Other branches of numerical mathematics, e.g., data fitting, nonlinear equations in ODE's and variational principles in PDE's

When we want to optimize a given system, we must first identify an *objective*, a measure of performance. The objective depends on certain characteristics of the system, called *unknowns* or *variables*. An *optimization problem* begins with a set of independent variables and often includes restrictions that define acceptable values of the unknowns, so called *constraints*. The goal is then to find *feasible* values of the variables that optimize the objective. The solution of an optimization problem needs not to be unique, nor is it generally guaranteed that it exists at all.
In mathematics, optimization is done by *minimizing* or *maximizing* an *objective function* subject to certain constraints on its variables. For example, companies may strive to minimize their expenses or to maximize production. If multiple objectives arise, they are often reformulated as a single function by forming a weighted combination of different goals or by treating some of them as constraints. In *unconstrained optimization*, the optimum is sought of an objective function of many variables, without any constraints. Additional complication of the different types of constraint functions arise in *constrained optimization*. For example, there could be budgetary constraints in economics or shape restrictions in a design problem. The process of identifying the objective function, the variables and the constraints for a given problem is known as *modeling*. *Optimization algorithms* can be used to find the solution of a formulated model. They are implemented as programs, i.e., the calculations are nowadays solely carried out on computers. There is no universal algorithm for optimization. Each particular type of problem demands an individual approach. The choice of a proper algorithm is crucial, as it determines whether a solution can be found, and if so, whether the problem is solved rapidly or slowly.
It is not always clear if an optimization algorithm succeeded in its task of finding a solution. In many subfields of optimization there are *optimality conditions*, which can be applied to check if the current set of variables is indeed a solution of the problem. This is generally not the case for *derivative-free optimization*, where derivative information is not available, ignored or just approximated. Here, optimality guarantees can usually not be given. However, for some *derivative-free methods* it is possible to prove that optima can be found.
When we are trying to find the right optimization technique, it is important to know the structure of a system. It is advantageous to determine special characteristics of some given problem so that it can be solved more efficiently by an optimization algorithm. For example, it may be possible to omit tests and computations for situations that do not occur or to exploit specific properties of a function. That is, an adequate solution method may only be found when the properties of the model are properly classified. Not every distinction made impacts a proposed program significantly, and no set of categories is ideal for every circumstance. But an obvious and reasonable good choice for classification according to algorithmic efficiency are the properties of the objective and constraint functions.

The following scheme has been proposed in [14]. It categorizes an optimization model based on the characteristics of the problem's functions such that significant algorithmic advantage can be gained.

Properties of the objective function:

- Univariate function
- Linear function
- Sum of squares of linear functions
- Quadratic function
- Sum of squares of nonlinear functions
- Smooth nonlinear function
- Sparse nonlinear function
- Nonsmooth nonlinear function
- Convex function

Properties of the constraint functions:

- No constraints
- Simple bounds
- Linear functions
- Sparse linear functions
- Smooth nonlinear functions
- Sparse nonlinear functions
- Nonsmooth nonlinear functions

Other features of an optimization problem are also to be taken into consideration. The size of a model affects both the program's storage and the amount of computational effort required to obtain a solution. Also, the computable information available to an algorithm is of particular interest. For instance, the derivatives of a function may be provided by the user or are easily obtainable by analytic calculation. However, most of the time derivative information is unavailable or impractical to obtain and only function evaluations are accessible.
A real-world problem is often modeled as a system of nonlinear equations, where each equation represents a different real-life situation affecting the objective. Such a nonlinear system can be stated in terms of a vector-valued function whose zeros are the solutions (if they exist). The resulting *root finding problem* may then be reformulated as an optimization problem where the sum of squares of nonlinear functions is to be minimized.

The topic of this work is unconstrained local optimization. With respect to the demonstrated classification scheme, our algorithms are specifically tailored for optimization problems with no constraints and an objective function which is the sum of squares of nonlinear functions. The involved functions are assumed to be differentiable but the presented algorithms forgo computing exact derivatives. There exists a large class of important smooth problems where it is expensive and difficult to access derivative information. If the gradient of such a differentiable function is not given, a gradient-based program would have to compute the derivatives numerically, which is very costly and error-prone. In such situations, and especially when the objective function is noisy and inaccurate, gradient-based methods are often outperformed by derivative-free algorithms. This motivation is partly based on the books [11, 14, 40].

1.3 Notation and Conventions

Throughout this thesis, all vectors x are considered as column vectors. Correspondingly, transposed vectors x^T are treated as row vectors. For vectors $x, y \in \mathbb{R}^n$ we denote by $x^\mathsf{T} y$ the Euclidean inner product, i.e.,

$$x^\mathsf{T} y = \sum_{i=1}^{n} x_i y_i,$$

where $x_i, y_i \in \mathbb{R}$ are the i-th entries of the vectors x and y, respectively. Inequalities between vectors are always interpreted component-wise, e.g., $x \geq y$ if and only if $x_i \geq y_i$ for all i. Depending on the context, elements of a matrix $A = (a_{ij})$ in row i and column j are denoted by a_{ij} or A_{ij}. The latter notation may also be used for submatrices; in particular, $A_{i:}$ denotes the i-th row and $A_{:j}$ the j-th column of A. We write I_n for the $n \times n$ identity matrix or just I if the dimension is secondarily. The size of an employed zero matrix 0 should be easily traceable.
The gradient of a continuously differentiable function $f : \mathbb{R}^n \to \mathbb{R}$ at $x \in \mathbb{R}^n$ is always written as $g(x) := \nabla f(x)$. If f is twice continuously differentiable, we denote the Hessian at x by the symmetric matrix $G(x) := \nabla^2 f(x)$. That is,

$$g_i = \frac{\partial f}{\partial x_i} \quad \text{and} \quad G_{i,j} = \frac{\partial^2 f}{\partial x_i \partial x_j}.$$

Function arguments are omitted in cases where variables are arbitrary or where it is clear which variable is being used.
Differences are indicated by Δ and are defined situational. The Laplace operator is not needed in this thesis, so there is no overlap in notation with ∇^2 and Δ.
In order to distinguish a sequence of vectors from vector components, we write the iteration index l as superscript, i.e., $x^l \in \mathbb{R}^n$ as opposed to $x_i \in \mathbb{R}$. For scalars and matrices it is instead a subscript to avoid confusion with powers. Thus, to improve the readability of formulas with iteration indices, we shall simply write

$$f_l := f(x^l), \quad g^l := g(x^l) \quad \text{and} \quad G_l := G(x^l).$$

The Landau symbol $\mathcal{O}$ is primarily used for work and storage counts. It describes an unspecified number $a_n = \mathcal{O}(w_n)$ with the property

$$\big|a_n\big| < Mw_n \quad \text{for some constant } M > 0 \text{ as } n \to \infty.$$

The big-O notation can also stand for an unspecific remainder term $r(s) = \mathcal{O}\big(\phi(s)\big)$ with the property

$$r(0) = 0 \text{ if } \phi(0) = 0 \quad \text{and} \quad \big|r(s)\big| < M\phi(s) \quad \text{for some constant } M > 0 \text{ as } s \to \infty.$$

The little-o notation is used for an unspecific remainder term $r(s) = o\big(\phi(s)\big)$ with the property

$$r(0) = 0 \text{ if } \phi(0) = 0 \quad \text{and} \quad \frac{r(s)}{\phi(s)} \to 0 \text{ as } s \to \infty.$$

This page intentionally left blank.

2 Mathematical Formulation

Optimization is the minimization or maximization of a function subject to constraints on its variables. But this definition is very vague. If we want to mathematically solve a given problem, we need to be more precise and we must have proper knowledge about the basic concepts. We want to be able to classify a model and to find a fitting and efficient solution method for it. Therefore, it is important to provide basic definitions and to discuss the most prominent topics covered in optimization. This section is inspired by the works [14, 38, 40]. There are many applications for different optimization techniques. Some of the concepts and methods are discussed in detail, others are skipped for the sake of readability or simply because they are not relevant for this thesis. But the reader is invited to look up further information in the recommended references.

2.1 Basic Definitions

Definition 2.1. An *optimization problem* or *mathematical program* has the form

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & c_i(x) = 0,\ i \in \mathcal{I}, \\ & c_i(x) \geq 0,\ i \in \mathcal{J}, \end{aligned} \tag{2.1}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is referred to as the *objective function* and $x \in \mathbb{R}^n$ is the vector of *variables*, also denoted as *unknowns* or *parameters*. The functions $c_i : \mathbb{R}^n \to \mathbb{R}$ are called *constraints*. In particular, they are named *equality constraints* or *inequality constraints* if their indices correspond to the index set $\mathcal{I}$ or $\mathcal{J}$, respectively.

Definition 2.2. Any point $\hat{x} \in \mathbb{R}^n$ that satisfies all the constraints in (2.1) is said to be *feasible*. The set of all feasible points is termed *feasible domain* or *feasible region*. An optimization problem for which no feasible points exist is called *inconsistent*.

Definition 2.3. If $\mathcal{I} = \mathcal{J} = \emptyset$ in (2.1), we call the optimization problem *unconstrained*, otherwise *constrained*. It is *bound constrained* if all constraints have the form of simple bounds $x_i \geq l_i$ or $x_i \leq u_i$ on the variables. If all constraints are linear, it is referred to as *linearly constrained*, otherwise *nonlinearly constrained*.

Remark. For unconstrained optimization problems, every point $x \in \mathbb{R}^n$ is feasible.

Definition 2.4. Let $\mathcal{C} \subseteq \mathbb{R}^n$ be the *feasible domain* of the optimization problem (2.1). A point $x^* \in \mathbb{R}^n$ is called a *local solution* of (2.1) or a *local minimizer* of f in $\mathcal{C}$ if

$$f(x^*) \leq f(x) \tag{2.2}$$

for all $x \in \mathcal{C}$ in some neighborhood of x^*. It is a *global solution* of (2.1) or a *global minimizer* of f in $\mathcal{C}$ if (2.2) holds for all $x \in \mathcal{C}$.

Remark. If the strict inequality in Definition 2.4 holds, a local (global) solution is referred to as *strict* local (global) minimizer of f, respectively.

In terms of money it is almost always optimal to maximize profit. So it often makes more sense of thinking of an optimization problem as a *maximization problem*. Definition 2.1 can be trivially reformulated as

$$\begin{aligned} \underset{x}{\text{maximize}} \quad & -f(x) \\ \text{subject to} \quad & c_i(x) = 0,\ i \in \mathcal{I}, \\ & c_i(x) \geq 0,\ i \in \mathcal{J}. \end{aligned} \tag{2.3}$$

That is, minimizing $-f$ is the same as maximizing f.

Remark. A local (global) solution of a maximization problem is defined analogously to Definition 2.4, only with a reversed inequality sign.

Definition 2.5. The function value at a local (global) minimizer or maximizer is called a local (global) *minimum* or *maximum*, respectively.

Remark. From now on we omit maximization problems in this work, since, according to (2.3), they can be stated in terms of minimization anyway.

Definition 2.6. An optimization problem is called *smooth* if the objective function and all constraints are continuously differentiable, otherwise it is said to be *nonsmooth*.

2.2 Classification of Optimization Problems

The majority of mathematical programs can be expressed in the form (2.1). Although this representation is used universally, it is important to determine special characteristics that allow a problem to be solved in an efficient way. Therefore, it is important to distinguish between different optimization problems. A classification scheme with respect to the problem functions is already listed in Section 1.2. But the following basic concepts in optimization provide a better overview of the main distinctions in the nature of problems. These notions are so significant in optimization that each of them can be viewed as own discipline.

Linear and nonlinear optimization

A constrained optimization problem where the objective function and all constraints are linear is called a *linear optimization problem* or *linear program* (LP). That is, (2.1) is specified as

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & c^{\mathsf{T}}x \\ \text{subject to} \quad & Ax = b, \\ & x \geq 0, \end{aligned} \tag{2.4}$$

where $c \in \mathbb{R}^n,\ x \in \mathbb{R}^n,\ b \in \mathbb{R}^m$ and $A \in \mathbb{R}^{m \times n}$.

If an optimization problem is not linear, it is termed *nonlinear*. A *nonlinear program* (NLP) can take on various forms. For example, a *quadratic program* (QP) can be written as follows:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & c^\mathsf{T} x + \frac{1}{2} x^\mathsf{T} Q x \\ \text{subject to} \quad & Ax \geq b, \end{aligned} \tag{2.5}$$

where $c \in \mathbb{R}^n,\ x \in \mathbb{R}^n,\ b \in \mathbb{R}^m,\ A \in \mathbb{R}^{m \times n}$ and $Q \in \mathbb{R}^{n \times n}$ is symmetric. If the variables in (2.5) are quadratically constrained, the problem is known as *quadratically constrained quadratic program* (QCQP). Another type in nonlinear programming are *polynomial programs* (PP), where the objective function and the constraints are polynomials.

Global and local optimization

In *global optimization* a global solution is to be found. Global minimizers are often difficult to recognize and it may be even harder to locate them. General nonlinear problems may possess local solutions that are not global solutions. In most cases it is easier to find such local minimizers, which is the subject of *local optimization*. Strictly speaking, an optimization problem is solved only when a global solution is found, but in practice, we often have to be satisfied with finding local solutions.
It should be noted that for some problems it is sufficient to find a feasible point only. They are called *constraint satisfaction problems* (CSP) and correspond to a constant objective function.

Constrained and unconstrained optimization

It is important to distinguish between problems that have constraints on the variables and those that do not, see Definition 2.3. *Constrained optimization* deals with problems that arise from systems in which restrictions play an essential role. These constraints may be simple bounds such as $l \leq x \leq u$ on the unknown x, or more general, linear or nonlinear constraints. Problems with complex restrictions are usually hard to solve. In *unconstrained optimization* all possible values of the variables are accepted. Many problems do not have restrictions on the unknowns and in some cases it might be safe to neglect natural constraints which do not affect the solution of a system. Unconstrained problems are generally considered easier to solve which is why they often arise as reformulations of constrained optimization problems. Such a transformed problem then has a penalizing term added to its objective function in order to counteract the constraint violation.
Optimality conditions exist for both constrained and unconstrained optimization. For instance, under certain regularity and convexity assumptions, the *Karush-John conditions* provide first-order derivative tests to check if a solution of a constrained problem is a candidate for an optimal solution. For more information on this matter, see [38, 45]. But in this thesis we are particularly interested in the optimality conditions for unconstrained optimization. They are stated as Theorems 2.8, 2.9, and 2.10 in Section 2.

Bound constrained optimization

An important special case in constrained optimization are *bound constrained problems*, see Definition 2.3. They find application in systems where the variables are quantities which are restricted to a given range. Furthermore, bound constrained optimization techniques are often involved in the development of algorithms for more general constraint problems.

Remark. We can convert a mathematical program (2.1) with inequality constraints to a problem with equality and bound constraints by introducing *slack variables* s_i and replacing the inequalities $c_i(x) \geq 0,\ i \in \mathcal{J}$, by

$$c_i(x) - s_i = 0, \quad s_i \geq 0, \quad \text{for all } i \in \mathcal{J}.$$

Bound constraints $l \leq x \leq u$ need not to be converted since there exist good techniques on how to handle them. Hence, we can reformulate (2.1) as

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & c_i(x) = 0, \quad i \in \mathcal{I} \cup \mathcal{J}, \\ & l \leq x \leq u, \quad l, u \in \mathbb{R}^n . \end{aligned} \tag{2.6}$$

Convex optimization

Many practical problems are modeled in terms of convex functions and convex sets. Because of its handy nature, convexity is exploited thoroughly by many algorithms. A *convex program* (CVXP) is a special case of a general constrained optimization problem (2.1) in which the objective function is convex, the equality constraints are linear and the inequality constraints are convex, that is $c_i(x) \leq 0$ for convex functions $c_i,\ i \in \mathcal{J}$. For convex programs, any local solution is a global solution and there exists at most one optimum if the objective function is strictly convex. Linear programs are always convex. The basics of convex optimization can be looked up in [45].

Continuous and discrete optimization

Sometimes the variables make sense only if they take integer values, e.g., when they describe a number of entities. In such cases, the mathematical formulation (2.1) includes *integer constraints*, which have the form $x_i \in \mathbb{Z}$, or *binary constraints*, which have the form $x_i \in \{0, 1\}$. Problems of this type are called *integer programs* (IP). They are a type of *discrete optimization*, where only finite discrete sets of values are considered. *Combinatorial optimization problems*, which often deal with permutations and graph structures, also fall into this category. If some of the variables are not restricted to be integer or binary, we call a problem *mixed integer program* (MIP). In particular, *mixed integer nonlinear programs* (MINLP) are proven to be powerful tools for modeling. They address nonlinear problems with continuous and integer variables. In *continuous optimization* the variables are required to be continuous, that is they take any values in $\mathbb{R}^n$ permitted by the constraints.

Stochastic and deterministic optimization

Sometimes, a problem cannot be fully specified because it depends on quantities that are unknown at the time of formulation. Such uncertainties arise in many types of systems and are subject of *stochastic optimization*. Instead of simply using a "best" guess for an unknown quantity, algorithms incorporate estimates of probabilities and try to produce solutions that optimize the expected performance of a model. Related disciplines, which also deal with uncertainties, are *robust* and *chance-constrained optimization*. Stochastic algorithms are not part of this work but it should be noted that they are often solved by various deterministic subproblems. We focus on *deterministic optimization*, where the model is completely known.

Smooth, nonsmooth and derivative-free optimization

Smooth optimization deals with problems stated with functions whose derivatives exist and are continuous, cf. Definition 2.6. That is, if a function is evaluated at one point, we can infer information about the function at neighboring points. For example, descent directions for many line search techniques are gradient based. Or if a function is twice differentiable, methods can extract curvature information from the Hessian. Such practices are generally not possible in *nonsmooth optimization*, where functions are not continuously differentiable or even discontinuous. That is, the behavior of a function is not predictable near a point of nonsmoothness. Furthermore, it is often impossible to identify a minimizer. Therefore, solving nonsmooth optimization problems is considered much harder than solving smooth ones. For example, in the nonconvex case, finding a descent direction is not easily possible. There are however reasonable good and established solution methods for nondifferentiable problems such as subgradient and bundle methods, see [1].
Nonsmooth optimization is not to be confused with *derivative-free optimization*, which includes problems that are to be solved by methods that do not access derivative information. *Derivative-free algorithms* are often heuristic methods which rely heavily on function evaluations; or they aim to mimic the gradient, or in some cases even the Hessian of a function by finite difference approximation. Programs of this type are subject of this thesis. They find application in both nonsmooth and smooth problems. The reasons for choosing such algorithms over a gradient based method are situational and are described in [38] as follows:

- Function evaluation is so cheap that there is no point in spending effort on working out derivatives.
- Function evaluation involves lengthy intermediate calculations or a significant number of branches, so that automatic differentiation[1] is either very storage intensive or inaccurate.
- Function values are obtained from experiments and not from computer programs. In this case there is usually no way to get derivatives except approximately by comparing function values.

[1] Automatic differentiation is a chain rule-based technique for evaluating derivatives, see [18].

2.3 Classification of Optimization Methods

In order to efficiently find solutions of a particular problem, different methods have to be considered. In practice, optimization algorithms proceed iteratively by generating a sequence of vectors which are designed to converge to a sought optimum. Basically, there are two techniques for obtaining the next iteration point, a line search and a trust region approach. They are explained in Section 2.4. In general, the terms *direct, indirect* or *stochastic* are commonly used to classify optimization methods. The following information is extracted from [38, 39].

Direct methods

Direct (search) methods are deterministic but make no model assumption. Hence, they have no access to derivative information at all. Such algorithms sample the objective function at a finite number of points at each iteration and decide which actions to take next solely based on the function evaluations and the corresponding pattern.
Examples are:

- Nonlinear simplex method
- Pattern search
- Generating set search

Indirect methods

Indirect (search) methods are deterministic and model-based. Usually, search directions can be determined and often mathematically rigorous convergence analysis can be performed. Nevertheless, heuristic-based methods are also common.
Examples are:

- Methods using quadratic fits
- Trust region methods based on model updates
- Response surface methods
- Surrogate function methods
- Lipschitz constant based methods
- Implicit filtering
- Multilevel coordinate search

Stochastic methods

Stochastic methods use random choices in their strategy.
Examples are:

- Simulated annealing
- Genetic algorithms
- Stochastic clustering algorithms
- Particle swarm methods
- Ant colony optimization
- Hit-and-run algorithms
- Tabu search

Remark. Most of the above examples mainly have motivational character and we will not further investigate them in this work. But some essential optimization techniques are explained in more detail in the upcoming section. Also, we note that there exist various hybrid methods, e.g., direct stochastic approaches. However, all of the algorithms treated in Section 4 are considered indirect methods.

Optimization methods can either be *feasible-point methods*, where all iterates are feasible, or *infeasible-point methods*, where feasibility is achieved only in the limit.
It is important to account for the amount of storage needed by an algorithm and for the amount of derivative information needed for computing. Therefore, a distinction can be made between *low storage*, *medium storage* and *high storage methods* depending on the number of memory locations. If n is the maximum of the number of variables and the number of constraints appearing in a problem, $\mathcal{O}(n)$ storage locations are considered low, whereas $\mathcal{O}(n^2)$ locations mark medium storage and $\mathcal{O}(n^3)$ or more locations are quantified as high storage. *Derivative-free, first-order* and *second-order methods* use function values only, function values and gradients, and function values, gradients and Hessians, respectively. The terms *no-derivative, first-derivative* and *second-derivative* are also commonly used for the respective methods.
As already mentioned in Section 2.2, derivative-free algorithms usually also assume differentiability and work well under certain conditions. But in general gradient-based algorithms are more reliable and perform faster. Also, the computation of the Hessian is mostly not worth the high programming effort since it increases the performance of an algorithm only slightly. Thus, the most frequently used algorithms in practice are first-order methods.
Let us recall the function-based classification scheme from Section 1.2. Algorithms can perform significantly better if they fit the nature of the problem functions. We therefore distinguish between optimization methods according to their ability to exploit the structure of a system. For example, *methods for large, sparse problems* take advantage of the location of nonzeros in the Hessian; or *methods for nonlinear least squares problems* assume that the objective function is a sum of squares of nonlinear functions and there are no constraints or simple bounds only. *Methods for unstructured problems* assume no structural knowledge of the functions involved.

2.4 Local Optimization Techniques

The advantage of global optimization over local optimization is obvious. Instead of searching for a locally unimprovable feasible point, the globally best point in the feasible domain is sought. But global methods are the hardest part of nonlinear programming and are associated with major drawbacks, such as high computational cost, problem-specific parameter tuning and limited problem size. Furthermore, if no global information is available, e.g., if the objective of an optimization problem is a black-box function[2] where only function values are accessible, a globally best point might not be identifiable. Often, finding the global optima of a problem is desirable but not essential. In many practical applications, any sufficient good feasible point is useful and a possible improvement over what is available without optimization. This motivates the need for local methods in derivative-free optimization. In this section we introduce commonly used basic **local optimization techniques** by following the works [11, 14, 37, 38, 40]. Some of these routines are also part of many global methods. The most popular global optimization techniques can be looked after in [36]. This reference is an excellent survey of global optimization as it presents algorithms and theory in sufficient detail.
Most basic local optimization algorithms are gradient-based. In this section, we assume that the objective function $f : \mathbb{R}^n \to \mathbb{R}$ is twice continuously differentiable at all points of interest. The gradient of f at $x \in \mathbb{R}^n$ is then denoted by $g(x) := \nabla f(x)$ and $G(x) := \nabla^2 f(x)$ is the symmetric Hessian at x.
The strategies of local optimization algorithms largely depend on whether constraints are involved or not. Introduced methods are therefore split up into **constrained** and **unconstrained optimization techniques**. **Derivative-free methods** are often modified versions of such techniques. But we start this segment with two essential concepts in iterative methods for obtaining the next iteration point, the **line search** and the **trust region**.

2.4.1 Line search and trust region methods

The basic iterative approach for finding a minimum of the objective function f of a nonlinear program is to construct a *descent sequence* for f. This is a sequence of feasible points x^l satisfying

$$f(x^{l+1}) < f(x^l) \tag{2.7}$$

in each iteration. Ideally, $\lim_{l\to\infty} x^l$ should then be an optimum.
Line search methods choose the next iteration point along promising search directions or on suitable search paths. Trust region methods, also called *restricted-step methods*, work in a different manner. There, the next iteration point is chosen by minimizing a model function within a region in which it is believed to approximate the objective function. If the approximation within such a so called *trust region* is adequate, the region is expanded. Otherwise, if the approximation is poor, the region is contracted.

[2] A black-box function is a function $f : \mathbb{R}^n \to \mathbb{R}$ for which the analytic form is not known.

Line search

Although we expect the objective function to be smooth, there also exist line search algorithms in nonsmooth optimization. But performance is degraded for the latter. We usually construct descent sequences iteratively by choosing in every step a *search path* $x(\alpha),\ \alpha \geq 0$, and a *search direction* p^l. Most of the time a linear search path is chosen, i.e., for a feasible iteration point x^l,

$$\begin{aligned} x(\alpha) &:= x^l + \alpha p^l, \\ x(0) &= x^l, \\ x'(0) &= p^l. \end{aligned}$$

For the following we write

$$x := x^l, \quad \bar{x} := x^{l+1} \quad \text{and} \quad p := p^l.$$

The next step is to determine the *stepsize parameter* α such that

$$f(x + \alpha p) < f(x). \tag{2.8}$$

The choice of α is called the *line search*. Then

$$\bar{x} := x + \alpha p$$

and $s := \bar{x} - x$ is called the *step*. The norm $\|s\|$ is called the *step size* or *step length*.

Remark. The above procedure describes a *straight* line search. If the search path $x(\alpha)$ is piecewise linear, the line search is called *bent*, otherwise *curved*. Newton-like methods are a prime example for algorithms which use straight line searches. Bent line searches are a very useful tool in bound constraint optimization.

Most line search algorithms require p to be a *descent direction*, i.e., it must hold that

$$g(x)^\mathsf{T} p < 0. \tag{2.9}$$

The success of a line search method depends on effective choices of both the direction p and the stepsize parameter α. We face a trade-off when computing α in (2.8). Descent should be enough so that convergence[3] can be guaranteed. But at the same time, α should be big enough so that $x(\alpha)$ is significantly different from x. The *Goldstein quotient*, defined by

$$\mu(\alpha) := \frac{f\big(x(\alpha)\big) - f(x)}{\alpha g(x)^\mathsf{T} p} \quad \text{for } \alpha > 0,$$

is used to find regions of sufficient descent. The function μ can be continuously extended to $[0, \infty[$ by $\mu(0) := 1$. The most useful *sufficient descent condition* is then

$$\mu(\alpha)\big|\mu(\alpha) - 1\big| \geq \beta \quad \text{for fixed } \beta > 0. \tag{2.10}$$

It yields that $f\big(x(\alpha)\big)$ is sufficiently smaller than $f(x)$ since $\mu(\alpha)$ cannot be too small; and $\mu(\alpha)$ cannot be too close to 1, so α has to be sufficiently positive.

[3]Definitions for various kinds of convergence are stated in Appendix A.3.

Definition 2.7. We call a line search *efficient* if it always produces steps satisfying

$$\inf_{l\geq 0} \frac{(f_l - f_{l+1})\|s^l\|^2}{\big((g^l)^\mathsf{T} s^l\big)^2} > 0,$$

where $f_k := f(x^k)$, $s^k := x^{k+1} - x^k$ and $g^k := \nabla f(x^k)$ for an iteration index k.

Remark. An *exact line search* is a straight line search where α is chosen as the (global) minimizer of $f\big(x(\alpha)\big)$. Exact line searches are efficient.

Trust region

The basic idea of trust region methods is to accept the minimum of a model function only as long as the model reflects the behavior of the objective function f. Nonlinear constraints are usually approximated too. Mostly, the model is quadratic:
If we truncate the Taylor series expansion

$$f(x+s) = f(x) + g(x)^\mathsf{T} s + \frac{1}{2} s^\mathsf{T} G(x) s + o\big(\|s\|^2\big) \tag{2.11}$$

after the second order term, we see that $f(x+s)$ is locally well approximated by the quadratic function

$$q(x+s) = f(x) + g(x)^\mathsf{T} s + \frac{1}{2} s^\mathsf{T} G(x) s.$$

The exact Hessian G is often not accessible and replaced by a symmetric Hessian approximation B. The quadratic model has then the form

$$q(x+s) = f(x) + g(x)^\mathsf{T} s + \frac{1}{2} s^\mathsf{T} B(x) s. \tag{2.12}$$

The difference between $q(x+s)$ and $f(x+s)$ is $o\big(\|s\|^2\big)$, which is small when $\|s\|$ is small.
First, a region around the current iterate is defined in which the quadratic function q is trusted to be an adequate representation of f. Then, the approximate minimizer of q in the trust region is chosen as step. Technically, the direction and the length of the step are determined simultaneously. The decision as to whether the model is acceptable is based on the norm of the step. If it is not acceptable, the trust region radius is reduced and a new minimizer is computed. On the other hand, if the model is appropriate, a new iteration point is determined and the radius is enlarged.
So at each iteration point $x^l \in \mathbb{R}^n$ a constrained subproblem of the form

$$\begin{aligned} \underset{p\,\in\,\mathbb{R}^n}{\text{minimize}} \quad & q_l(p) := f_l + g^{l\mathsf{T}} p + \frac{1}{2} p^\mathsf{T} B_l p \\ \text{subject to} \quad & \|p\| \leq \delta_l \end{aligned} \tag{2.13}$$

is solved, where $\delta_l > 0$ is the current trust region radius, $B_l := B(x^l)$ and $\|\cdot\|$ is usually the Euclidean norm. That is, the minimizer p^l of (2.13) lies in the ball of radius δ_l. The next iteration point is then chosen as $x^{l+1} = x^l + p^l$, provided that the model is accepted.

A crucial part of every trust region algorithm is the strategy for choosing the radius δ_l at each iteration. The typical procedure is to base this choice on the ratio of the *actual reduction* in f to the *predicted reduction* in f by q. That is, given an iteration point x^l and the corresponding minimizer, i.e., the trial step p^l,

$$\rho_l := \frac{f(x^l) - f(x^l + p^l)}{q_l(0) - q_l(p^l)}.$$

The denominator is always nonnegative since p^l is obtained by minimizing over a region that contains $p = 0$, see (2.13). Hence, if ρ_l is negative, we find

$$f(x^l + p^l) > f(x^l)$$

which is not acceptable, see (2.7). Consequently, the step must be rejected. In this case, or if ρ_l is close to zero, we shrink the trust region by reducing δ_l at the next iteration. The trust region is not altered if ρ_l is positive but significantly smaller than 1. It gets expanded if ρ_l is close to 1 because this corresponds to a good agreement between f and q over the step.

Remark. Various realizations of line search and trust-region algorithms exist for both constrained and unconstrained optimization problems. A combination of the two techniques is also possible. For example, Nocedal and Yuan proposed such an algorithm for nonlinear programming in [41]. The Levenberg-Marquardt algorithm in Section 4.2 can be viewed as a trust region method.

2.4.2 Unconstrained optimization techniques

We will now give an overview of the most common local techniques in unconstrained optimization. Usually, we lack a global perspective on the objective function f. Often, all we know are some function values of f and derivatives at specific points. In such cases, especially when function evaluation is expensive, gradient-based algorithms are reliable and the most used local methods in practice. The underlying theory for such solvers are the following *optimality conditions for unconstrained optimization*, which are proven in [40, p. 14–15].

Theorem 2.8 (First order necessary condition)**.** *Suppose that f is continuously differentiable in an open neighborhood of $x^* \in \mathbb{R}^n$. If x^* is a local minimizer of f, then $g(x^*) = 0$.*

Theorem 2.9 (Second order necessary condition)**.** *Suppose that f is twice continuously differentiable in an open neighborhood of $x^* \in \mathbb{R}^n$. If x^* is a local minimizer of f, then $g(x^*) = 0$ and $G(x^*)$ is positive semidefinite.*

Theorem 2.10 (Second order sufficient condition)**.** *Suppose that f is twice continuously differentiable in an open neighborhood of x^*. If $g(x^*) = 0$ and $G(x^*)$ is positive definite, then x^* is a strict local minimizer of f.*

It is easily verified that condition (2.9) actually defines a direction p of descent. Since

$$f(x + \alpha p) = f(x) + \alpha g(x)^\mathsf{T} p + o(\alpha)$$

and $g(x)^\mathsf{T} p < 0$, we find that $f(x + \alpha p) < f(x)$ for sufficient small α and thus (2.7) is satisfied. The particular choice of the search direction p describes different methods.

Method of steepest descent

The method of steepest descent, also known as *gradient descent*, is a line search that moves along the direction in which f decreases most rapidly local to the iteration point x^l. That is, the *steepest descent direction* $p^l = -g^l$ is chosen in every step. The advantage of gradient descent is its low computational demand as it requires no calculation of second derivatives. Unfortunately, in practice, this method often exhibits oscillatory behavior termed *zigzagging*. Under assumptions, local convergence can be proven but comes with an arbitrarily slow rate of linear convergence, see [38].

Line search methods may use search directions other than the steepest descent direction. This feature is utilized by *Newton-like methods*, which generally perform better than gradient descent.

Newton-like methods

They are based on local quadratic models of the objective function f, which have the form (2.12). As previously mentioned, second derivatives in G are usually not computed but an approximation $B \approx G$. That is, a local minimizer of the quadratic function q is computed as an approximation of a local minimizer of f. For numerical reasons, q is needed to be strongly convex, i.e., B has to be positive definite. In this case, q has a unique minimizer which has the form $s^* = -B^{-1}g$, since (cf. (2.12))

$$\nabla_s q(x+s) = g + Bs = 0.$$

Hence, we suspect the minimizer of f to lie close to s^* or at least in direction s^*. The matrix B is invertible and its inverse is positive definite, see Theorem A.10 in Appendix A. Thus, Newton-like methods perform a line search along the *Newton direction*, which is defined as

$$p := -B^{-1}g. \tag{2.14}$$

This is a descent direction, see (2.9), since $g^\mathsf{T}p = -\underbrace{g^\mathsf{T}B^{-1}g}_{>\,0} < 0$ unless $g = 0$.

If B happens to be the identity matrix, (2.14) simply is the steepest descent direction.

Proposition 2.11. *If B is positive definite, then the search direction $p := -B^{-1}g$ satisfies for $g \neq 0$ the* angle condition

$$\frac{g^\mathsf{T}p}{\|g\|_2\|p\|_2} \leq \frac{-1}{\sqrt{\kappa_2(B)}}.$$

Remark. The notation κ_2 stands for the condition number of a matrix in terms of the matrix 2-norm, see Definition A.14 in Appendix A.

The above angle condition is exploited by various algorithms. The vectors $-g$ and p are bounded away from 90° since the ratio on the left hand side is the cosine of the angle between g and p. An easy proof of Proposition 2.11 is given in [38].

In iteration l, a typical Newton-like method

1. computes an approximation $B_l \approx \nabla^2 f(x^l)$ at the iterate x^l,
2. solves $B_l q^l = -g^l$ and corrects q^l to get a p^l satisfying the angle condition,
3. performs an efficient line search along $x^l + \alpha p^l$ to compute a stepsize α_l
4. and sets $x^{l+1} := x^l + \alpha_l p^l$ as the next iteration point.

The way of computing B_l distinguishes different Newton-like methods.

Newton methods choose in in each iteration B_l as the exact Hessian at the current point x^l, i.e., $B_l = \nabla^2 f(x^l)$. A Cholesky decomposition $B_l = L_l L_l{}^\mathsf{T}$, where L_l is lower triangular, is computed in order to solve

$$B_l q^l = -g^l. \tag{2.15}$$

Efficient line searches in *damped* Newton methods will usually choose $\alpha < 1$. In *undamped* Newton methods, $\alpha = 1$ in every iteration. Computing the Hessian is a large amount of work. A numerical approximation is usually not very accurate. Basically, just analytic Hessians are useful which is why such techniques are usually only used if second order derivative information is known.

Modified Newton methods are used to safeguard positive definiteness. It may happen that $G_l = \nabla^2 f(x^l)$ becomes indefinite or negative definite or even singular. Then p^l might no longer be a descent direction. Modified Cholesky decompositions

$$G_l + E_l = L_l L_l{}^\mathsf{T}$$

are computed, where E_l is a positive semidefinite and usually diagonal *correction matrix*. Then, the matrix $B_l := L_l L_l{}^\mathsf{T}$ is positive definite and can be used in the local quadratic model.

Discrete Newton methods approximate the Hessian G_l at the iterate. In particular, B_l is computed by finite differences, which are explained in more detail in Section 2.4.4. A finite difference approximation $\tilde{G}_l$ of G_l is in general not symmetric. But symmetry is guaranteed by setting $\tilde{B}_l := \frac{1}{2}\big(\tilde{G}_l + \tilde{G}_l{}^\mathsf{T}\big)$. A modified Cholesky decomposition is computed as well in order to correct $\tilde{B}_l$ to a positive definite matrix B_l and to solve (2.15).

Quasi-Newton methods avoid the computation of the Hessian by performing an update procedure. They are based on the idea of building up curvature information as the iterations proceed. The *Quasi-Newton equation*

$$B_l(x^l - x^{l-1}) = g^l - g^{l-1} \tag{2.16}$$

originates from a multidimensional generalization of the secant method for root-finding. Quasi-Newton methods enforce this equation for Hessian approximations

in each iteration in order to solve $\nabla f(x) = 0$. The matrix B_l in (2.16) reflects the change in the gradient $\bar{y} := g^l - g^{l-1}$ with respect to the step $\bar{s} := x^l - x^{l-1}$. It is required to be positive definite. Hence, we can rewrite (2.16) as

$$\bar{s} = H\bar{y}, \tag{2.17}$$

where $H := B_l^{-1}$. This representation has the huge advantage that in each iteration equation (2.15) can be solved without the need for inversion. The computed search direction has then the form

$$q^l = -Hg^l.$$

During a single iteration, new information about the curvature of the function f is obtained along one search direction. Therefore, it is expected that H differs from the previous approximation $\bar{H} := B_{l-1}^{-1}$ only by a matrix of low rank.
Thus, after the iteration point x^l has been found by a line search, Quasi-Newton methods do not compute the Hessian at the new iterate but simply use an update (here in the sense of inverse Hessian approximation)

$$H = \bar{H} + U, \tag{2.18}$$

for some *update matrix* U.

Remark. The initial matrix B_0 is often a finite difference approximation of the Hessian at the starting point x^0 or the identity matrix if no additional information is given. With the latter choice, the first iteration is equivalent to gradient descent. Newton's method spends $\mathcal{O}(n^3)$ operations for the Cholesky decomposition of B_l in each iteration. A Quasi-Newton method needs only $\mathcal{O}(n^2)$ operations for solving (2.15) in every step.

Formula (2.18) is often understood as rank-one update

$$H = \bar{H} + uv^\mathsf{T}, \tag{2.19}$$

for some vectors $u, v \in \mathbb{R}^n$. The reformulated Quasi-Newton equation (2.17) yields

$$\bar{s} = \big(\bar{H} + uv^\mathsf{T}\big)\bar{y} = H\bar{y}, \quad \text{or} \quad u\big(v^\mathsf{T}\bar{y}\big) = \bar{s} - \bar{H}\bar{y}.$$

We assume that $\bar{s}$ is not equal to $\bar{H}\bar{y}$ and v is a vector such that $v^\mathsf{T}\bar{y}$ is nonzero. Then the vector u is given by

$$u = \frac{\bar{s} - \bar{H}\bar{y}}{v^\mathsf{T}\bar{y}}$$

and H takes the form

$$H = \bar{H} + \frac{(\bar{s} - \bar{H}\bar{y})v^\mathsf{T}}{v^\mathsf{T}\bar{y}}. \tag{2.20}$$

In the *Symmetric Rank 1* (SR1) update, v is chosen as a multiple of u in order to maintain the symmetry of the Hessian approximation in the iteration process. In this case, the rank-one update (2.20) becomes

$$H = \bar{H} + \frac{(\bar{s} - \bar{H}\bar{y})(\bar{s} - \bar{H}\bar{y})^\mathsf{T}}{(\bar{s} - \bar{H}\bar{y})^\mathsf{T}\bar{y}}.$$

The SR1 update possesses a nice feature, a *finite termination* property for quadratic functions. If n linearly independent iterates are taken, then the Hessian approximation converges to the exact Hessian of the quadratic function after at most n updates. Hence, the solution can be reached in at most $n + 1$ steps. A precise statement and its proof are given in [38].
Any rank-two correction in regard to the update formula (2.18) can can be written as

$$H = \bar{H} + uu^\mathsf{T} - vv^\mathsf{T},$$

for some vectors $u, v \in \mathbb{R}^n$ with $u^\mathsf{T} v = 0$.

The *Broyden-Fletcher-Goldfarb-Shanno* (BFGS) update is such a rank-two correction and is widely considered to be the most effective update formula in unconstrained optimization. It is accurately described in [11] and [14] and has the form

$$H = \left(I - \frac{\bar{s}\bar{y}^\mathsf{T}}{\bar{y}^\mathsf{T}\bar{s}}\right)\bar{H}\left(I - \frac{\bar{y}\bar{s}^\mathsf{T}}{\bar{y}^\mathsf{T}\bar{s}}\right) + \frac{\bar{s}\bar{s}^\mathsf{T}}{\bar{y}^\mathsf{T}\bar{s}},$$

where $I \in \mathbb{R}^{n\times n}$ is the identity matrix. Using the *Sherman-Morrison-Woodbury formula* [15, p. 65] we can transform it into the direct update

$$B = \bar{B} - \frac{\bar{B}\bar{s}\bar{s}^\mathsf{T}\bar{B}}{\bar{s}^\mathsf{T}\bar{B}\bar{s}} + \frac{\bar{y}\bar{y}^\mathsf{T}}{\bar{y}^\mathsf{T}\bar{s}},$$

for $B := B_l$ and $\bar{B} := B_{l-1}$.

Remark. The SR1 and BFGS updates ensure that all (inverse) Hessian approximations remain positive definite, provided that the respective update directions fulfill the *Wolfe condition* [38, p. 71]. However, in practice rounding errors may cause the updated matrix to become singular or indefinite. This can be avoided by updating the Cholesky decomposition of an approximate Hessian itself. According to Gill [14], if the Cholesky factors of B are available, equation (2.15) can be solved with similar cost as in inverse Hessian updating , i.e., by using $\mathcal{O}(n^2)$ operations only. The updates of the Cholesky factors of B and the update of H are obtained in a comparable number of operations.

For smooth medium size problems without special structure, Quasi-Newton methods are considered to be the most efficient methods in unconstrained optimization. Usually, they exhibit locally superlinear convergence. Newton methods can even achieve quadratic convergence rates but are far more restrictive, as their performance relies heavily on available information. Precise convergence proofs for Newton and Quasi-Newton methods can be found in [40].

Nonlinear conjugate gradient method

Linear conjugate gradient methods are among the most useful techniques for solving large linear systems of equations of the form

$$Ax = b \tag{2.21}$$

for the vector $x \in \mathbb{R}^n$, where $b \in \mathbb{R}^n$ is known and $A \in \mathbb{R}^{n \times n}$ is symmetric positive definite. The quadratic function

$$q(x) := \frac{1}{2} x^\mathsf{T} A x - b^\mathsf{T} x$$

is convex and thus the quadratic program

$$\underset{x \in \mathbb{R}^n}{\text{minimize}} \quad q(x)$$

has a unique minimizer x^* and is equivalent to solving (2.21), since $\nabla q(x^\star) = 0$. Fletcher and Reeves [12] showed how to extend the linear conjugate gradient method by Hestenes and Stiefel [20] to general unconstrained nonlinear optimization problems. The proposed algorithm and its resulting modifications are known as *nonlinear conjugate gradient methods.*
In contrast to Newton-like methods, conjugate gradient (CG) algorithms do not need to store any matrices. A quadratic function is not explicitly modeled, so Hessian approximations are not needed and only gradients are used.
In the l-th iteration of a CG method, the new iteration point x^{l+1} is determined as

$$x^{l+1} = x^l + \alpha_l p^l,$$

where x^l is the current iterate and $\alpha_l > 0$ is the stepsize parameter found by a line search along the search direction

$$p^l := \begin{cases} -g^l & \text{if } l = 0, \\ -g^l + \beta_{l-1} p^{l-1} & \text{if } l \geq 1. \end{cases} \tag{2.22}$$

The choice of the formula for the *conjugate gradient parameter* $\beta_{l-1} \in \mathbb{R}$ defines different CG algorithms. For example, Fletcher and Reeves used the parameter

$$\beta_{l-1} := \frac{\left\| g^l \right\|_2}{\left\| g^{l-1} \right\|_2}.$$

However, CG methods of this form impose strong restrictions on the line search technique used and convergence analysis is rather involved. A modified CG algorithm which requires only an efficient line search and that reduces the likelihood for zigzagging by preconditioning is explained in [37]. Zigzagging can be avoided by choosing the search direction p as close as possible to the previous search direction $p_{old} \in \mathbb{R}^n$ with respect to the ellipsoidal norm[4]. The following statement holds for a fixed positive definite preconditioner $B \in \mathbb{R}^{n \times n}$ and is proven in [37].

[4]The ellipsoidal norm is introduced in Appendix A, see Definition A.13.

Theorem 2.12. *For all $p \in \mathbb{R}^n$ with $g^\mathsf{T} p < 0$, the distance*

$$\left\|p - p_{old}\right\|_P^2 := (p - p_{old})^\mathsf{T} P (p - p_{old})$$

is minimal for

$$p = p_{old} - \lambda P^{-1} g, \quad \text{where } \lambda := \frac{g^\mathsf{T} p_{old} - g^\mathsf{T} p}{g^\mathsf{T} P^{-1} g}.$$

If an efficient line search is used, convergence analysis [37] shows that p^l is either parallel to the Newton direction (2.14) or certain conditions [37, p. 19–21] are satisfied. If the latter do not hold in an iteration step, the preconditioned CG algorithm consequently chooses the Newton direction in order to preserve local linear convergence. This is called a *restart* since the method initially starts with a line search along this direction (unlike algorithms without preconditioning, cf. (2.22)).

In iteration l, the preconditioned nonlinear conjugate gradient method

1. computes the gradient $g^l = \nabla f(x^l)$ at the iterate x^l,
2. chooses either the search direction $p^l = p^{l-1} - \lambda_l P^{-1} g^l$ or makes a restart, i.e., $p^l = -P^{-1} g^l$,
3. performs an efficient line search along $x^l + \alpha p^l$ to compute a stepsize α_l
4. and sets $x^{l+1} := x^l + \alpha_l p^l$ as the next iteration point.

The preconditioned CG method can be transformed into a traditional CG method by setting $P = I$ and by scaling the corresponding vectors and parameters. Similar to the SR1 update, a finite termination property for quadratic functions can be proven, see [37]. If such a CG algorithm is applied to a quadratic function, it stops after at most n iterations with a minimizer or with a direction of infinite descent, where n is the number of variables.

Remark. Conjugate gradient algorithms use very little memory and can therefore be used for large-scale problems where other methods fail because involved matrices are too large or too dense. But the saving in storage typically results in an increased number of iterations. In general, convergence is much slower than in Newton or Quasi-Newton methods.

The preceding methods are the most important local techniques for unconstrained optimization. In practice however, it is often the case that not all possible values of the variables are acceptable. We will now briefly discuss basic techniques used for solving such constrained problems.

2.4.3 Constrained optimization techniques

A mathematical program of type (2.6) demands special strategies in order to handle its constraint functions. This constrained optimization problem (COP) can be equivalently formulated as

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & c(x) = 0, \quad x \in \mathbf{x}, \end{aligned} \tag{2.23}$$

where

$$\mathbf{x} := [\underline{x}, \overline{x}] := \{x \in \mathbb{R}^n \mid \underline{x} \leq x \leq \overline{x}\}$$

is a bounded or unbounded *box* in $\mathbb{R}^n$ describing the bounds on the variables. In this part, the objective function $f : \mathbf{x} \to \mathbb{R}$ and the constraint function $c : \mathbf{x} \to \mathbb{R}^r$ are assumed to be continuously differentiable. A bound constrained optimization problem (BOPT) then takes the form

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & x \in \mathbf{x}. \end{aligned} \tag{2.24}$$

That is, its feasible region is simply the box $\mathbf{x}$.

Active set methods are among the most effective techniques for solving a **bound constrained optimization problem** (2.24). We will only explain the basic idea behind such solvers. There are many different strategies associated with the iteration procedure which is why we do not give a model algorithm. However, adequate theory such as optimality conditions for bound constraint optimization, as well as a pseudocode of a BOPT algorithm can be found in [37, p. 24–30]. In [23], even a derivative-free modification of this algorithm is presented, the *limited memory method for bound constrained optimization* (LMBOPT). Active set algorithms for bound constrained optimization are based on the following Definition 2.13. The appropriate definition regarding general constrained optimization problems (2.23) is stated in [40, p. 308].

Definition 2.13. Let $\hat{x} \in \mathbf{x}$ for a box $\mathbf{x} = [\underline{x}, \overline{x}]$ in $\mathbb{R}^n$. An index $i \in \{1, \ldots, n\}$ is *active* for $\hat{x} \in \mathbb{R}^n$ if $\hat{x}_i = \underline{x_i}$ or $\hat{x}_i = \overline{x}_i$. The component $\hat{x}_i$ which corresponds to an active index is also called *active*. If an index (or component) is not active, it is *nonactive* or *free*. An element for which all indices are active is termed *corner* of $\mathbf{x}$.

Given a feasible point $x \in \mathbf{x}$ and its gradient g, such algorithms try to find the optimum without leaving the box. If $g_i \neq 0$ for a nonactive component x_i, a reduction in the objective function f can be obtained by slightly changing x_i. The direction chosen by a line search depends on whether $g_i > 0$ or $g_i < 0$. However, if x_i is active, only points along one direction are feasible. The value of f can can then be reduced by moving slightly in this feasible direction only when

$$\begin{cases} g_i < 0 & \text{if } x_i = \underline{x_i}, \\ g_i > 0 & \text{if } x_i = \overline{x_i}. \end{cases}$$

Performing a classical line search along a ray may lead to infeasible points because of the bound constraints. In order to stay in the feasible region, the search path must have the ability to be bent. This is achieved by projecting the ray into the box with the use of feasible projections.

Definition 2.14. Let $\mathbf{x} = [\underline{x}, \overline{x}]$ be a box in $\mathbb{R}^n$ and $x \in \mathbb{R}^n$. The *feasible projection* π of x onto $\mathbf{x}$ is defined as

$$\pi[x]_i := \max\left(\underline{x_i}, \min(\overline{x_i}, x_i)\right) = \begin{cases} \underline{x_i} & \text{if } x_i < \underline{x_i}, \\ \overline{x_i} & \text{if } x_i > \overline{x_i}, \\ x_i & \text{otherwise.} \end{cases}$$

Remark. We note that for $x \in \mathbf{x}$ we have $\pi[x] = x$. Conversely, if $\pi[x] = x$, we find that $x \in \mathbf{x}$ and hence $\pi[\mathbb{R}^n] = \mathbf{x}$.

Thus, active set methods for bound-constrained optimization find new iteration points by performing a line search along the *bent search path*

$$x(\alpha) := \pi[x + \alpha p],$$

where p is the search direction, $x \in \mathbf{x}$ and α is the stepsize parameter. The bent search path is piecewise linear and has breakpoints at the $\alpha > 0$ with $x_i + \alpha p_i \in \{\underline{x_i}, \overline{x_i}\}$. In each iteration, the active components remain unchanged and only a subset of the nonactive ones is changed. To account for this, algorithms use a *working set* $\mathcal{W} \subset \{1, \ldots, n\}$ and require $p_i = 0$ for $i \notin \mathcal{W}$. The sensible choice of the working set is crucial for the performance of such methods, see [37].

According to Neumaier and Schichl [38], there are basically four successful classes of algorithms for solving a **constrained optimization problem** (2.23), namely

- penalty methods,
- augmented Lagrangian methods,
- barrier methods,
- and reduced gradient methods.

In bound constrained optimization, the clever choice of a line search technique along a bent search path guarantees feasibility of a generated sequence of iterates. Thus, only the objective function needs to be considered in determining whether an improvement has occurred. In problems where nonlinear constraint functions are involved, feasibility cannot be easily maintained. In such cases, it is not clear if a new iterate x^{l+1} is better than the old point x^l. It is necessary to add a penalizing term to the objective function in order to account for possible constraint violations. The resulting new objective is referred to as *merit* or *penalty function*.

Penalty methods transform constrained problems of the form (2.23) into unconstrained problems by adding penalties to the objective function. They are classified as *infeasible methods* since, most of the time, the iteration points x^l are infeasible. The condition $c(x) = 0$ is only achieved in the limit. Thus, the penalty function is

$$f_\sigma(x) := f(x) + \sigma\phi\big(\|c(x)\|\big), \tag{2.25}$$

for some *penalty parameter* $\sigma > 0$ and a continuously differentiable penalizing functional $\phi : \mathbb{R}_+ \to \mathbb{R}_+$ with $\phi(0) = 0$. The functional needs to be monotone increasing and must grow at least as fast as f at points far away from the constraint surface. The behavior of the unconstrained *penalty problem*

$$\begin{array}{ll} \underset{x \in \mathbb{R}^n}{\text{minimize}} & f_\sigma(x) \end{array} \tag{2.26}$$

strongly depends on the particular choice of the parameter σ and the function ϕ. Usually, ϕ is fixed and σ is adapted accordingly. A penalty function is called *exact* if the unconstrained minimization points of the resulting penalty problem are also the solution of the constrained problem, otherwise *inexact*. In order to solve (2.26), an algorithm has to find a good search direction. But the function f_σ can be nonsmooth, i.e., the gradient of f_σ is no viable option. A way to find acceptable directions in every iteration is to replace the objective function with a quadratic model and approximately solve convex quadratic subprograms before each line search. The resulting method is called *sequential quadratic programming* (SQP).

Augmented Lagrangian methods also penalize the objective function to counteract constraint violation. They are known as *methods of multipliers* since explicit Lagrangian multipliers are successively estimated. If $z \in \mathbb{R}^r$ is a fixed vector, then the problem

$$\begin{array}{ll} \underset{x}{\text{minimize}} & f(x) - z^\mathsf{T} c(x) \\ \text{subject to} & c(x) = 0, \quad x \in \mathbf{x}, \end{array} \tag{2.27}$$

is equivalent to (2.23). The objective function $\mathcal{L}_z(x) := f(x) - z^\mathsf{T} c(x)$ of this problem is the *Lagrangian* of the constrained optimization problem (2.23) and z is referred to as *Lagrange multiplier*. As in (2.25), we add a penalizing term to the objective and get the merit function

$$f_{\sigma,z}(x) := f(x) - z^\mathsf{T} c(x) + \sigma\phi\big(\|c(x)\|\big),$$

yielding the unconstrained minimization problem

$$\begin{array}{ll} \underset{x \in \mathbb{R}^n}{\text{minimize}} & f_{\sigma,z}(x). \end{array}$$

Remark. The particular choice of the parameter σ can be difficult for penalty methods. If σ is large, then f_σ in (2.25) is dominated by the penalty term (unless infeasibility is small). Geometrically, this corresponds to a steep valley with narrow grooves in which line search techniques can only take tiny steps. So for numerical reasons, we want to keep σ reasonable small. Augmented Lagrangian methods achieve this by reducing the slope of the valley with clever choices for the multiplier z. That is, z is taken such that the gradient of $\mathcal{L}_z$ is small.

In practice, often a solution of only limited accuracy is satisfactory. A disadvantage of infeasible methods is that if the iteration process is terminated prematurely, the final iterate may not be feasible and hence not usable. This is not the case for the following *feasible methods*, where feasibility of all iteration points is enforced.

Barrier methods use an analogous idea to that of penalty methods, i.e., a term is added to the objective function f such that the constrained optimization problem transforms into a unconstrained one. But instead of merely adding a penalty for feasibility, points are simply not allowed to become infeasible. If the unconstrained minimum of f exists, it will most likely occur at an infeasible point. Thus, a modified objective function is designed to create a barrier which prevents iterates from leaving the feasible region. This is why barrier methods are also known as *interior point methods*. The optimization problem is now required to be of the form (cf. (2.23))

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & c(x) \geq 0, \quad x \in \mathbf{x}. \end{aligned} \tag{2.28}$$

The initial starting point $x_0 \in \mathbf{x}$ needs to be strictly feasible, i.e., $c(x_0) \in \mathcal{S}$ with $\mathcal{S} := \{x \in \mathbf{x} \mid c(x) > 0\}$. A *barrier function* $\varphi : \mathcal{S} \to \mathbb{R}_+$ with the property

$$\varphi(x) \to \infty \text{ as } x \to \partial\mathcal{S}$$

is added to the objective function. That is, the modified objective has the form

$$f_\mu(x) := f(x) + \mu\varphi(x),$$

where $\mu > 0$ is some *barrier parameter*. The unconstrained problem to be solved is

$$\underset{x \in \mathbb{R}^n}{\text{minimize}} \quad f_\mu(x). \tag{2.29}$$

Algorithms fix the barrier function φ and adapt μ accordingly. Two popular choices for φ are the *log-barrier function*

$$\varphi(x) := -\sum_{i=1}^{r} \log\big(c_i(x)\big)$$

and the *inverse barrier function*

$$\varphi(x) := \sum_{i=1}^{r} \frac{1}{c_i(x)}.$$

Remark. We observe that both the log-barrier and inverse barrier functions yield very large values for points close to boundary of the feasible region, i.e., points x with $c_i(x) \approx 0$ for some i. Feasible points which lie away from the boundary correspond to small function values. But the solution to an inequality-constrained optimization problem (2.28) is often found at the boundary where the barrier function *blows up*. In such a case the barrier term in (2.29) needs to be removed. This is the reason why proper algorithms gradually reduce the barrier parameter μ (which may initially be chosen large) towards zero.

Reduced-gradient methods for solving constrained optimization problems of the form (2.23) are based on extending methods for linear constraints to the nonlinear case. The basic idea is to stay on a subset of the variables for which the nonlinear constraints hold while reducing the objective function. The requirement of satisfying the constraints usually reduces the dimensionality of the optimization. Feasibility is maintained in every iteration due to the clever choice of an active set strategy and the determination of suitable search directions. In fact, reduced-gradient-type algorithms for nonlinear constrained problems mostly differ in the technique used for achieving feasibility and reducing the objective function. But they are all based on the following underlying principle.

We consider the COP (2.23), i.e., $c(x)$ represents r nonlinear equality constraints for $x \in \mathbb{R}^n$. At iteration l of an algorithm the current iterate $x^l \in \mathbb{R}^n$ is feasible, meaning that $c(x^l) = 0$. The next point x^{l+1} must satisfy $c(x^{l+1}) = 0$ and $f(x^{l+1})$ has to be sufficiently smaller than $f(x^l)$. The desired result is the iteration step

$$s^l := x^{l+1} - x^l.$$

A typical iteration starts with enforcing the constraints for the new iterate by ensuring that

$$c(x^{l+1}) = c(x^l + s^l) = 0. \tag{2.30}$$

In order to accomplish this we linearly approximate c by applying its Taylor series expansion about x^l, i.e.,

$$c(x^l + s^l) \approx c(x^l) + \nabla c(x^l)s^l, \tag{2.31}$$

where $\nabla c(x^l) \in \mathbb{R}^{r \times n}$ is the Jacobian of c evaluated at x^l (cf. (3.4)). From (2.30) and (2.31) and the fact that x^l is feasible, we see that a vector $p^l \in \mathbb{R}^n$ which approximates s^l satisfies

$$\nabla c(x^l)p^l = 0. \tag{2.32}$$

This is a set of r linear equality constraints in terms of p^l. We can now exploit the following property of methods for linear constraints, which is shown in [14, p. 68].
The step from any feasible point to any other feasible point must be orthogonal to the rows of the matrix $\nabla c(x^l)$.
So, if $M_l \in \mathbb{R}^{n \times (n-r)}$ denotes a matrix whose columns form a basis for the set of vectors orthogonal to the rows of $\nabla c(x^l)$, then any p^l that satisfies (2.32) must be a linear combination of the columns of M_l. Hence, we find that

$$p^l = M_l p_M \tag{2.33}$$

for some vector $p_M \in \mathbb{R}^{n-r}$. The vector p^l lies entirely in the null space of $\nabla c(x^l)$. Hence, the reduction to $n - r$ degrees of freedom is represented by p_M. Such a p_M can be determined by minimizing an approximation to the objective function, solely expressed in terms of p_M.

Remark. A possible choice for p_M is the the vector $p_M = -{M_l}^\mathsf{T} g_l$, where $g_l = \nabla f(x^l)$. If it is possible to define p_M in terms of second-order information, an improved rate of convergence can be achieved.

Reduced-gradient methods do not perform a line search along the classical search direction $x^l + \alpha p^l,\ \alpha \geq 0$. In order to restore feasibility, they take as approach that s^l has the form

$$s^l = \alpha p^l + Y_l p_Y,$$

where $Y_l \in \mathbb{R}^{n \times r}$ is a matrix whose columns form a basis for the column space of $\nabla c(x^l)^\mathsf{T}$. The vector $P_Y \in \mathbb{R}^r$ can be understood as adjustment of the search to the nonlinear constraints. Two adaptive subproblems need to be solved for finding a value for α and a corresponding p_Y such that (2.30) holds. Furthermore, α is required to yield

$$f(x^l + s^l) < f(x^l)$$

with sufficient decrease in objective value. More details can be found in [14, p. 222].

The *generalized reduced-gradient* (GRG) method is an example of a reduced-gradient-type method. It is based on a particular form of the matrix M_l which arises from partitioning the Jacobian $\nabla c(x^l)$. Assuming that the Jacobian is nonsingular and that its first r columns are linearly independent, we can partition it as

$$\nabla c(x^l) = \begin{bmatrix} B & N \end{bmatrix}, \tag{2.34}$$

with nonsingular $B \in \mathbb{R}^{r \times r}$ and $N \in \mathbb{R}^{r \times (n-r)}$. Using a *variable-reduction method*, which is explained in [14, p. 163], the matrix M_l takes the form of

$$M_l = \begin{bmatrix} -B^{-1}N \\ I_{n-r} \end{bmatrix}. \tag{2.35}$$

Now let p^l be partitioned according to the columns of (2.34), i.e.,

$$p^l = \begin{pmatrix} p_B \\ p_N \end{pmatrix},$$

where $p_B \in \mathbb{R}^r$ is the vector of *dependent* or *basic* variables and $p_N \in \mathbb{R}^{n-r}$ the vector of *independent* or *nonbasic* variables. The constraints (2.30) imply that the r basic variables p_B can be expressed in terms of the $n - r$ nonbasic variables p_N. That is, $p_M = p_N$ and thus the original variables of the problem correspond to the reduction in dimensionality to $n - r$. The term $M_l{}^\mathsf{T} g_l$ with M_l as in (2.35) is given the name *reduced gradient*.

Remark. Reduced-gradient methods ensure that the nonlinear constraints remain satisfied at every iteration, which mostly comes with high computational cost. But they can perform well with problems where the constraints are almost linear.

2.4.4 Derivative-free methods

A rule of thumb for choosing a solution method for an optimization problem is *to make use of as much derivative information as can reasonably be provided* [14, p. 285]. So, if we face an optimization problem where only function values are available, it is essential to determine whether the objective function of a problem is smooth, even though its derivatives may not be computable. *Derivative-free methods* are often designed to accumulate curvature information by approximating the gradient or even the Hessian with *finite differences*. In fact, an obvious strategy is to use an established gradient-method and replace its exact gradient with a finite difference approximation, as it is done within this thesis. Finite difference analogues of the Gauss-Newton and Levenberg-Marquardt method are presented in Section 4. However, if the approximation of derivatives is unreliable or comes with high computational cost, one might prefer methods using heuristics which are aimed to find a good solution in reasonable time.

Finite difference approximations in derivative-free algorithms for the i-th component g_i of the gradient g of f are usually realized by using *forward differences*

$$g_i(x) \approx \frac{f(x + he_i) - f(x)}{h} \tag{2.36}$$

or *central differences*

$$g_i(x) \approx \frac{f(x + he_i) - f(x - he_i)}{2h}, \tag{2.37}$$

where $x \in \mathbb{R}^n$, $h > 0$ is sufficiently small and $e_i \in \mathbb{R}^n$ is defined as the coordinate vector

$$e_i := \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \leftarrow i\text{-th position.}$$

The errors of the finite difference approximations for smooth functions can be easily determined. For the forward differences, rearranging the Taylor series expansion

$$f(x + h) = f(x) + hg(x) + \mathcal{O}(h^2)$$

yields

$$\frac{f(x + h) - f(x)}{h} - g(x) = \mathcal{O}(h).$$

And by combining the Taylor series expansions of $f(x + h)$ and $f(x - h)$ we find

$$\frac{f(x + h) - f(x - h)}{2h} - g(x) = \mathcal{O}(h^2)$$

for the central differences, where we truncated after the second derivative terms.

If we let $h \to 0$, we see that the central differences yield a more accurate approximation to the gradient. However, they require $\mathcal{O}(2n)$ additional function evaluations instead of just $\mathcal{O}(n)$ additional evaluations in forward differences. Hence, central differences are usually not used unless the relative error[5] in the forward-difference approximation is unacceptable. Such an unacceptable error may occur when the difference in function values is small relative to h. For example, forward-difference approximations may perform badly during the last few iterations of an algorithm where $\|g(x)\|$ is small. That means that the approximation can be unreliable, even for well scaled problems. In such cases, one may switch to central differences. Nevertheless, we should try to return to forward differences in the case that a central-difference approximation was needed at some point far from the optimum.
Backward differences evaluate the function f at x and $x-h$ instead of $x+h$ and x in forward differences. Higher order finite differences are based on the previous formulas (2.36) and (2.37) and are constructed recursively.

Direct search methods (cf. Section 2.3) do not approximate derivatives numerically. In fact, they do not even attempt to approximate $f(x)$ in a neighborhood of x by a smooth function [47, p. 615]. For example, the DUD algorithm in Section 4.3 forgoes computing gradients and finite differences but does linear approximations to involved functions and is therefore not considered a direct method. Instead, direct optimization algorithms solely rely upon the comparison of function values. That is, in each iteration, they simply aim to find points for which a reduction in the objective is achieved. That is, such methods work *directly* with values of the objective function in the search for an optimum.
Direct search methods perform well in certain cases but their general usefulness is limited. For smooth functions, they are usually outperformed by gradient methods and derivative-free methods using finite difference approximations both in reliability and convergence speed [2, p. 120]. Due to the heuristic nature of direct algorithms, it is often the case that a large number of parameters is demanded. A successful optimization might therefore heavily depend on the choice of those parameters. In [14, p. 63] it is recommended that such methods should be used only for nonsmooth problems or if no other suitable alternative method is available .

Remark. The **nonlinear simplex algorithm** [35] by Nelder and Mead is an example of a direct search method. It is based on a *simplex*, which is the generalization of the notion of a triangle to n dimensions. In attempting to find a minimizer, the simplex moves, expands, contracts, and distorts its shape. This algorithm performs well in two dimensions. However, as the dimension of the problem grows it becomes inefficient and is therefore not part of this work.

Line searches without gradients use a concept different than that discussed in Section 2.4.1. It is not possible to compute the classical sufficient descent conditions and it is not even clear if a search direction is a descent direction when the gradient is not available. Hence, such line searches techniques must search along the full path instead of in one direction only. An appropriate algorithm and the theory which comes along with it is described in [38, p. 121–128].

[5]The relative error is defined in Appendix A, Definition A.19.

This page intentionally left blank.

3 Nonlinear Equations and Least Squares

The derivative-free optimization algorithms in this thesis aim to solve problems of the following classes:

(i)	Systems of nonlinear equations:	$F(x) = 0,$	$F : \mathbb{R}^n \to \mathbb{R}^m$
(ii)	Nonlinear programs:	$\min_x f(x),$	$f : \mathbb{R}^n \to \mathbb{R}$
(iii)	Nonlinear least squares problems:	$\min_x \frac{1}{2}\|F(x)\|_2^2,$	$F : \mathbb{R}^n \to \mathbb{R}^m$

We can turn the nonlinear system (i) into a nonlinear least squares problem (iii) by observing that $F(x) = 0$ if and only if $\|F(x)\|_2^2 = 0$. And it is easy to see that the nonlinear least squares problem (iii) is a special case of the unconstrained optimization problem (ii) as we simply minimize the function $f : \mathbb{R}^n \to \mathbb{R}$ defined by $f(x) := \frac{1}{2}\|F(x)\|_2^2$.
Nonlinear programs have been thoroughly discussed in Section 2. In the following, **systems of nonlinear equations** are explained in more detail. Our algorithms try to solve them by minimizing **nonlinear least squares** problems. Often, solutions to **linear least squares** subproblems are computed in the iterative process. Additionally, least squares methods are particularly powerful in **data fitting** and **regression**, which is why we provide some insight into these topics as well. The references used for this section are [2, 8, 11, 13, 14, 15, 19, 40, 47, 49].

3.1 Systems of Nonlinear Equations

A *system of nonlinear equations* can be written in terms of a vector valued function $F : \mathbb{R}^n \to \mathbb{R}^m$, resulting in the representation

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

That is, F has the form

$$F(x) = \begin{pmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_m(x) \end{pmatrix}, \tag{3.2}$$

where the functions $f_i : \mathbb{R}^n \to \mathbb{R},\ i = 1, 2, \ldots, m$, are assumed to be continuously differentiable and often interpreted as *residuals*. At least one of the f_i's is nonlinear. A vector $x^* \in \mathbb{R}^n$ which satisfies the system (3.1) is called a *solution* or *root* of the nonlinear equations. Methods which aim to solve (3.1) are therefore often referred to as *root-finding algorithms*. In general, the system (3.1) may have no solution, a unique solution or many solutions. Standard solvers for nonlinear equations aim to find exact solutions of *well-determined* systems, i.e., systems for which the number of variables n and the number of equations m are the same. If $m < n$, (3.1) is *underdetermined* and may have infinitely many solutions and if $m > n$, the system is *overdetermined* and may have no non-trivial solution. So, it is usually not possible to obtain a solution for an overdetermined system which satisfies all equations. However, in optimization, *methods for least squares* can be employed to approximately solve such a system.

3.2 Nonlinear Least Squares

Numerous optimization algorithms are designed to iteratively seek a solution of the *nonlinear least squares* (NLLS) problem

$$\underset{x \in \mathbb{R}^n}{\text{minimize}} \quad f(x) = \frac{1}{2}\sum_{i=1}^{m} f_i(x)^2 = \frac{1}{2}F^{\mathsf{T}}F, \tag{3.3}$$

where $F = F(x)$ is as in (3.2) and $m \geq n$. A minimizer $x^* \in \mathbb{R}^n$ of (3.3) is called a *least squares solution.*

Nonlinear least square problems are easier to solve than most general unconstrained optimization problems. The special form of the objective function and its derivatives allow for efficient algorithms. The least squares solution x^* is contained among the zeros of the gradient $\nabla f(x)$ of f, which can be expressed in terms of the Jacobian. Let $J \in \mathbb{R}^{m\times n}$ be the Jacobian matrix of F, defined for $x \in \mathbb{R}^n$ by

$$J(x) = \left[\frac{\partial f_i}{\partial x_j}(x)\right]_{\substack{i=1,\ldots,m \\ j=1,\ldots,n}} = \begin{bmatrix} \nabla f_1(x)^{\mathsf{T}} \\ \nabla f_2(x)^{\mathsf{T}} \\ \vdots \\ \nabla f_m(x)^{\mathsf{T}} \end{bmatrix}, \tag{3.4}$$

where $\nabla f_i(x)$ is the gradient of $f_i,\ i = 1, 2, \ldots, m$. Thus, the gradient of f is

$$\nabla f(x) = \sum_{i=1}^{m} f_i(x)\nabla f_i(x) = J(x)^{\mathsf{T}}F(x). \tag{3.5}$$

If the $f_i,\ i = 1, \ldots, m$, are twice continuously differentiable, then the Hessian of f is

$$\begin{aligned} \nabla^2 f(x) &= \sum_{i=1}^{m} \nabla f_i(x)\nabla f_i(x)^{\mathsf{T}} + \sum_{i=1}^{m} f_i(x)\nabla^2 f_i(x) \\ &= J(x)^{\mathsf{T}}J(x) + \sum_{i=1}^{m} f_i(x)\nabla^2 f_i(x). \end{aligned} \tag{3.6}$$

In practice, the computation of the residuals $f_i,\ i = 1, \ldots, m$, is often relatively easy or inexpensive. Hence, the gradient of f can be efficiently obtained by using formula (3.5). But the distinctive feature of least squares problems is found in representation (3.6). The first term $J^{\mathsf{T}}J$ comes practically "for free" as the Jacobian $J = J(x)$ is usually already stored from a preceding calculation of the gradient $J^{\mathsf{T}}F$. Second order information is only processed in the summation term. It is negligible when the residuals f_i are expected to be small or almost linear near the solution, implying that $\nabla^2 f_i(x)$ is small. Therefore, in such cases, we find that

$$\nabla^2 f(x) \approx J(x)^{\mathsf{T}}J(x). \tag{3.7}$$

Most methods for nonlinear least squares problems exploit these structural properties of the gradient and the Hessian. For example, the finite differences analogues of the Gauss-Newton and Levenberg-Marquardt algorithms, which are discussed in Section 4, use Jacobian approximations in their routines in order to employ formulas (3.5) and (3.7).

Remark. We did not exclude the case where $m = n$ in the NLLS problem (3.3). Even if a nonlinear system of equations is well-determined, an exact solution may not exist. We can choose a NLLS method instead of an algorithm specifically designed for nonlinear equations. This is advisable when inaccuracies in the definition of F prevent the system (3.1) from having a solution. However, if the Jacobian $J(x)$ of F is singular, convergence to a point that is not a solution of (3.1) is possible. This is due to the fact that the gradient $J(x)^\mathsf{T}F(x)$ may vanish even if $F(x)$ does not.

3.2.1 Linear least squares

A special case of the nonlinear least squares problem (3.3) is the *linear least squares* (LLS) problem

$$\underset{x \in \mathbb{R}^n}{\text{minimize}} \quad \frac{1}{2}\big\|Ax - b\big\|_2^2, \tag{3.8}$$

where $A \in \mathbb{R}^{m\times n},\ b \in \mathbb{R}^m$ and $m \geq n$. That is, $F(x) = Ax - b$ and each residual is a linear function $f_i(x) = A_{i:}x - b_i,\ i = 1, \ldots, m$, where $A_{i:}$ denotes the i-th row of the matrix A.
We note that $f(x) = \frac{1}{2}\|Ax - b\|_2^2$ is a differentiable function of $x \in \mathbb{R}^n$. Hence, the minimizers of f satisfy

$$\nabla f(x) = A^\mathsf{T}(Ax - b) = 0,$$

yielding the *normal equations*

$$A^\mathsf{T}Ax = A^\mathsf{T}b. \tag{3.9}$$

Solving (3.9) for x is the classical way to solve the LLS problem. The following statement is well known and its proof can be found in [15, p. 260] or [49, p. 81].

Theorem 3.1. *Let $A \in \mathbb{R}^{m\times n},\ b \in \mathbb{R}^m$ and $m \geq n$. The solution to the LLS problem (3.8) is the set of points*

$$\left\{x^* \in \mathbb{R}^n \mid A^\mathsf{T}(Ax^* - b) = 0\right\}.$$

If A has full rank, i.e., if the columns of A are linearly independent, then x^* is unique, $A^\mathsf{T}A$ is nonsingular and

$$x^* = \left(A^\mathsf{T}A\right)^{-1}A^\mathsf{T}b.$$

If we have a matrix $A \in \mathbb{R}^{m\times n}$ $(m \geq n)$ with full column rank n, then it is easy to see that $A^\mathsf{T}A \in \mathbb{R}^{n\times n}$ is symmetric and positive definite. Hence, a typical *algorithm for solving the normal equations* (3.9) performs a Cholesky decomposition of $A^\mathsf{T}A$:

1. Compute the matrix $A^\mathsf{T}A$ and the vector $A^\mathsf{T}b$.
2. Compute the Cholesky decomposition $A^\mathsf{T}A = LL^\mathsf{T}$.
3. Solve the lower-triangular system $Ly = A^\mathsf{T}b$ for y (forward substitution).
4. Solve the upper-triangular system $L^\mathsf{T}x = y$ for x (back substitution).

Remark. Such a procedure can be easily executed in MATLAB by using the *backslash operator* \ explained in [30, p. 7811–7819]. Assuming that $C = A^\mathsf{T}A$ and $d = A^\mathsf{T}b$ have already been computed, we can solve the normal equations (3.9) simply by entering **C\d**. MATLAB then selects an implemented *Cholesky Solver* [29, p. 196–197] which precisely follows the steps 2–4 of the preceding page. Furthermore, if $A^\mathsf{T}A$ is neither positive nor negative definite, an *LDL solver* [29, p. 828–829] is invoked which may be still able to find a non-unique solution to the normal equations but needs to solve an additional system for the diagonal matrix D.

The computational work of a **method for normal equations** is $\mathcal{O}\big(mn^2 + \frac{n^3}{3}\big)$, where $\mathcal{O}(mn^2)$ operations are needed for the matrix multiplication $A^\mathsf{T}A$ and $\mathcal{O}\big(\frac{n^3}{3}\big)$ for the Cholesky decomposition. The compression of the input data $A \in \mathbb{R}^{m\times n}$ to the matrix $A^\mathsf{T}A \in \mathbb{R}^{n\times n}$ is attractive for the user since $m \gg n$ in most applications. Consequently, these algorithms are considered very fast and are frequently used in practice. However, care has to be taken after $A^\mathsf{T}A$ has been formed. Since $A^\mathsf{T}A$ is symmetric and positive definite, it is invertible and we find for its condition number

$$\kappa_2(A^\mathsf{T}A) = \big\|A^\mathsf{T}A\big\|_2 \cdot \big\|(A^\mathsf{T}A)^{-1}\big\|_2 = \frac{\sigma^2_{\max}(A)}{\sigma^2_{min}(A)} = \kappa_2^2(A), \tag{3.10}$$

where $\sigma_{\max}(A)$ is the largest and $\sigma_{\min}(A)$ the smallest singular value of A. Equation (3.10) is justified by (A.1.3) in Appendix A and the fact that the singular values of $A^\mathsf{T}A$ are the squared singular values of A [15, p. 77]. Hence, we find that the relative error in the computed solution is proportional to the square of the condition number of A [15, p. 263]. Furthermore, rounding errors which appear due to bad conditioning may cause the Cholesky factorization process to break down. Thus, algorithms for solving the normal equations are numerically unstable and should not be used if involved matrices are expected to be *ill-conditioned*.
Popular methods for solving the LLS problem (3.8) which avoid the squaring of the condition number are either based on reduced QR decomposition or on singular value decomposition (SVD).
A **method for least squares via QR decomposition** usually uses a modified Gram-Schmidt (MGS) [15, p. 255] or Householder [15, p. 249] orthogonalization in order to compute a reduced QR factorization of $A \in \mathbb{R}^{m\times n}$ $(m \geq n)$ with full rank n. That is

$$A = QR = \Big[Q_1, Q_2\Big] \begin{bmatrix} R_1 \\ 0 \end{bmatrix} = Q_1 R_1, \tag{3.11}$$

where $R \in \mathbb{R}^{m\times n}$ is a matrix whose submatrix $R_1 \in \mathbb{R}^{n\times n}$ is upper triangular with positive diagonal elements, and $Q \in \mathbb{R}^{m\times m}$ is an orthogonal matrix which has been partitioned into submatrices $Q_1 \in \mathbb{R}^{m\times n}$ and $Q_2 \in \mathbb{R}^{m\times(n-m)}$ containing the first n and the last $m-n$ columns of Q, respectively. Once these factors have been computed, relationship (3.11) can already be used to solve the LLS problem (3.8). We find that

$$\big\|Ax - b\big\|_2^2 = \big\|QRx - b\big\|_2^2 = \big\|Rx - Q^\mathsf{T}b\big\|_2^2 = \big\|R_1x - {Q_1}^\mathsf{T}b\big\|_2^2 + \big\|{Q_2}^\mathsf{T}b\big\|_2^2,$$

which is minimal when $x = R_1^{-1}{Q_1}^\mathsf{T}b$. By Theorem 3.1, this minimizer is unique and algorithms can simply acquire it by computing ${Q_1}^\mathsf{T}b$ and by back substitution.

The associated cost is dominated by the computation of the QR decomposition. It is $\mathcal{O}(2mn)$ for MGS [15, p. 255] and $\mathcal{O}\big(2mn^2 - \frac{2}{3}n^3\big)$ if Householder reflections are used [15, p. 264]. Compared to these efforts, the $\mathcal{O}(mn)$ operations of the matrix-vector product and the $\mathcal{O}(n^2)$ operations need for the back substitution are not significant. In the case of Householder transformations, we find that the relative error in the computed solution is proportional to the condition number of A [49, p. 199]. A similarly good result can be achieved for MGS if it is applied to the augmented matrix $[A \mid b]$, see [15, p. 265]. Hence, methods for least squares via QR decomposition do not degrade the conditioning of the problem and are generally numerically stable. Nevertheless, if A is close to being *rank-deficient*[6], algorithms may break down during their back substitution phase. The big disadvantage of these methods clearly is their speed. If $m \gg n$, about twice as much arithmetic operations as in the normal equations approach are needed. Furthermore, in most scenarios more memory is required since $Q_1 \in R^{m\times n}$ needs more storage than the compressed matrix $A^\mathsf{T}A$, which is only of size $n \times n$. So, basically, we face a trade-off between numerical stability and efficiency in the choice of a proper algorithm for LLS.
Another approach for solving the LLS problem (3.8) is based on the singular value decomposition (SVD) of A, which is also applicable in the rank-deficient case and thus typically applied when we do not know the rank of the matrix. The basic idea behind such solvers can be found in [8, p. 64–66] and an appropriate algorithm is given in [49, p. 83–84]. A typical estimate of the cost associated with the SVD is $\mathcal{O}(2mn^2 + 11n^3)$. Hence, *methods for least squares via SVD* are generally more expensive than those that have been discussed.

Remark. The backslash operator \ can also be used to directly compute a solution of a LLS problem of the form (3.8) by simply entering $\mathbf{A}\backslash\mathbf{b}$. If A is rectangular, MATLAB invokes a *QR-Solver* [29, p. 1133–1134] which is based on QR decomposition with column pivoting. Such a solver can handle rank-deficient matrices as well but can be quite costly, see [15, p. 276–280].

Nonlinear least squares problems have no closed form solution and are generally solved by iterative refinement. Often, linear least squares problems arise as subproblems in each iteration, i.e., they are solved in order to approximate the nonlinear system. In fact, all of the algorithms for NLLS discussed in Section 4 utilize LLS at some point in their iteration process.
Least squares problems may be the largest source of unconstrained optimization problems. The main application of NLLS is data fitting, which is described in the following section. LLS methods are not only powerful tools for solving NLLS problems but are also quite impressive by themselves. They are frequently used as they are a standard approach (alongside maximum likelihood estimation) in linear regression, see Section 3.4.1.

[6] A matrix is said to be *rank-deficient* if it does not have full rank.

3.3 Data Fitting

Least squares problems occur in *data fitting*, where we are attempting to fit model functions to data. A typically fuzzy set is given by data (t_i, y_i), $i = 1, \dots m$, where the data values y_i have been sampled for t_i of some independent variable t. The *observations* y_i may be subject to experimental error. We want to describe and idealize the data set by a smooth curve with few parameters. Hence, it is desired to choose a nonlinear model function $\phi(t, x)$ with n adjustable parameters x which best fits the data. Usually, n is much smaller than the number of data points m. If this were not the case, an arbitrary model could give a close but not necessarily good fit. The most intuitive fitting is based on minimizing the sum of squares of the residuals $f_i(x) := y_i - \phi(t_i, x)$, $i = 1, \dots, m$. That is, a least squares solution is sought in the sense of minimizing $f(x)$ in (3.3). The model function ϕ is valid if x^* can be found such that $f(x^*)$ is small.
So, the $x_i,\ i = 1, \dots n$, are interpreted as parameters that need to be manipulated in order to adjust ϕ to the data. We note that the nonlinearity in $\phi(t, x)$ refers to nonlinearity in x. The above optimization problem therefore basically is a *nonlinear parameter estimation* problem.
In practice, ϕ is a smooth function that can be formed from common basic functions. Popular choices include low-degree polynomials, exponential functions, trigonometric functions and linear functions.

Remark. The Euclidean norm $\|\cdot\|_2$ as measure of the discrepancy between the model and the observed data is justified by statistical considerations [2]. However, it may happen that outliers in data may dominate the whole optimization problem. In such cases, methods for nonlinear least squares become inefficient since they are based on the premise that the residuals are small near the solution, see (3.6). There are various other reasonable choices as how to minimize the $f_i(x)$. Most notable are the applications of the 1-norm $\|\cdot\|_1$ and the maximum norm $\|\cdot\|_\infty$, resulting in nonsmooth *least absolute value* and *least maximum value (minimax)* problems, respectively [14, p. 96–97]. These problems are not within the scope of this thesis.

The discussed *least squares fitting* is perhaps the most used technique by data analysts, engineers and scientists to fit a function to data without any assumptions about probability distributions. Because of its interdisciplinary popularity, one may encounter different notations for the same problem. Many data fitting problems are stated in the convention used by statisticians, with n being the sample size and p the number of parameters. That is, we want to find a *parameter vector* $\theta \in \mathbb{R}^p$, $p \ll n$, such that the *residual sum of squares* (RSS)

$$Q(\theta) := \sum_{i=1}^{n} \big(y_i - f_i(\theta)\big)^2 \tag{3.12}$$

is minimal. We denote the n given data points by $(x_i, y_i),\ x_i \in \mathbb{R}^m,\ y_i \in \mathbb{R},\ m \geq 1$, where the x_i and the y_i are components of the *explanatory vector* x and the *observed data vector* y, respectively. The model $f : \mathbb{R}^p \to \mathbb{R}^n$ is called *response function* and we write $f_i(\theta) = f(x_i, \theta)$ for the i-th component of the vector $f(x, \theta)$. Once f is chosen, the residuals $r_i := y_i - f_i(\theta)$ are functions of θ only.
In data analysis, *regression models* are often used to express the relationship between *dependent variables* and *independent variables*.

3.4 Regression

In many applications, some subset of variables may be characterized as dependent on some other subset of independent variables. Often, a complete data set is given by a single data vector $y \in \mathbb{R}^n$ and we seek to understand its dependence on the other variables. In regression, *dependence* refers to a stochastic relationship, i.e., we assume that at least one of the variables $y_i \in \mathbb{R}$ is random, being subject to unexplained fluctuations or measurement error. The independent variables $x_i \in \mathbb{R}^m,\ m \geq 1$, usually called *regressors*, *features*, *predictors* or *explanatory variables*, are used to explain or predict the behavior of the dependent variable y_i, also known as *response* or *outcome variable*. A *regression model* aims to express this relationship by some function F, i.e.,

$$y_i \approx F(x_i) \tag{3.13}$$

with desirable small *data error*, also referred to as *noise*,

$$\varepsilon_i := y_i - F(x_i).$$

The explanatory variables x_i can be random or fixed, e.g., controlled by the experimenter. Hence, using the *regression function* F on the x_i, the response y_i can then be predicted or measured. The task in regression is to find a function F for which (3.13) holds as closely as possible. Sometimes, the mathematical form of the functional relation is known, except for some unknown parameters. Then (3.13) can be written as

$$y_i = F(x_i, \theta) + \varepsilon_i,$$

where F is entirely known but the parameters $\theta = (\theta_1, \theta_2, \ldots, \theta_p)^\mathsf{T},\ p \leq n$, are unknown and need to be estimated. For n data points (x_i, y_i) we thus find the model

$$y = F(X, \theta) + \varepsilon, \tag{3.14}$$

where $y \in \mathbb{R}^n$, $X \in \mathbb{R}^{n \times m}$ is a *design matrix*[7] of n predictors $x_i \in \mathbb{R}^m$, $\theta \in \mathbb{R}^p$ and $\varepsilon \in \mathbb{R}^n$ represents the error term consisting of n data errors $\varepsilon_i \in \mathbb{R}$. The errors ε_i are random variables which may or may not be uncorrelated and normally distributed with zero mean and constant variance. If the regression function F is nonlinear in the parameters θ, relationship (3.14) is called *nonlinear regression model*.
Typically, we have little or no idea about the underlying data generating process; but we always assume that a *true* relationship in the sense of (3.14) exists, i.e., that y can be perfectly fitted by some regression function F with a *true parameter* vector θ^*. Thus, finding a proper regression model often reduces to estimating the parameters.

In general, there exist no explicit formulas for the computation of the unknown θ in (3.14), i.e., usually iterative procedures are required. Least squares fitting is perhaps the most popular method for the estimation of these parameters. That is, the RSS (3.12) is minimized subject to θ, with f being the known regression function F or a fitting model. The minimizer $\hat{\theta}$, i.e., the estimated parameter vector which gives the best fit to the data, is known as *least squares estimator*.

[7] A design matrix X is defined such that X_{ij} represents the value of the j-th variable associated with the i-th object.

Remark. Suppose that we want to fit data according to the regression model (3.13), i.e.,

$$y_i = F(x_i) + \varepsilon_i.$$

The choice of the model function f for least squares fitting depends on the observed data and the precise goal of the experimenter. Often, f is an approximation to, or equal to, the regression function F; but it may also be chosen arbitrarily to allow for different approaches or numerically stable computations. If we take a closer look at the residuals for a given function f, we find for $i = 1, \ldots, n$ that

$$\begin{aligned} r_i &= y_i - f(x_i, \theta) \\ &= \big(y_i - F(x_i)\big) + \big(F(x_i) - f(x_i, \theta)\big) \\ &= \varepsilon_i + \big(F(x_i) - f(x_i, \theta)\big). \end{aligned}$$

The *approximation error* $F(x_i) - f(x_i, \theta)$ represents the discrepancy between F and f at x_i. Hence, $f(x, \theta)$ is considered to be a good fitting model if the approximation errors and the noise terms ε_i are about the same size.
Furthermore, it is important to note that even if $F(x)$ and $f(x, \theta)$ have the same form, there is no guarantee that the estimated parameters θ used in $f(x, \theta)$ will be identical to those underlying the regression function $F(x)$ [19, p. 5].

The error term in regression models is often assumed to be normally distributed because many physical phenomena follow a normal distribution. A constant variance implies a constant spread of errors; whereas uncorrelated ε_i with zero mean ensure that the generated noise is purely random. Least squares fitting generally works well with data following a normal distribution. This is because the occurrence of large noise terms is rather exceptional and, as we have seen on page 34, the strength of methods for least squares are problems with small residuals.

3.4.1 Linear regression

In *linear regression*, the relationship between the variables is expressed as

$$y_i \approx \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip}, \tag{3.15}$$

where now $\theta = (\beta_1, \beta_2, \ldots, \beta_p)^\mathsf{T}$ and x_{ij} denotes the j-th independent variable of the i-th observation $x_i \in \mathbb{R}^p$, i.e., $m = p$. Here, the important requirement is linearity in the parameters, not the regressors. That is, the x_i might include powers or transformations of the original measurements, e.g., $x_{ij} = z^2$ or $x_{ij} = \log(z)$. If $x_{i1} = 1$, $i = 1, \ldots, n$, then we call the corresponding coefficient β_1 the *regression intercept*.
For a given set of n observations (x_i, y_i), a *linear regression model* can be written as

$$y = X\beta + \varepsilon, \tag{3.16}$$

where $y \in \mathbb{R}^n$, $X \in \mathbb{R}^{n \times p}$ is a design matrix of n predictors $x_i \in \mathbb{R}^p$, $\beta \in \mathbb{R}^p$ is the vector of coefficients $\beta_1, \ldots \beta_p$ and $\varepsilon \in \mathbb{R}^n$ is a random vector representing the noise.

The most familiar linear regression model assumes that $n > p$ and that ε is uncorrelated and normally distributed with zero mean and constant variance.
Estimating the unknown coefficients β of (3.16) by the use of a least squares fitting is widely known as *ordinary least squares* (OLS). The problem takes the form

$$\underset{\beta \in \mathbb{R}^p}{\text{minimize}} \quad \left\|y - X\beta\right\|_2^2, \tag{3.17}$$

cf. (3.12). The minimizer $\hat{\beta}$ of (3.17) is then called *OLS estimator* for β.
If $n > p$, we can express the linear relationship (3.15) for a complete data set as the overdetermined system of equations

$$X\beta \approx y.$$

The corresponding standard LLS problem (see Section 3.2.1) is equivalent to the least squares fit (3.17). Hence, any OLS estimator must satisfy the normal equations

$$X^\mathsf{T} X\beta = X^\mathsf{T} y. \tag{3.18}$$

Furthermore, if X has full rank, (3.18) may be solved directly by a method for normal equations. Because then, according to Theorem 3.1, the matrix $X^\mathsf{T} X$ is invertible and the unique OLS estimator $\hat{\beta}$ for β can be computed as

$$\hat{\beta} = (X^\mathsf{T} X)^{-1} X^\mathsf{T} y. \tag{3.19}$$

Remark. The ordinary least squares method is a special case of the *generalized least squares* (GLS) method, which can be applied to regression models with a certain degree of correlation between distinct errors. GLS and *maximum likelihood estimation* (MLE) are well explained in [47, p. 27–42].

Remark. The *Gauss-Markov Theorem*, e.g., stated and proven in [17, p. 115–116], theoretically constitutes the need for least squares fitting in linear regression. If the errors in (3.16) are uncorrelated with zero mean and constant variance and if the matrix X is of full rank, it says that the OLS estimator (3.19) defined by the normal equations is the *best linear unbiased estimator* (BLUE). Here, "best" is in the sense of minimum variance of linear combinations $a^\mathsf{T}\hat{\beta}$, "linear" refers to the form of the estimator $\hat{\beta} = Ay$ and "unbiased" means that the expectation of the estimator is that of the quantity to be estimated, i.e., $\mathbb{E}(\hat{\beta}) = \beta$.[8] However, we point out that under the same assumptions there exist *biased* estimators with smaller variance.

In this section, the importance of least square methods in nonlinear optimization has been depicted, theoretical background has been given and a very prominent field of application, data fitting, has been described. We will now proceed to the main part of this thesis, where we study particular derivative-free techniques for solving NLLS problems, i.e., for finding the roots of overdetermined (and well-determined) systems of nonlinear equations.

[8] $\mathbb{E}(X)$ denotes the expected value of the random variable X.

This page intentionally left blank.

4 Algorithms

Two of the most popular techniques for unconstrained optimization of an objective function in the nonlinear least squares sense (3.3) are the **Gauss-Newton method** and the **Levenberg-Marquardt algorithm**. In this section, we present derivative-free versions of these gradient-methods. The so called **finite difference analogues** of the **Gauss-Newton method** and the **Levenberg-Marquardt algorithm** [6] omit the computation of derivatives by using finite difference approximations instead. Furthermore, we discuss a derivative-free procedure which does not even apply a numerical approximation procedure for the derivatives. The **DUD algorithm** [44] is a secant method that was specifically designed for data fitting applications where the evaluation of the response function is rather expensive; e.g., for cases where the response is determined by numerical integration over a defining system. DUD (for Doesn't Use Derivatives) can be understood as Gauss-Newton-like method that makes efficient use of function evaluations.

This section is basically structured as follows:

- In each subsection, a method is introduced.
- First, the respective procedure is described.
- Then, additional information and mathematical theory is provided.
- Lastly, at the end of each subsection, implementation details are given.

The demonstrated algorithms are followed by *pseudocodes*, which essentially summarize the key principles and omit programming language specific details. We note that encountered loops may be replaced by more efficient techniques (e.g., vectorization) in an actual computer program. In Appendix C, one can find full length MATLAB implementations which relate to the depicted pseudocodes. Numerical results for the considered algorithms are presented in Section 5. The literature sources for this part of the work are [2, 6, 8, 9, 14, 22, 26, 40, 44, 46, 47, 50].

4.1 Gauss-Newton Method

We consider the NLLS problem (3.3) and suppose that the components of F are twice continuously differentiable functions. That is, we want to find $x \in \mathbb{R}^n$ such that

$$f(x) = \frac{1}{2} \left\| F(x) \right\|_2^2 \tag{4.1}$$

is minimal. The classical Newton method for unconstrained optimization (discussed in Section 2 on page 19) seeks such a least squares solution along the Newton direction (2.14) in each step. Hence, at iteration l of the algorithm, the point

$$x^{l+1} = x^l - B_l^{-1} g^l \tag{4.2}$$

is computed, where $B_l = \nabla^2 f(x^l)$ is the Hessian and $g^l = \nabla f(x^l)$ the gradient at x^l. The particular form of f allows to rewrite (4.2) in terms of (3.5) and (3.6) as

$$x^{l+1} = x^l - \left[J(x^l)^\mathsf{T} J(x^l) + \sum_{i=1}^{m} f_i(x^l) \nabla^2 f_i(x^l) \right]^{-1} J(x^l)^\mathsf{T} F(x^l), \tag{4.3}$$

where J denotes the Jacobian of F. We obtain the *Gauss-Newton* (GN) method by simply ignoring the summation term in (4.3). That is, the Gauss-Newton step

$$x^{l+1} = x^l - \left[J(x^l)^\mathsf{T} J(x^l) \right]^{-1} J(x^l)^\mathsf{T} F(x^l) \tag{4.4}$$

does not contain any second order derivatives. The GN algorithm is therefore a modification of the classical Newton method which replaces the Hessian in (4.2) with $B_l = J_l{}^\mathsf{T} J_l$, where $J_l = J(x^l)$.
The GN method is the prime example of NLLS optimization algorithms as it takes direct advantage of the special structure of the problems, as depicted in (3.7). That is, the exorbitant cost of providing the full Hessian of f is avoided since $J^\mathsf{T} J$ is considered to be a good approximation whenever the components of F, namely f_i, $i = 1, \ldots, m$, are small or close to being linear. In the zero residual case, the GN algorithm can keep up with the quadratic convergence of Newton's method, see Section 4.1.2. Hence, GN is just as good as Newton's method near a solution of (3.3). But it has the big advantage that it needs not to calculate second derivatives, which results in less function evaluations and saves computational cost. So, the GN method performs exceptionally well in most data fitting applications, where the residuals are usually small and where function evaluation is often rather expensive.
However, the procedure has major drawbacks in efficiency and robustness for problems with large residuals, rank-deficient or ill-conditioned design matrices; or for bad choices of starting points. In general, we must expect linear convergence for GN, see Section 4.1.2. Global convergence can only be proven under a uniform full rank assumption on the Jacobian, meaning that J_l has full rank for every l in the region of interest, see [40, p. 256]. So, if a program starts with a bad initial approximation, it will be very slow and may not find a solution at all. If the residuals are large, the Jacobian term no longer dominates the second order term in (3.6), i.e., the procedure loses a big amount of derivative information and will thus likely fail to converge.

A more natural way of deriving the *GN* method is based on Taylor Series expansion. That is, we can find a tangent hyperplane approximation of the function $F(x)$ at the iteration point x^l that has the form

$$F(x) \approx F(x^l) + J(x^l)p, \tag{4.5}$$

where $p := x - x^l$. Hence, minimizing (4.1) corresponds to solving the LLS problem

$$\underset{p \in \mathbb{R}^n}{\text{minimize}} \quad \frac{1}{2} \left\| J(x^l)p + F(x^l) \right\|_2^2. \tag{4.6}$$

This problem can be solved explicitly and its solution p^l is given by the normal equations (cf. (3.9))

$$J(x^l)^\mathsf{T} J(x^l)p^l = -J(x^l)^\mathsf{T} F(x^l), \tag{4.7}$$

i.e.,

$$p^l = -\Big[J(x^l)^\mathsf{T} J(x^l)\Big]^{-1} J(x^l)^\mathsf{T} F(x^l). \tag{4.8}$$

Thus, the minimizer p^l on the plane corresponds to the Gauss-Newton search direction. Solving the GN step (4.4) is thereby the same as solving a LLS problem, so applying the GN method is in effect applying a sequence of LLS fits to a nonlinear function. If we assume that $J_l = J(x^l)$ has full rank, ${J_l}^\mathsf{T} J_l$ is positive definite and, according to Theorem 3.1, the GN direction p^l is unique. Moreover, unless $f(x^l)$ is a stationary point, it is a descent direction (see (2.9)). This immediately follows from the positive definiteness of the inverse of ${J_l}^\mathsf{T} J_l$, as we find for $g_l = \nabla f(x^l)$ and $F^l = F(x^l)$ that

$$ {g_l}^\mathsf{T} p^l = \big({J_l}^\mathsf{T} F^l\big)^\mathsf{T} p^l = -\underbrace{\big({J_l}^\mathsf{T} F^l\big)^\mathsf{T} \big({J_l}^\mathsf{T} J_l\big)^{-1} {J_l}^\mathsf{T} F^l}_{>\,0} < 0. \tag{4.9}$$

Otherwise, if J_l does not have full rank, the GN step (4.4) is undefined and condition (4.9) needs not hold. So, the sum of squares of the residuals (4.1) may not decrease at every iteration. Furthermore, the solution of the normal equations (4.7) is not unique and its computation is expensive as we cannot apply the standard solvers, namely the method for normal equations or the method for least squares via QR decomposition, discussed in Section 3.2.1.
If J_l is ill-conditioned, there are numerical problems in solving (4.6). Typically, the matrix ${J_l}^\mathsf{T} J_l$ has then very small eigenvalues. This may lead to extremely slow convergence [47, p. 622–623] or even local divergence, as the strict lower bound criteria for the spectrum in Theorem 4.4 of Section 4.2.2 cannot be met.
So, if rank-deficiency or ill-conditioning of the Jacobian is to be expected, one should aim for a different optimization method; for example, the Levenberg-Marquardt algorithm from Section 4.2, which is basically a more robust version of GN.

Remark. We notice that the standard GN method uses as stepsize $\alpha = 1$. This simplest form of GN works well for a large class of problems. However, as in Newton methods, the *damped Gauss-Newton* algorithm uses an efficient line search procedure which computes a stepsize parameter $0 < \alpha < 1$ in each step. Typically, α is chosen such that it satisfies the sufficient descent condition (2.10). In fact, many modifications of GN exist, most of which use different ways to compute α or aim to improve the accuracy of the approximated Hessian.

In practice, typically only the Jacobian $J(x^0)$ of the initial point x^0 is known. Analytic derivatives are unavailable and implementations of the Gauss-Newton method require a procedure for the computation of $J(x)$ in step 4 of the following example Algorithm 1.

Algorithm 1 Gauss-Newton (GN)

Input: function $F(x)$, Jacobian $J(x)$ of $F(x)$, initial approximation x^0, termination parameters tol, $maxit$
Output: approximate solution of NLLS problem (3.3)
Require: $maxit \in \mathbb{N}$, $0 < tol \ll 1$

1: initialize $x = x^0$;
2: **for** $k = 1$ to $maxit$ **do**
3: evaluate $F = F(x)$;
4: compute $J = J(x)$;
5: solve $J^\mathsf{T} Jp = -J^\mathsf{T} F$ for p; ▷ normal equations of $\min_p \left\| Jp + F \right\|_2^2$
6: **if** $\|p\| < tol$ **then**
7: **break**; ▷ stop iteration if the stepsize is too small
8: **end if**
9: set $x = x + p$;
10: **end for**
11: **return** x;

4.1.1 Finite difference analogue

The *finite difference analogue of the Gauss-Newton method* (FDGN) by Brown and Dennis [6] uses finite differences to approximate the Jacobian in each iteration. More precisely, it approximates $J(x)$ by the corresponding matrix of difference quotients

$$\frac{\Delta F(x, h)}{h}, \tag{4.10}$$

where $h \in \mathbb{R}$ and $\Delta F(x, h)$ denotes the matrix whose ij-th component is given by

$$f_i\left(x + he_j\right) - f_i(x),$$

and e_j is the vector with unity in the j-th position and zeros elsewhere. That is, equation (4.4) transforms into the FDGN step

$$x^{l+1} = x^l - \left[\frac{\Delta F(x^l, h_l)^\mathsf{T} \Delta F(x^l, h_l)}{h_l^2}\right]^{-1} \frac{\Delta F(x^l, h_l)^\mathsf{T}}{h_l} \cdot F(x^l), \tag{4.11}$$

where $h_l \in \mathbb{R}$ denotes the increment in iteration l. If $h_l^{-1} \Delta F(x^l, h_l)$ has full rank and unless $f(x^l)$ is a stationary point, step (4.11) is heading in a descent direction, cf. (4.9). We will see that the employment of finite differences does not jeopardize the commonly known convergence properties held by GN. The GN step (4.4) can be considered as the special case of the FDGN step (4.11) with $h_l = 0$, since it holds that

$$\lim_{h_l \to 0} \frac{\Delta F(x^l, h_l)}{h_l} = J(x^l). \tag{4.12}$$

Remark. (Forward differences) In Section 2.4.4, we have mentioned that it is a common strategy to replace the gradient of a scalar function with forward differences (2.36), yielding a truncation error that is $\mathcal{O}(h)$. So, since $F : \mathbb{R}^n \to \mathbb{R}^m$ is vector-valued, it is reasonable to use the same idea to approximate the ij-th element J_{ij} of $J = J(x)$ by

$$A_{ij} = \frac{f_i(x + he_j) - f_i(x)}{h},$$

or equivalently, to approximate the j-th column $J_{:j}$ by

$$A_{:j} = \frac{F(x + he_j) - F(x)}{h}. \tag{4.13}$$

We can still expect a similar approximation error, as it is proven in [8, p. 78] that

$$\left\|A_{:j} - J_{:j}\right\| = \mathcal{O}(h)$$

for sufficiently small h and an arbitrary vector norm. This ultimately gives the error

$$\left\|A - J\right\|_1 = \mathcal{O}(h),$$

where $A := h^{-1}\Delta F(x, h)$. This equality holds because the matrix 1-norm is the maximum of the 1-norms of the columns of $A - J$.
Forward differences are prone to cancellation and rounding errors. Whereas rounding errors are generally negligible [14, p. 128], cancellation errors need to be examined closer. Hence, let us denote the computed function values of F by $\hat{F}$ and assume that

$$\hat{F}(x) = F(x) + \epsilon \quad \text{and} \quad \hat{F}(x + he_j) = F(x + he_j) + \epsilon_h,$$

where ϵ and ϵ_h are the absolute errors in F at x and $x + he_j$, respectively. So, if the inexact values are used in the difference quotient (4.13), we find

$$\frac{\hat{F}(x) - \hat{F}(x + he_j)}{h} = A_{:j} + \frac{\epsilon - \epsilon_h}{h},$$

where $(\epsilon - \epsilon_h)/h$ describes the cancellation error. Therefore, decreasing h will reduce the truncation error but, unfortunately, will increase the cancellation error.
Furthermore, we notice that we require an additional evaluation of F in (4.13). That is, we need n additional function evaluations for the approximation of the Jacobian.

The FDGN copes with the discrepancy concerning the truncation and cancellation errors by decreasing the increment just enough so that the truncation error is reduced to an acceptable level. In fact, the convergence theory for the FDGN (Theorem 4.6 in Section 4.2.2) dictates the choice for the finite-difference intervals in the iteration process. More precisely, h_l should be chosen as $\mathcal{O}\big(\|F(x^l)\|\big)$ in iteration l, where $\|\cdot\|$ denotes an arbitrary vector norm.

Algorithm 2 replaces the exact computation of the Jacobian $J(x)$ in Algorithm 1 (GN) with a forward difference approximation, whose implementation is explained in Section 4.1.3.

Algorithm 2 Finite difference analogue of Gauss-Newton (FDGN)

Input: function $F(x)$, initial approximation x_0, termination parameters tol, $maxit$
Output: approximate solution of NLLS problem (3.3)
Require: $maxit \in \mathbb{N}$, $0 < tol \ll 1$
Ensure: h is sufficiently small in each iteration - ideally $|h|$ is $\mathcal{O}\big(\|F(x)\|\big)$ but δ is used as safeguard to prevent absurd choices of h relative to the iterate x

initialize $x = x_0$;
$n \leftarrow$ dim. of x;
$m \leftarrow$ dim. of $F(x)$;
$h,\ \delta \leftarrow n$-dim. zero vector;
$J \leftarrow m \times n$ zero matrix;
for $k = 1$ to $maxit$ **do**
    evaluate $F = F(x)$;
    ▷ finite difference approximation $J = J(x)$ of the Jacobian of $F(x)$
    for $i = 1$ to m **do**
        for $j = 1$ to n **do**
            if $|x_j| < 10 \cdot \sqrt{eps}$ **then**
                $\delta_j = 10^{-2} \cdot \sqrt{eps}$;
            else
                $\delta_j = 10^{-3} \cdot |x_j|$;
            end if
            $h_j = \min\big(\|F\|,\ \delta_j\big)$;
            $e_j \leftarrow$ the j-th unit vector of dim. n;
            compute the forward difference approximation

$$J_{ij} = \frac{F_i(x + h_j e_j) - F_i(x)}{h_j};$$

        end for
    end for
    solve $J^\mathsf{T} J p = -J^\mathsf{T} F$ for p; ▷ normal equations of $\min_p \big\|Jp + F\big\|_2^2$
    if $\|p\| < tol$ **then**
        break; ▷ stop iteration if the stepsize is too small
    end if
    set $x = x + p$;
end for
return x;

4.1.2 Convergence analysis

The convergence analysis for the GN method and the FDGN is contained within the theory of the Levenberg-Marquardt algorithm in Section 4.2.2. In particular, Theorems 4.4 and 4.6 hold also for the FDGN, whereas Corollaries 4.5 and 4.7 correspond to GN. In short, it is shown that both converge locally with order at least one. And whenever

$$|h_l| = \mathcal{O}\big(\|F(x^l)\|\big) \text{ and } \big\|F(x^l)\big\| \to 0 \text{ as } l \to \infty,$$

the FDGN is quadratically convergent. And, since the GN step is the special case of the FDGN step with $h_l = 0$ for every l, the GN method is quadratically convergent whenever

$$\big\|F(x^l)\big\| \to 0 \text{ as } l \to \infty.$$

A global convergence statement cannot be given, as both methods cannot guarantee that all steps are taken in a descent direction, see page 45.

4.1.3 Implementation details

According to Theorem 4.6 on page 62, the increment h_l should be chosen as $\mathcal{O}\big(\|F(x^l)\|\big)$ in each iteration l. However, this choice must be regulated in the actual implementation of the FDGN algorithm to prevent absurd values of h_l relative to x^l; e.g., suppose that $\|x^0\| = 0.001$ and $\|F(x^0)\| = 1000$. Therefore, we need to implement the ij-th component of the matrix of difference quotients for step (4.11) as

$$\frac{f_i\big(x^l + (h_l)_j e_j\big) - f_i(x^l)}{(h_l)_j},$$

where now $h_l \in \mathbb{R}^n$. This enables us to adjust the incremental change for each column of the matrix. We let $\delta^l \in \mathbb{R}^n$ and determine the j-th entry of h_l by

$$(h_l)_j = \min\big(\|F(x^l)\|, \delta_j^l\big), \tag{4.14}$$

where

$$\delta_j^l = \begin{cases} 10^{-2} \cdot \sqrt{eps} & \text{if } \big|x_j^l\big| < 10 \cdot \sqrt{eps}, \\ 10^{-3} \cdot \big|x_j^l\big| & \text{if } \big|x_j^l\big| \geq 10 \cdot \sqrt{eps}. \end{cases} \tag{4.15}$$

This definition of the $(h_l)_j$ guarantees that the conditions of Theorem 4.6 will be met as the algorithm approaches $F(x^*) = 0$. Although the theorem states that $\|\cdot\|$ in (4.14) can be taken as an arbitrary norm, we found that FDGN performs best for the 2-norm. The number *eps* denotes the machine epsilon which in our case, using double precision arithmetic and MATLAB, corresponds to

$$eps = 2^{-52} \approx 2.22 \cdot 10^{-16}.$$

The value $\sqrt{eps}$ has been found to be the optimal compromise for the truncation and cancellation error dilemma in forward differences, see [43, p. 230].

Therefore, (4.15) is just a slight, but noticeable modification of the initially proposed rule by Brown and Dennis [6], who used (also for a double precision format)

$$\delta_j^l = \begin{cases} 10^{-9} & \text{if } \left|x_j^l\right| < 10^{-6}, \\ 10^{-3} \cdot \left|x_j^l\right| & \text{if } \left|x_j^l\right| \geq 10^{-6}. \end{cases}$$

MATLAB offers a convenient way for solving the normal equations in step 5 of Algorithm 1 and step 20 of Algorithm 2. As discussed in Section 3.2.1, the backslash operator \ invokes an efficient Cholesky or LDL solver. That is, $\mathcal{O}(mn^2 + \frac{n^3}{3})$ operations are needed in order to solve the LLS problem. If $J^\mathsf{T}J$ does not have full rank, the computation is performed regardless. However, a warning message is displayed when the matrix is ill-conditioned or nearly singular. If that is the case and if the procedure has not yet converged, it will most likely fail to do so.
The total cost of the GN method and the FDGN is usually dominated by the effort spent on the calculation of the Jacobian and the finite difference approximation, respectively. Computing the forward differences requires $\mathcal{O}(mn)$ operations but also $n + 1$ evaluations of F, which are typically quite expensive. Both algorithms demand a user-supplied analytic Jacobian as input. But one would usually invoke a backward-mode automatic differentiation procedure which needs $\mathcal{O}(1)$ evaluations of F for computing J.
We found the value 10^{-5} to be satisfying for the step tolerance tol, as taking smaller steps typically only results in marginal gain in accuracy for an immoderate expense. Also, it is beneficial to implement an additional stopping criterion that terminates the algorithm when a sufficiently good minimization (specified by the user) has been achieved. Other practical ways to stop the methods include predefined limits for the number of function evaluations and the algorithmic time.
The actual code of the MATLAB programs can be found as C.1 (GN) and C.2 (FDGN) in Appendix C. In C.2, we got rid of the outer for-loop for the finite difference approximation (see step 8 of Algorithm 2) by using vectorization. Both programs terminate when the objective function is minimized or some user-defined maximum number of function evaluations is reached.

4.2 Levenberg-Marquardt Algorithm

The Gauss-Newton method and its finite difference analogue do not always converge reliably. Often, problems are encountered when the second-order term in (3.6) is significant or when the Jacobian or its finite difference approximation is rank deficient. A method that eliminates these flaws is the *Levenberg-Marquardt* (LM) algorithm. The LM step is basically a regularized version of the GN step (4.4). It has the form

$$x^{l+1} = x^l - \left[\mu_l I + J(x^l)^\mathsf{T} J(x^l)\right]^{-1} J(x^l)^\mathsf{T} F(x^l), \tag{4.16}$$

where $\mu_l \in \mathbb{R}$ is nonnegative and commonly known as *damping parameter* in the literature. The idea of adding a regularization term to the diagonal of ${J_l}^\mathsf{T} J_l$ was initially proposed by Levenberg [27] and was later rediscovered by Marquardt [28]. The diagonal matrix $\mu_l I$ can be perceived as compensation for the lost information

$$\sum_{i=1}^{m} f_i(x^l) \nabla^2 f_i(x^l) \tag{4.17}$$

in the Hessian approximation of the GN method, cf. (3.6) and (3.7). That is, the LM algorithm is superior to GN when the residual $F(x^l)$ is large and not close to being linear, because we can account for the omitted part (4.17) of the Hessian $\nabla^2 f(x^l)$. Furthermore, since ${J_l}^\mathsf{T} J_l$ is at least positive semidefinite, the damping term $\mu_l I$ causes the eigenvalues of $\mu_l I + {J_l}^\mathsf{T} J_l$ to be at least μ_l. So, whenever $\mu_l > 0$, the matrix $\mu_l I + {J_l}^\mathsf{T} J_l$ is positive definite and thus nonsingular. Hence, increasing the damping parameter adds regularity to the matrix, making it arbitrarily well-conditioned for sufficiently large μ_l. Accordingly, the LM algorithm can handle both rank-deficient and ill-conditioned ${J_l}^\mathsf{T} J_l$, i.e., it is more robust than GN.
Performing the LM step (4.16) in iteration l of an algorithm is equivalent to solving

$$\left[\mu_l I + J(x^l)^\mathsf{T} J(x^l)\right] p = -J(x^l)^\mathsf{T} F(x^l)$$

for $p := x^{l+1} - x^l$. It is easy to see that these are the normal equations of the LLS problem

$$\underset{p \in \mathbb{R}^n}{\text{minimize}} \quad \frac{1}{2} \left\| \begin{bmatrix} \sqrt{\mu_l} I \\ J(x^l) \end{bmatrix} p + \begin{bmatrix} 0 \\ F(x^l) \end{bmatrix} \right\|_2^2. \tag{4.18}$$

If $\mu_l = 0$, step (4.16) is simply the GN step and we refer to the discussion on page 45. If $\mu_l > 0$, the matrix $\mu_l I + {J_l}^\mathsf{T} J_l$ has full rank and, according to Theorem 3.1, problem (4.18) has the unique solution

$$p^l = -\left[\mu_l I + J(x^l)^\mathsf{T} J(x^l)\right]^{-1} J(x^l)^\mathsf{T} F(x^l). \tag{4.19}$$

Unless x^l is a stationary point, p^l is a descent direction. Due to the positive definiteness of the inverse of $\mu_l I + {J_l}^\mathsf{T} J_l$, we have for $g_l = \nabla f(x^l)$ and $F^l = F(x^l)$ that

$$g_l{}^\mathsf{T} p^l = \left({J_l}^\mathsf{T} F^l\right)^\mathsf{T} p^l = -\underbrace{\left({J_l}^\mathsf{T} F^l\right)^\mathsf{T} \left(\mu_l I + {J_l}^\mathsf{T} J_l\right)^{-1} {J_l}^\mathsf{T} F^l}_{>\,0} < 0. \tag{4.20}$$

The relationship (4.19) can also be expressed as

$$\left[I + \frac{1}{\mu_l} J(x^l)^\mathsf{T} J(x^l)\right] p^l = -\frac{1}{\mu_l} J(x^l)^\mathsf{T} F(x^l).$$

If $\mu_l \to \infty$, we see that the matrix on the LHS approximates the identity. Hence, for very large μ_l we get

$$p^l \approx -\frac{1}{\mu_l} \nabla f(x^l),$$

which is the steepest descent direction with arbitrarily small stepsize $1/\mu_l$, see page 18. Conversely, if $\mu_l \to 0$, the vector p^l approaches the GN direction (4.8). The LM algorithm can therefore be viewed as compromise between the method of steepest descent and GN, i.e., linear local convergence is expected but it can be quadratic near a solution, see Section 4.2.2. The behavior of the LM procedure heavily depends on the particular choice of the damping parameter in each iteration, whose influence may be summarized as follows:

- The regularization term $\mu_l I$ improves the reliability of the algorithm. It accounts for the second order information of the Hessian $\nabla f(x^l)$ and adds regularity to the matrix ${J_l}^\mathsf{T} J_l$.
- A positive μ_l guarantees that the LM step is well defined and that p^l is a descent direction.
- If $\mu_l = 0$, the LM algorithm performs a GN step which computes a direction which may not be unique or a descent direction.
- So, changing μ_l changes the search direction p^l, which interpolates between the GN direction and the steepest descent direction.
- Changing μ_l also changes the step length $\|p^l\|$. As $\mu_l \to \infty$, the step length tends to zero. Thus, by choosing μ_l large enough, we can reduce the objective $f(x^l)$.
- However, values of μ_l which are too large, can lead to arbitrary slow linear convergence.

Many different implementations of the LM algorithm exist. Various more or less heuristic arguments for the choice of the damping parameters have been proposed. A common practice is to adjust μ_l in every iteration by a rule which is based on the past behavior of the algorithm. Typically, we want to choose a large value for μ_l when we are far away from the minimum. As we get closer to a solution, the Hessian tends to become positive definite, i.e., less regularization is necessary and we can decrease μ_l.
Marquardt [28] presented the following strategy. We start with a small value for the damping parameter, e.g., $\mu_l = 10^{-2}$. If, the computed search direction (4.19) leads to

$$f(x^l + p^l) < f(x^l), \tag{4.21}$$

we move on to the next iteration with $x^{l+1} = x^l + p^l$ and set $\mu_{l+1} = \mu_l/\nu$ for some $\nu > 1$. And if $f(x^l)$ does not decrease in iteration l, we gradually increase μ_l by some factor, e.g., $\mu_l \leftarrow \nu\mu_l$, and recompute the direction p^l until (4.21) holds.

In this thesis, we aim to implement the LM algorithm with as few function calls as possible. Marquardt's strategy has the disadvantage that it may need additional function evaluations per iteration in order to ensure that condition (4.21) holds. We want to forgo this testing for descent in our implementation. Thus, it is desirable to determine a proper value for μ_l a priori in each iteration. One way to do this is to adapt the damping parameter to the magnitude of the current function value, i.e., by setting

$$\mu_l = \mathcal{O}\big(\|F(x^l)\|\big) \tag{4.22}$$

in iteration l. That is, when $\|F(x^l)\|$ is large, the method takes a small step in the steepest descent direction. As the procedure nears a solution, $\|F(x^l)\|$ is usually small and the method takes a large step in the GN direction. This effect can be strengthened by fine-tuning the damping parameters, i.e., by multiplying μ_l with a small or large factor when $\|F(x^l)\|$ is small or large, respectively. Consequently, it is very likely that the iterates x^l will form a descent sequence.
The choice (4.22) is not purely heuristic, as it is supported by theoretical arguments. Brown and Dennis showed in [6] that (4.22) can guarantee local quadratic convergence of the algorithm, see also Theorem 4.6 in Section 4.2.2.
The implementation details of the resulting Algorithm 3 are given in Section 4.1.3.

Algorithm 3 Levenberg-Marquardt (LM)

Input: function $F(x)$, Jacobian $J(x)$ of $F(x)$, initial approximation x_0, termination parameters tol, $maxit$
Output: approximate solution of NLLS problem (3.3)
Require: $maxit \in \mathbb{N}$, $0 < tol \ll 1$
Ensure: the damping parameter μ is $\mathcal{O}\big(\|F(x)\|\big)$ and $0 \leq \mu < \infty$ in each iteration

initialize $x = x_0$;
$n \leftarrow$ dim. of x;
$I \leftarrow n \times n$ identity matrix;
for $k = 1$ to $maxit$ **do**
    evaluate $F = F(x)$;
    compute $J = J(x)$;
    if $\|F\|_\infty \geq 10$ **then** ▷ μ large → method of steepest descent
        $c = 10$;
    else if $\|F\|_\infty \leq 1$ **then** ▷ $\mu = 0$ → Gauss-Newton method
        $c = 10^{-3}$;
    else
        $c = 10^{-1}$;
    end if
    $\mu = c \cdot \|F\|_\infty$;
    solve $(\mu I + J^\mathsf{T} J)p = -J^\mathsf{T} F$ for p; ▷ normal equations of (4.18)
    if $\|p\| < tol$ **then**
        break; ▷ stop iteration if the stepsize is too small
    end if
    set $x = x + p$;
end for
return x;

We note that Algorithm 3 may still generate iterates that do not fulfill the descent condition (4.21). This can be fixed by incorporating a line search technique into LM. That is, if p^l is a solution of (4.18), we can determine a stepsize parameter $\alpha_l \geq 0$ with

$$x^{l+1} = x^l - \alpha_l p^l$$

such that (4.21) is guaranteed. This is particularly interesting when we want to enforce the global convergence property of the algorithm, which is discussed on page 62. However, searching for a proper α_l results in additional function evaluations. Thus, implementing a line search is somewhat counterproductive to our initial goal of reducing function calls. And Algorithm 3 is very robust anyway, see Section 5.

Remark (Scaling). Alternative updating strategies for the LM algorithm often include replacing the *stabilizer matrix* I in (4.16) with a diagonal matrix D_l. The idea behind it is to introduce parameter dependence on the step direction. Each component of the gradient can be scaled so that the method may take larger steps along directions where the gradient is small, avoiding slow convergence. Marquardt [28] suggested the choice $D_l = \operatorname{diag}(J_l{}^{\mathsf{T}} J_l)$, which makes the algorithm invariant to scaling. This means that if the parameters x_i of the problem were replaced by the parameters $\tilde{x}_i = \lambda_i x_i$ for some scaling factors $\lambda_i \in \mathbb{R}$, then the algorithm would generate the same sequence of iterates of the values of f. Parameter scaling can be very useful, however, it prevents the researcher from imposing the proper scale of measurements. In addition, a parameter whose corresponding entry on the diagonal of D_l is small decreases the damping effect, making the algorithm less robust.

Remark (LM as trust region method). The Levenberg-Marquardt algorithm can also be implemented as trust region method, see page 16 or Moré [32]. In fact, this technique was first described before the notion of a trust region did even exist.
For a spherical trust region, the subproblem to be solved at iteration l is

$$\begin{aligned} \underset{p \in \mathbb{R}^n}{\text{minimize}} \quad & \frac{1}{2} \left\| J(x^l)p + F(x^l) \right\|_2^2 \\ \text{subject to} \quad & \|p\|_2 \leq \delta_l, \end{aligned} \tag{4.23}$$

where $\delta_l > 0$ is the trust region radius. That is, the quadratic model (cf. (2.13)) is

$$q_l(p) = \underbrace{\frac{1}{2}\left\|F(x^l)\right\|_2^2}_{= f(x^l)} + p^{\mathsf{T}} J(x^l)^{\mathsf{T}} F(x^l) + \frac{1}{2} p^{\mathsf{T}} J(x^l)^{\mathsf{T}} J(x^l) p,$$

i.e., $J_l{}^{\mathsf{T}} J_l$ is used as Hessian approximation. The minimizer of (4.23) is characterized as follows. When the solution p^l of the GN normal equations (4.7) lies strictly inside the ball of radius δ_l, i.e., when $\|p^l\| < \delta_l$, then it also solves the problem (4.23). Otherwise, there exists a $\mu > 0$ such that the minimum $\hat{p}^l$ of (4.23) satisfies $\|\hat{p}^l\| = \delta_l$ and

$$\left[\mu_l I + J(x^l)^{\mathsf{T}} J(x^l)\right] \hat{p}^l = -J(x^l)^{\mathsf{T}} F(x^l).$$

The proof of this claim can be found in [40, p. 258–259]. Provided that the model is accepted, the next iteration point is then chosen as $x^{l+1} = x^l + \hat{p}^l$.

The LM algorithm performs well in practice and has become a standard for the optimization of medium-sized NLLS problems. Due to its robustness, it may find a solution even when it starts very far from the minimum. However, for well-behaved functions and for reasonable good starting parameters, the LM procedure usually tends to be a bit slower than the GN method.
On large-residual problems, the linear convergence of the LM algorithm can be very slow. Furthermore, if the residuals are large near a solution, both the LM and GN methods will perform poorly, since then $J^\mathsf{T}J$ is a bad model of the Hessian. In such a case, if applicable, a Newton or quasi-Newton method would be a better choice.
In terms of data fitting, the LM algorithm is especially useful when the data is not well approximated by a model function. However, if the residuals are too large, the function is probably a poor fit to the data and should be replaced by a better model.

4.2.1 Finite difference analogue

The *finite difference analogue of the Levenberg-Marquardt algorithm* (FDLM) follows the same principle as the FDGN from Section 4.1.1. That is, contrary to the LM algorithm, the FDLM uses forward differences to approximate the Jacobian in each iteration. Hence, relationship (4.16) transforms into the FDLM step

$$x^{l+1} = x^l - \left[\mu_l I + \frac{\Delta F(x^l, h_l)^\mathsf{T} \Delta F(x^l, h_l)}{h_l^2}\right]^{-1} \frac{\Delta F(x^l, h_l)^\mathsf{T}}{h_l} \cdot F(x^l), \tag{4.24}$$

where

$$\frac{\Delta F(x^l, h_l)}{h_l}, \quad h_l \in \mathbb{R},$$

denotes the matrix of difference quotients (4.10) in iteration l. If $\mu_l = 0$, equation (4.24) is simply the FDGN step (4.11). Whenever $\mu_l > 0$, the matrix

$$\mu_l I + \frac{\Delta F(x^l, h_l)^\mathsf{T} \Delta F(x^l, h_l)}{h_l^2}$$

is positive definite and thus, unless $f(x^l)$ is a stationary point, step (4.24) is heading in a descent direction, cf. (4.20). Because of (4.12), the LM step (4.16) can be understood as the special case of the FDLM step (4.24) with $h_l = 0$.

We refer to the discussion on page 47 for more information regarding the forward difference approximation. Analogously to the FDGN, we set

$$h_l = \mathcal{O}\big(\|F(x^l)\|\big)$$

in iteration l of the FDLM, as this is in agreement with Theorem 4.6, which is stated in the following Section 4.2.2. Furthermore, the same theorem indicates that we should take the damping parameter as in the LM algorithm implementation, namely as

$$\mu_l = \mathcal{O}\big(\|F(x^l)\|\big).$$

There is also an intuitive explanation for this choice, see page 53. Thus, Algorithm 4 basically incorporates the forward difference approximation from Algorithm 2 (FDGN) and the damping strategy from Algorithm 3 (LM), see Section 4.2.3 for details.

Algorithm 4 Finite difference Levenberg-Marquardt (FDLM)

Input: function $F(x)$, initial approximation x_0, termination parameters tol, $maxit$
Output: approximate solution of NLLS problem (3.3)
Require: $maxit \in \mathbb{N}$, $0 < tol \ll 1$
Ensure: the damping parameter μ is $\mathcal{O}(\|F(x)\|)$ and $0 \leq \mu < \infty$ in each iteration, h is sufficiently small in each iteration - ideally $|h|$ is $\mathcal{O}\big(\|F(x)\|\big)$ but δ is used as safeguard to prevent absurd choices of h relative to the iterate x

initialize $x = x_0$;
$n \leftarrow$ dim. of x;
$m \leftarrow$ dim. of $F(x)$;
$h,\ \delta \leftarrow n$-dim. zero vector;
$J \leftarrow m \times n$ zero matrix;
for $k = 1$ to $maxit$ **do**
    evaluate $F = F(x)$;
    if $\|F\|_\infty \geq 10$ **then** ▷ μ large → method of steepest descent
        $c = 10$;
    else if $\|F\|_\infty \leq 1$ **then** ▷ $\mu = 0$ → Gauss-Newton method
        $c = 10^{-3}$;
    else
        $c = 10^{-1}$;
    end if
    $\mu = c \cdot \|F\|_\infty$;
    ▷ finite difference approximation $J = J(x)$ of the Jacobian of $F(x)$
    for $i = 1$ to m **do**
        for $j = 1$ to n **do**
            if $|x_j| < 10 \cdot \sqrt{eps}$ **then**
                $\delta_j = 10^{-2} \cdot \sqrt{eps}$;
            else
                $\delta_j = 10^{-3} \cdot |x_j|$;
            end if
            $h_j = \min(\|F\|,\ \delta_j)$;
            $e_j \leftarrow$ the j-th unit vector of dim. n;
            compute the forward difference approximation

$$J_{ij} = \frac{F_i(x + h_j e_j) - F_i(x)}{h_j};$$

        end for
    end for
    solve $(\mu I + J^\mathsf{T} J)p = -J^\mathsf{T} F$ for p; ▷ normal equations of (4.18)
    if $\|p\| < tol$ **then**
        break; ▷ stop iteration if the stepsize is too small
    end if
    set $x = x + p$;
end for
return x;

4.2.2 Convergence analysis

We are following [6] for the convergence analysis of the FDLM. The statements and proofs also hold for the FDGN by merely setting the damping parameter μ_l to zero whenever it occurs. We obtain the convergence properties of the LM and the GN methods as immediate consequences.
In the following, we assume that $\frac{\Delta F(x,h)}{h} \to J(x)$ as $h \to 0$. Hence, we may use the notational convention $\frac{\Delta F(x,0)}{0} \equiv J(x)$ for the limit.

Lemma 4.1. *Let $F : \mathbb{R}^n \to \mathbb{R}^m$ be continuously differentiable on some open and convex set $\mathcal{C} \subset \mathbb{R}^n$. Then, for every $x, y \in \mathcal{C}$ it holds that*

$$F(y) - F(x) = \int_0^1 J\big(x + t(y-x)\big)\,(y-x)\,dt = \int_x^y J(z)\,dz.$$

Lemma 4.2. *Let $G : \mathcal{C} \subset \mathbb{R}^n \to \mathbb{R}^{m\times n}$, where $\mathcal{C} \subset \mathbb{R}^n$ is an open and convex set and let $x, y \in \mathcal{C}$. Then, if G is integrable on the interval $[x, y]$, it holds that*

$$\left\| \int_0^1 G\big(x + t(y-x)\big)\,(y-x)dt \right\| \le \int_0^1 \left\| G\big(x + t(y-x)\big)\,(y-x) \right\| dt.$$

These two basic properties of the integral are verified in [8, p. 74–75] and we only need them to prove the following statement.

Lemma 4.3. *Let the Jacobian J of F be Lipschitz continuous on some open and convex set $\mathcal{C} \subset \mathbb{R}^n$, i.e., there exists a constant $L \ge 0$ such that*

$$\big\|J(x) - J(y)\big\|_2 \le L\big\|x - y\big\|_2, \quad \forall x, y \in \mathcal{C}. \tag{4.25}$$

Then, for every $x, y \in \mathcal{C}$ it holds that

(i) $\big\|J(x)^\mathsf{T} - J(y)^\mathsf{T}\big\|_2 \le L\big\|x - y\big\|_2$,

(ii) there exist constants $C, \tilde{C} \ge 0$ such that

$$\big\|J(x) - J(y)\big\|_1 \le CL\big\|x - y\big\|_1 \tag{4.26}$$

and $\|A\|_2 \le \tilde{C}\|A\|_1$, where $A \in \mathbb{R}^{m\times n}$ is some rectangular matrix, and

(iii) $\big\|F(x) - F(y) - J(y)(x-y)\big\|_1 \le \frac{CL}{2}\big\|x - y\big\|_1^2$.

Proof. (i) We show that the nonzero eigenvalues of AB and BA are the same for matrices $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times m}$. Let us suppose that $v \in \mathbb{R}^m$ is an eigenvector of AB corresponding to some eigenvalue $\lambda \neq 0$. Then $ABv = \lambda v$, $Bv \neq 0$ and

$$(BA)Bv = B(ABv) = \lambda Bv,$$

so Bv is an eigenvector of BA corresponding to the same eigenvalue. Hence, by setting $B = A^\mathsf{T}$ we see that the matrices $A^\mathsf{T}A$ and AA^T have the same nonzero eigenvalues. It is easy to show that $A^\mathsf{T}A$ and AA^T are positive semidefinite and thus

$$\big\|A\big\|_2 = \sqrt{\lambda_{\max}(A^\mathsf{T}A)} = \sqrt{\lambda_{\max}(AA^\mathsf{T})} = \big\|A^\mathsf{T}\big\|_2, \tag{4.27}$$

where $\lambda_{\max}$ denotes the largest eigenvalue (see equation (A.1.1) in Appendix A). So,

$$\big\|J(x)^\mathsf{T} - J(y)^\mathsf{T}\big\|_2 = \big\|J(x) - J(y)\big\|_2, \quad \forall x, y \in \mathcal{C}.$$

(ii) Since the norms $\|\cdot\|_1$ and $\|\cdot\|_2$ are equivalent, we can find constants $\tilde{c}, \tilde{C} \geq 0$ with

$$\tilde{c}\,\|A\|_1 \leq \|A\|_2 \leq \tilde{C}\,\|A\|_1, \quad A \in \mathbb{R}^{m\times n}.$$

Moreover, we have for $C', C'' \geq 0$ and by the Lipschitz continuity (4.25) of J that

$$\big\|J(x) - J(y)\big\|_1 \leq C'\big\|J(x) - J(y)\big\|_2 \leq C'L\big\|x - y\big\|_2 \leq \underbrace{C''C'}_{=:\,C} L\big\|x - y\big\|_1, \quad \forall x, y \in \mathcal{C}.$$

(iii) In order to prove the inequality, we let $x, y \in \mathcal{C}$ and utilize the two auxiliary results Lemma 4.1 and Lemma 4.2. Now, since J is Lipschitz on $\mathcal{C}$, it is integrable on the interval $[x, y]$ and we find

$$\begin{aligned}
&\big\|F(x) - F(y) - J(y)(x-y)\big\|_1 = \left\|\int_y^x J(z)\,dz - J(y)(x-y)\right\|_1 \\
&= \left\|\int_y^x J(z) - J(y)\,dz\right\|_1 = \left\|\int_0^1 \big[J\big(y + t(x-y)\big) - J(y)\big](x-y)\,dt\right\|_1 \\
&\leq \int_0^1 \big\|J\big(y + t(x-y)\big) - J(y)\big\|_1 \cdot \big\|x - y\big\|_1\,dt \overset{(4.26)}{\leq} CL\big\|x-y\big\|_1^2 \int_0^1 t\,dt \\
&= \frac{CL}{2}\big\|x - y\big\|_1^2.
\end{aligned}$$

□

Theorem 4.4. *Let the Jacobian J of F be Lipschitz continuous on some open and convex set $\mathcal{C} \subset \mathbb{R}^n$, i.e., there exists a constant $L \geq 0$ such that*

$$\big\|J(x) - J(y)\big\|_2 \leq L\big\|x - y\big\|_2, \quad \forall x, y \in \mathcal{C}.$$

Furthermore, let

$$\nabla f(x^*) = J(x^*)^\mathsf{T}F(x^*) = 0$$

for some $x^* \in \mathcal{C}$. If $L\|F(x^*)\|_2$ is a strict lower bound for the spectrum of $J(x^*)^\mathsf{T}J(x^*)$ and μ_l is any bounded nonnegative sequence in $\mathbb{R}$, then there exists a constant $M > 0$ such that if every $|h_l| \leq M$ and $h_l \to 0$ as $l \to \infty$, the FDLM converges locally to x^*.

Proof. For each $x \in \mathcal{C}$, let $\lambda_{\min}(x, h)$ denote the smallest eigenvalue of

$$\frac{\Delta F(x,h)^\mathsf{T}\Delta F(x,h)}{h^2}.$$

That is, we may identify the least eigenvalue of $J(x^*)^\mathsf{T} J(x^*)$ with $\lambda_{\min}(x^*, 0)$. By assumption, we have

$$\lambda_{\min}(x^*, 0) > L \left\| F(x^*) \right\|_2 \geq 0.$$

Hence, we can find an open, convex neighborhood $\mathcal{S}$ of x^* with $\overline{\mathcal{S}} \subset \mathcal{C}$ and $M' > 0$ such that for all $x \in \overline{\mathcal{S}}$ and $|h| \leq M'$:

$$\lambda_{\min}(x, h) \geq \lambda := \frac{\lambda_{\min}(x^*, 0) + L \left\| F(x^*) \right\|_2}{2}.$$

Thus, $\lambda_{\min}$ is a nonnegative and jointly continuous function of $x \in \overline{\mathcal{S}}$ and $h \in [-M', M']$. By continuity, there exists a uniform bound $B \geq 0$ with

$$\left\| \frac{\Delta F(x, h)^\mathsf{T}}{h} \right\|_2 \leq B, \quad x \in \overline{\mathcal{S}},\ |h| \leq M'.$$

We consider the FDLM step (4.24) for $|h_l| \leq M'$ and define the absolute error in the current iterate $x^l \in \overline{\mathcal{S}}$ as

$$\epsilon_l := \left\| x^l - x^* \right\|_2.$$

If we set

$$R_l := \mu_l I + \frac{\Delta F(x^l, h_l)^\mathsf{T} \Delta F(x^l, h_l)}{h_l^2},$$

we notice that the smallest eigenvalue of the square matrix R_l is

$$\mu_l + \lambda_{\min}\big(x^l, h_l\big) \geq \mu_l + \lambda > 0.$$

This implies that R_l is positive definite and thus invertible. The eigenvalues of the inverse R_l^{-1} are the reciprocals of the eigenvalues of R_l. And, since R_l^{-1} is also positive definite, its singular values coincide with its eigenvalues (see Appendix A). Hence, we can bound its 2-norm:

$$\frac{1}{\mu_l + \lambda_{\min}(x^l, h_l)} \overset{(A.1.1)}{=} \left\| R_l^{-1} \right\|_2 \leq \frac{1}{\mu_l + \lambda}.$$

Moreover, the nonsingularity of R_l guarantees the existence of x^{l+1}, the next iterate obtained from (4.24). The line segment passing through the points $\big(x^*, F(x^*)\big)$ and $\big(x^l, F(x^l)\big)$ can be represented by the point-slope form

$$F(x^*) - F(x^l) = J\big(x^l - t(x^* - x^l)\big)(x^* - x^l),$$

where $t \in (0, 1)$. Accordingly, the error $\epsilon_{l+1} := \left\| x^{l+1} - x^* \right\|_2$ can be bounded as follows:

$$
\begin{aligned}
\epsilon_{l+1} &= \Big\| x^l - R_l^{-1} h_l^{-1} \Delta F(x^l, h_l)^{\mathsf{T}} F(x^l) - x^* \Big\|_2 \\
&= \Big\| x^l - x^* - R_l^{-1} h_l^{-1} \Delta F(x^l, h_l)^{\mathsf{T}} \Big[F(x^*) + J\big(x^l - t(x^* - x^l)\big)(x^l - x^*) \Big] \Big\|_2 \\
&\leq \Big\| \Big[I - R_l^{-1} h_l^{-1} \Delta F(x^l, h_l)^{\mathsf{T}} J\big(x^l - t(x^* - x^l)\big) \Big] (x^l - x^*) \Big\|_2 \\
&\quad + \Big\| -R_l^{-1} h_l^{-1} \Delta F(x^l, h_l)^{\mathsf{T}} F(x^*) \Big\|_2 \\
&\leq \big\| R_l^{-1} \big\|_2 \cdot \Big\| R_l - h_l^{-1} \Delta F(x^l, h_l)^{\mathsf{T}} J\big(x^l - t(x^* - x^l)\big) \Big\|_2 \cdot \Big\| x^l - x^* \Big\|_2 \\
&\quad + \big\| R_l^{-1} \big\|_2 \cdot \Big\| \underbrace{J(x^*)^{\mathsf{T}} F(x^*)}_{=\,0} - h_l^{-1} \Delta F(x^l, h_l)^{\mathsf{T}} F(x^*) \Big\|_2 \\
&\leq (\mu_l + \lambda)^{-1} \Bigg\{ \Big\| R_l - h_l^{-1} \Delta F(x^l, h_l)^{\mathsf{T}} J\big(x^l + t(x^* - x^l)\big) \Big\|_2 \cdot \epsilon_l \\
&\quad + \Big\| J(x^*) - h_l^{-1} \Delta F(x^l, h_l) \Big\|_2 \cdot \big\| F(x^*) \big\|_2 \Bigg\} \\
&= (\mu_l + \lambda)^{-1} \Bigg\{ \Big\| \mu_l I + h_l^{-2} \Delta F(x^l, h_l)^{\mathsf{T}} h_l^{-1} \Delta F(x^l, h_l) \\
&\quad - h_l^{-1} \Delta F(x^l, h_l)^{\mathsf{T}} \Big[J(x^l) - J(x^l) - J\big(x^l + t(x^* - x^l)\big) \Big] \Big\|_2 \cdot \epsilon_l \\
&\quad + \Big\| J(x^*) - J(x^l) + J(x^l) - h_l^{-1} \Delta F(x^l, h_l) \Big\|_2 \cdot \big\| F(x^*) \big\|_2 \Bigg\} \\
&\leq (\mu_l + \lambda)^{-1} \Bigg\{ \Big[\mu_l + B \Big\| h_l^{-1} \Delta F(x^l, h_l) - J(x^l) \Big\|_2 \\
&\quad + B \Big\| J(x^l) - J\big(x^l + t(x^* - x^l)\big) \Big\|_2 \Big] \cdot \epsilon_l \\
&\quad + \Big[\Big\| J(x^*) - J(x^l) \Big\|_2 + \Big\| J(x^l) - h_l^{-1} \Delta F(x^l, h_l) \Big\|_2 \Big] \cdot \big\| F(x^*) \big\|_2 \Bigg\}.
\end{aligned}
$$

We note that we used the submultiplicative property of the induced matrix 2-norm, relationship (4.27), $\|I\|_2 = 1$ and the preceding boundaries for the above estimation. Furthermore, according to Lemma 4.3 (iii), we observe that

$$
\Big\| \Delta F(x^l, h_l) - J(x^l) h_l \Big\|_1 \leq \frac{CL}{2} \left| h_l \right|^2 .
$$

Hence, if we also apply Lemma 4.3 (ii) and consider the Lipschitz continuity of J, we get for the error

$$
\begin{aligned}
\epsilon_{l+1} &\leq (\mu_l + \lambda)^{-1} \Bigg\{ \Big[\mu_l + 2^{-1} B \tilde{C} C L \left| h_l \right| + BL \big\| t(x^* - x^l) \big\|_2 \Big] \epsilon_l \\
&\qquad + \Big[L \big\| x^l - x^* \big\|_2 + 2^{-1} \tilde{C} C L \left| h_l \right| \Big] \cdot \big\| F(x^*) \big\|_2 \Bigg\} \\
&\leq (\mu_l + \lambda)^{-1} \Big[\mu_l + 2^{-1} B \tilde{C} C L \left| h_l \right| + BL \epsilon_l + L \big\| F(x^*) \big\|_2 \Big] \epsilon_l \\
&\qquad + 2^{-1} (\mu_l + \lambda)^{-1} \tilde{C} C L |h_l| \cdot \big\| F(x^*) \big\|_2 .
\end{aligned} \tag{4.28}
$$

Now, we select an even smaller neighborhood of x^*, namely $\mathcal{N}(x^*, r) \subset \mathcal{S}$ with $r > 0$, and $M'' > 0$ with $M'' < M'$ such that

$$\lambda_{\min}(x^*, 0) - L\left\|F(x^*)\right\|_2 > B\tilde{C}CLM'' + 2BLr.$$

Then

$$\lambda > 2^{-1}B\tilde{C}CLM'' + BLr + L\left\|F(x^*)\right\|_2,$$

which ensures that $0 \leq \delta < 1$, where

$$\delta := \sup_l (\mu_l + \lambda)^{-1} \left[\mu_l + 2^{-1}B\tilde{C}CLM'' + BLr + L\left\|F(x^*)\right\|_2\right].$$

Let us choose $M > 0$ with $M < M''$ such that

$$M \leq \frac{2\lambda(1-\delta)r}{\tilde{C}CL\left\|F(x^*)\right\|_2}, \quad \text{for } \left\|F(x^*)\right\|_2 \neq 0, \tag{4.29}$$

or

$$M \leq \frac{2(\lambda - BLr)}{B\tilde{C}CL}, \quad \text{if } \left\|F(x^*)\right\|_2 = 0. \tag{4.30}$$

Finally, let us assume that we have $\epsilon_l \leq r$ and $|h_l| \leq M$ for $l \geq 0$. From the estimate (4.28), we obtain for the choice (4.29) that

$$\epsilon_{l+1} \leq \delta\epsilon_l + (\mu_l + \lambda)^{-1}\lambda(1-\delta)r \leq \delta\epsilon_l + (1-\delta)r,$$

and (4.30) yields

$$\epsilon_{l+1} \leq (\mu_l + \lambda)^{-1}\left[\mu_l + \lambda - BLr + BLr\right]r = r.$$

That is, the error satisfies

$$\epsilon_{l+1} \leq \delta\epsilon_l + (1-\delta)r \leq r,$$

which shows that the FDLM is locally well-defined. The convergence of x^l to x^*, as long as $h_l \to 0$, is explained as follows. We observe that the FDLM step (4.24) can be rewritten as the first order linear difference equation

$$x^{l+1} - x^l = -h_l\left[h_l^2\mu_l I + \Delta F(x^l, h_l)^{\mathsf{T}}\Delta F(x^l, h_l)\right]^{-1}\Delta F(x^l, h_l)^{\mathsf{T}}F(x^l),$$

where the inhomogeneous part (the RHS) is a sequence converging to zero as $l \to \infty$. It is shown in [48] that the homogeneous part (the LHS) of the difference equation must then converge to zero as well. □

Corollary 4.5. *Under the assumptions of Theorem 4.4, the LM algorithm converges locally for any bounded nonnegative sequence μ_l in $\mathbb{R}$ and the order of convergence is at least one.*

Proof. The LM step (4.16) is the special case of the FDLM step (4.24) with $h_l = 0$. Consequently, M can be set to zero in Theorem 4.4 and thus (4.28) yields

$$\epsilon_{l+1} \leq \delta\epsilon_l, \quad 0 \leq \delta < 1,$$

where $M'' = 0$ in the definition of δ. Hence, the convergence is at least linear. □

Theorem 4.6. *Let the Jacobian J of F be Lipschitz continuous with constant $L \geq 0$ on some open and convex set $\mathcal{C} \subset \mathbb{R}^n$ and let $x^* \in \mathcal{C}$ with $F(x^*) = 0$. If μ_l is a bounded nonnegative sequence in $\mathbb{R}$, then there exists a constant $M > 0$ such that if every $|h_l| \leq M$ and $h_l \to 0$ as $l \to \infty$, the FDLM converges locally to x^*. If μ_l and $|h_l|$ are additionally $\mathcal{O}\big(\|F(x^l)\|\big)$, then the method is quadratically convergent.*

Proof. The course of action is as in the proof of Theorem 4.4 until inequality (4.28). Since x^* is a zero of F, the estimate (4.28) reduces to

$$\epsilon_{l+1} \leq (\mu_l + \lambda)^{-1}\big[\mu_l + 2^{-1}B\tilde{C}CL|h_l| + BL\epsilon_l\big]\epsilon_l. \tag{4.31}$$

The local convergence is shown analogously by choosing $M > 0$ as in (4.30). In order to see that the convergence is quadratic, we consider $\big\|F(x^l)\big\|_2$ for $x^l \in \overline{\mathcal{N}}(x^*, r)$. According to Taylor's Theorem, we have

$$F(x^l) - F(x^*) \approx J(x^*)(x^l - x^*),$$

and since there is a uniform upper bound on $\|J(x^*)\|_2$ for $\|x - x^*\|_2 \leq r$, we find

$$\begin{aligned} \big\|F(x^l)\big\|_2 &= \big\|F(x^l) \underbrace{-F(x^*)}_{=\,0}\big\|_2 \\ &\leq \sup_x \big\|J(x)\big\|_2 \underbrace{\big\|x^l - x^*\big\|_2}_{=\,\epsilon_l}, \end{aligned} \tag{4.32}$$

where $x \in [x^l, x^*] \subset \overline{\mathcal{N}}(x^*, r)$. By assumption, μ_l and $|h_l|$ are $\mathcal{O}\big(\|F(x^l)\|\big)$ with arbitrary vector norm $\|\cdot\|$. Hence, relationship (4.32) and the equivalence of norms yield that they are also $\mathcal{O}(\epsilon_l)$. So, it is easy to see from (4.31) that $\epsilon_{l+1} = \mathcal{O}(\epsilon_l^2)$. □

Corollary 4.7. *Under the assumptions of Theorem 4.6, the LM algorithm converges quadratically.*

Remark (Global convergence). For positive damping parameters, the LM algorithm and the FDLM produce steps that are taken in directions of descent, see condition (4.20). The generated sequence of the x^l may be descending if we are careful in the choice of the nonnegative sequence μ_l. Furthermore, we can find a local solution of the optimization problem by requiring that for $\mathcal{C} \subset \mathbb{R}^n$ open and convex, $x^0 \in \mathcal{C}$, the level set

$$\mathcal{C}_0 := \{x \in \mathcal{C} \mid f(x) \leq f(x^0)\}$$

is in $\operatorname{int}\mathcal{C}$ and bounded, see condition (A.3.3) from Theorem A.23 in Appendix A. Now, if the x^l form a descent sequence in $\mathcal{C}_0$ (which can be enforced by the implementation of a line search in LM or FDLM), then some subsequence is convergent to a stationary point $x^* \in \mathcal{C}_0$ of f by Proposition A.26 (i). If the respective algorithm is locally convergent, then, as soon as some term of the convergent subsequence gets sufficiently close to x^*, the entire sequence x^l will converge to it.
This does generally not apply for the GN method and the FDGN because the generation of a descent direction in each iteration cannot be guaranteed, see page 45.

4.2.3 Implementation details

The LM algorithm, like the GN method, demands an user-supplied analytic Jacobian as input. The forward difference approximation in each iteration of the FDLM is implemented in the exact same way as for the FDGN, see Section 4.1.3.
In what follows, we describe the incorporation of the damping strategy from page 53 for the LM algorithm and the FDLM. The damping parameter μ_l needs to be $\mathcal{O}\big(\|F(x^l)\|\big)$ in iteration l of the respective method. This can be realized by choosing a constant $c > 0$ such that

$$\mu_l = c \cdot \big\|F(x^l)\big\|.$$

Although the norm $\|\cdot\|$ can be chosen arbitrarily in order to satisfy the requirement of Theorem 4.6, we found that the algorithms performed best for the maximum norm $\|\cdot\|_\infty$. From a heuristic point of view this makes sense, since it yields a smaller value for μ_l than the p-norms but μ_l is usually still large enough to ensure a well-conditioned matrix $\mu_l I + {J_l}^\mathsf{T} J_l$. That is, by choosing μ_l sufficiently but not overly large we produce a more sensitive regularization. We recall that we aim to use a relatively large value of μ_l when we are far from a solution, and then decrease it significantly when we are close to it. This behavior is predefined by $\big\|F(x^l)\big\|_\infty$, but it turns out that we can speed up the convergence of the algorithms by properly adjusting the factor c. Brown and Dennis [6] initially proposed the adaptive choice

$$c = \begin{cases} 10 & \text{whenever } 10 \leq \big\|F(x^l)\big\|_\infty, \\ 1 & \text{whenever } \ 1 < \big\|F(x^l)\big\|_\infty < 10, \\ 10^{-1} & \text{whenever } \quad \big\|F(x^l)\big\|_\infty \leq 1, \end{cases}$$

for double precision arithmetic. However, we could drastically increase the efficiency of our MATLAB implementations by setting

$$c = \begin{cases} 10 & \text{whenever } 10 \leq \big\|F(x^l)\big\|_\infty, \\ 10^{-1} & \text{whenever } \ 1 < \big\|F(x^l)\big\|_\infty < 10, \\ 10^{-3} & \text{whenever } \quad \big\|F(x^l)\big\|_\infty \leq 1. \end{cases}$$

This slight adjustment tremendously impacts the overall performance of our programs. We tested the LM algorithm and the FDLM on the 35 test functions from Section 5 with different starting points and, apart from a few exceptions, observed a significant decrease in the number of iterations and function evaluations needed for the minimization.
The normal equations in step 15 of Algorithm 3 and step 28 of Algorithm 4 are solved by applying the backslash operator \ and require $\mathcal{O}(mn^2 + \frac{n^3}{3})$ operations, see page 36. The FDLM needs $\mathcal{O}(mn)$ additional operations and $n + 1$ evaluations of F for the computation of the forward differences. The amount of extra work for the LM algorithm depends on the invoked numerical differentiation procedure.
The actual code of the MATLAB implementations can be found as C.3 (LM) and C.4 (FDLM) in Appendix C. In C.4, we got rid of the outer for-loop for the finite difference approximation (see step 16 of Algorithm 4) by using vectorization. Both programs terminate when the objective function is minimized or some user-defined maximum number of function evaluations is reached.

4.3 DUD Algorithm

The algorithm DUD (for Doesn't Use Derivatives), by Ralston and Jennrich [44], was specifically designed for data fitting. That is, for problems where one seeks a parameter vector $\theta \in \mathbb{R}^p$, $p \ll n$, such that the RSS (3.12), in vector notation

$$Q(\theta) = \left\| y - f(\theta) \right\|_2^2, \tag{4.33}$$

is minimal. We recall that $y \in \mathbb{R}^n$ represents the observed data and that $f : \mathbb{R}^p \to \mathbb{R}^n$ is the response function whose components $f_i : \mathbb{R}^p \to \mathbb{R}$ are the model predictions corresponding to the i-th data point y_i.
DUD can be thought of as a Gauss-Newton-like method, in the sense that it minimizes the RSS (4.33) by solving a sequence of linear least square problems. However, instead of approximating the nonlinear function f by its tangent function, DUD uses an affine function for the linearization. More precisely, in iteration l of the GN method the response $f(\theta)$, for θ close to the iterate θ^l, is approximated by the linear function

$$\tilde{\ell}(\theta) := f(\theta^l) + J(\theta^l)(\theta - \theta^l), \tag{4.34}$$

cf. (4.5), where J now denotes the Jacobian of f. This equation describes the tangent hyperplane to the surface $f(\theta)$ at θ^l. The next parameter θ^{l+1} is then chosen to minimize $\left\| y - \tilde{\ell}(\theta) \right\|_2^2$, i.e., it is the point on the tangent hyperplane closest to y. In contrast, DUD approximates the response $f(\theta)$ by a linear function ℓ that agrees with $f(\theta)$ at $p+1$ points. This describes a secant hyperplane to the surface $f(\theta)$. The exact definition of the function $\ell : \mathbb{R}^p \to \mathbb{R}^n$ is given in the following description of iteration l of DUD, where we suppress the iteration index to improve the readability.

DUD maintains a current set of $p+1$ approximations to the estimator $\hat{\theta} \in \mathbb{R}^p$, the minimizer of the RSS (4.33). These parameters, denoted by $\theta_1, \ldots, \theta_{p+1} \in \mathbb{R}^p$, have been computed in previous iterations and are so that they span the parameter space. The order of subscripts is from the oldest to the youngest, i.e., θ_1 is the oldest member of the current set, meaning that it has gone through the largest number of iterations of the algorithm. DUD approximates $f(\theta)$ by a linear function $\ell(\theta)$ that is equal to $f(\theta)$ at $\theta_1, \ldots, \theta_{p+1}$. The parameter $\theta_{new} \in \mathbb{R}^p$, which ideally reduces the RSS (4.33), is determined by minimizing the distance between $\ell(\theta)$ and y. Then, in order to obtain the parameter set for the next iteration, one of the old members (if possible the oldest) gets replaced by θ_{new} (or optionally by a point on the line between θ_{new} and θ_{p+1}).

Before we discuss the initialization of the parameter set and its detailed updating strategy, we take a closer look at the computation of θ_{new} in the current iteration.
Since $\theta_1, \ldots, \theta_{p+1}$ span the p-dimensional parameter space, it can be shown that the vectors $\theta_i - \theta_{p+1}$, $i = 1, \ldots, p$, are linearly independent and thus form a basis for $\mathbb{R}^p$.

Hence, any $\theta \in \mathbb{R}^p$ can be expressed as

$$\begin{aligned}\theta &= \theta_{p+1} + \underbrace{\theta - \theta_{p+1}}_{\in\, \mathbb{R}^p} \\ &= \theta_{p+1} + \sum_{i=1}^{p} \alpha_i(\theta_i - \theta_{p+1}) \\ &= \theta_{p+1} + \Delta\Theta\alpha, \end{aligned} \tag{4.35}$$

where $\alpha \in \mathbb{R}^p$ and $\Delta\Theta \in \mathbb{R}^{p\times p}$ is the matrix with i-th column $\theta_i - \theta_{p+1}$, $i = 1, \ldots, p$. Now, let

$$\ell(\theta) := f(\theta_{p+1}) + \Delta F\alpha, \tag{4.36}$$

where $\Delta F \in \mathbb{R}^{n\times p}$ is the matrix with i-th column $f(\theta_i) - f(\theta_{p+1})$, $i = 1, \ldots, p$. Since θ_i, $i = 1, \ldots, p$, can be expressed by choosing α in (4.35) as the i-th unit vector. We observe that $\ell(\theta_{p+1}) = f(\theta_{p+1})$ for $\alpha = 0$ and if α is the i-th unit vector, i.e., $\alpha = e_i$ for $i \in \{1, \ldots, p\}$, then

$$\ell(\theta_i) = f(\theta_i).$$

Furthermore, by using (4.35) and the fact that $\Delta\Theta$ is nonsingular by definition, we can write (4.36) as

$$\ell(\theta) = f(\theta_{p+1}) + \Delta F\Delta\Theta^{-1}(\theta - \theta_{p+1}),$$

the secant hyperplane which passes through the points $\big(\theta_i, f(\theta_i)\big)$, $i = 1, \ldots, p+1$. That is, DUD basically replaces the Jacobian of the GN method by $\Delta F\Delta\Theta^{-1}$, cf. (4.34). In effect, DUD's linear approximation (4.36) is written as the function $\ell(\alpha)$ dependent on α and the corresponding LLS problem of minimizing the distance

$$\tilde{Q}(\alpha) = \big\|y - \ell(\alpha)\big\|_2^2$$

is solved. It is easy to see that this yields the normal equations (c.f. (3.9))

$$\Delta F^{\mathsf{T}}\Delta F\alpha = \Delta F^{\mathsf{T}}\big(y - f(\theta_{p+1})\big)\,, \tag{4.37}$$

whose solution is given by the linear least squares estimator

$$\hat{\alpha} = \big(\Delta F^{\mathsf{T}}\Delta F\big)^{-1}\Delta F^{\mathsf{T}}\big(y - f(\theta_{p+1})\big)\,. \tag{4.38}$$

The parameter θ_{new} is then obtained by setting $\alpha = \hat{\alpha}$ in (4.35), i.e.,

$$\theta_{new} = \theta_{p+1} + \Delta\Theta\hat{\alpha}.$$

We can solve (4.37) by a method for normal equations, as explained in Section 3.2.1. However, the matrix $\Delta F^{\mathsf{T}}\Delta F$ may not be positive definite. Ralston and Jennrich [44] suggested computing (4.38) directly by employing Gauss-Jordan elimination with pivoting as described in [22]. This method uses pivot tolerance in order to prevent a full inversion of $\Delta F^{\mathsf{T}}\Delta F$ if the matrix is essentially singular. The order of pivoting is determined by stepwise regression, which is described in more detail in Section 4.3.2. This approach may produce a plausible estimate of $\hat{\alpha}$ even when $\Delta F^{\mathsf{T}}\Delta F$ is singular.

Ideally, θ_{new} reduces the RSS, i.e., it holds that

$$Q(\theta_{new}) < Q(\theta_{p+1}), \tag{4.39}$$

where θ_{p+1} is the estimate computed in the previous iteration and thus the youngest member of the parameter set. But similar to the GN method, condition (4.39) may not be achieved without the employment of a line search. In fact, since derivatives are not used, there is no guarantee that θ_{new} even lies in a descent direction from θ_{p+1}. DUD can attempt to find a new estimate which decreases the RSS by invoking a step shortening procedure. A technique that has a good chance to find a better estimate is to search the line between θ_{new} and θ_{p+1} in opposed directions. That is, we select

$$\theta_{new} = d\theta_{new} + (1 - d)\theta_{p+1},$$

where d is the first member of the sequence

$$d_i = \begin{cases} 1 & \text{for } i = 0, \\ -(-\frac{1}{2})^i & \text{for } i = 1, \dots, m, \end{cases} \tag{4.40}$$

in order that (4.39) is satisfied. If there is no such d_i, we keep the last found estimate, i.e., the θ_{new} which corresponds to $d = d_m$. So if there is no improvement, this allows us to take different optimization paths by varying the user supplied input $m \in \mathbb{N}$.

Remark. With each additional iteration i a point on the line closer to θ_{p+1} is chosen. We observe that m not only determines the size and the direction of the partial step, but also the number of additional function calls. If $Q(\theta_{new}) > Q(\theta_{p+1})$, the RSS needs to be computed for every new estimate found by the line search. However, DUD was primarily intended for problems in which the cost of evaluating the response function f, and hence the RSS, is exorbitant. If this is the case, the step shortening procedure has to be used sparingly, i.e., it should only be invoked when the algorithm would otherwise fail to converge.

DUD only demands a single user supplied starting parameter $\theta_{p+1} \in \mathbb{R}^p$ for the initialization of the parameter set. For $i = 1, \dots, p$, the estimate θ_i is obtained from θ_{p+1} by displacing its i-th component by a user-definable nonzero number $h_i \in \mathbb{R}$ times the corresponding component of θ_{p+1}. This type of parameter generation enables the user to construct p additional linearly independent initial vectors. Then, the algorithm proceeds by computing the response $f(\theta_i)$ and subsequently the residual sum of squares $Q(\theta_i)$ for each θ_i, $i = 1, \dots, p + 1$. Finally, the subscripts of these vectors are relabeled according to the cost of the estimates, i.e., so that

$$Q(\theta_1) \geq \cdots \geq Q(\theta_{p+1}). \tag{4.41}$$

That is, the initial parameter set is ordered from the worst to the best estimate.

Remark. We mentioned earlier that the order of subscripts is from the oldest to the youngest in an iteration of DUD. That is, in subsequent iterations, the parameters are not reordered in the sense of (4.41). Usually, the oldest parameter θ_1 gets replaced by θ_{new}, which is then relabeled as θ_{p+1}. The subscripts of the remaining parameters are simply reduced by one, which consequently defines the new parameter set. We recall that θ_{new} ideally does but needs not reduce the RSS. Hence, we notice that relationship (4.41) may not hold in an arbitrary iteration of the algorithm.

We now elaborate the exact updating strategy for the parameter set. The parameter vector differences $\theta_i - \theta_{p+1}$, $i = 1, \ldots, p$, used in (4.35) must be linearly independent in order to ensure that the search does not collapse into a subplane of the parameter space. Ralston and Jennrich [44] argue that, "Theoretically, if the current set of parameter vector differences span the parameter space, then the new set will span it also, if and only if the component of α corresponding to the discarded parameter vector is nonzero."
Most of the time, θ_{new} replaces the oldest estimate θ_1. However, if the component of α corresponding to θ_1 is close to being zero, two parameters of the set are replaced. More precisely, if $|\alpha_1| < 10^{-5}$ we first replace θ_i by θ_{new}, where i is the first subscript for which $|\alpha_i| \geq 10^{-5}$. Secondly, old members of the set are not retained indefinitely, so θ_1 is replaced by the estimate $(\theta_1 + \theta_{new})/2$.

DUD repeats its iterations until a stopping criterion is satisfied, e.g., until the cost function $Q(\theta_{new})$ is sufficiently small. We can summarize the algorithm as follows:

1. Generate p more vectors $\theta_1, \ldots, \theta_p$ from the initial parameter θ_{p+1}.
2. Relabel the subscripts of these estimates so that $Q(\theta_1) \geq \cdots \geq Q(\theta_{p+1})$.
3. Generate the matrices ΔF and $\Delta\Theta$ by computing
$$\Delta F_{:i} = f(\theta_i) - f(\theta_{p+1}) \quad \text{and} \quad \Delta\Theta_{:i} = \theta_i - \theta_{p+1}, \quad i = 1, \ldots, p.$$
4. Compute $\alpha = \big(\Delta F^{\mathsf{T}} \Delta F\big)^{-1} \Delta F^{\mathsf{T}} \big(y - f(\theta_{p+1})\big)$. Use a stepwise regression modification of the Gauss-Jordan algorithm for matrix inversion in order to prevent a full inversion of the matrix $\Delta F^{\mathsf{T}} \Delta F$ if it is essentially singular.
5. Set $\theta_{new} = \theta_{p+1} + \Delta\Theta\alpha$. Optionally, if $Q(\theta_{new}) \geq Q(\theta_{p+1})$, attempt to find a better estimate for θ_{new} by employing the described line search.
6. Terminate the algorithm if a stopping criterion is met. Else, replace the oldest estimate θ_1 by θ_{new}, relabel the parameters according to their age and go to 3. Exception: If $|\alpha_1| < 10^{-5}$, first replace θ_i by θ_{new}, where i is the first index with $|\alpha_i| \geq 10^{-5}$, and secondly replace θ_1 by $(\theta_1 + \theta_{new})/2$.

This is just a very rough outline of the procedure. The following pseudocode provides a better insight on how to realize the implementation of DUD. Further important notes can be found in Section 4.3.3, where the implementation details are given. Algorithm 5 (DUD) treats the current parameter set simply as matrix $\Theta \in \mathbb{R}^{p \times (p+1)}$ whose $p + 1$ columns represent the parameter vectors $\theta_1, \ldots, \theta_{p+1}$. Accordingly, the corresponding $p + 1$ function evaluations are then also stored in the columns of a matrix, namely $F \in \mathbb{R}^{n \times (p+1)}$. Step 21 of Algorithm 5 refers to Algorithm 7, which is presented in Section 4.3.2. The step shortening procedure (steps 29–34) is only executed if the user enters a positive integer, which determines the possible number of searches (and thus iterations). The algorithm terminates if convergence occurs or if a user specified maximum number of iterations is reached. DUD can also be used to solve standard NLLS problems of the form (3.3) by simply observing that (4.33) is

$$Q(\theta) = \big\| f(\theta) \big\|_2^2$$

when no data is given.

Algorithm 5 DUD

Input: function $f(x, \theta)$, initial parameter θ_{p+1}, vectors x and y given by the set of data $(x_i, y_i),\ i = 1, \ldots, n$, maximal number m of line searches per iteration, vector h for additional parameter generation, termination parameters tol, $maxit$

Output: approximate solution of data fitting problem, i.e., minimizer of RSS of the form (3.12); or, if no data (x, y) has been entered, approximate solution of standard NLLS problem

Require: $\theta_{p+1} \in \mathbb{R}^p$, $h \in \mathbb{R}^p$, $h_i \in \mathbb{R} \setminus \{0\}\ \forall i$, $m \in \mathbb{N}_0$, $maxit \in \mathbb{N}_0$, $0 < tol \ll 1$

Ensure: prevent full inversion of the matrix $\Delta F^\mathsf{T} \Delta F$ if it is essentially singular, the parameters span the parameter space and should not remain indefinitely in the parameter set - discard the oldest member in each iteration

$p \leftarrow$ dim. of θ_{p+1};
$n \leftarrow$ dim. of $f(x, \theta)$;
$Q \leftarrow (p+1)$-dim. zero vector; ▷ storage for ... ▷ ... different RSS
$F \leftarrow n \times (p+1)$ zero matrix; ▷ ... function evaluation
$\Delta F \leftarrow n \times p$ zero matrix; ▷ ... function differences
$\Delta\Theta \leftarrow p \times p$ zero matrix; ▷ ... parameter differences
$\Theta \leftarrow p \times (p+1)$ matrix where each column is θ_{p+1};
▷ generate p more initial parameters by displacing the i-th component of θ_{p+1}
for $i = 1$ to p **do**
    $\Theta_{ii} = \Theta_{ii} + h_i$; ▷
end for
▷ columns of Θ are the $p+1$ parameter vectors needed for starting
▷ columns of F get the $p+1$ function evaluations needed for starting
for $i = 1$ to $p+1$ **do**
    evaluate $F_{:i} = f(x, \Theta_{:i})$;
    compute $Q_i = \|y - F_{:i}\|_2^2$; ▷ RSS for $\Theta_{:i}$ is stored in Q_i
end for
sort the entries of Q such that $Q_1 \geq \cdots \geq Q_{p+1}$;
rearrange the columns of Θ and F according to the sorting order for Q;
▷ the parameters $\Theta_{:i},\ i = 1, \ldots, p+1$, are now arranged from worst to best, i.e., so that $\Theta_{:1}$ corresponds to the highest RSS and $\Theta_{:p+1}$ to the lowest
$\theta_{p+1} = \Theta_{:p+1}$;
▷ generate the matrix ΔF of function differences
for $i = 1$ to p **do**
    $\Delta F_{:i} = F_{:i} - F_{:p+1}$;
end for
invoke Algorithm 7 to compute $\alpha = \left(\Delta F^\mathsf{T} \Delta F\right)^{-1} \Delta F^\mathsf{T} (y - F_{:p+1})$;
▷ the stepwise regression procedure prevents the inversion of the matrix $\Delta F^\mathsf{T} \Delta F$ if it is essentially singular and computes an estimate of α
▷ generate the matrix $\Delta\Theta$ of parameter differences
for $i = 1$ to p **do**
    $\Delta\Theta_{:i} = \Theta_{:i} - \theta_{p+1}$;
end for
compute $\theta_{new} = \theta_{p+1} + \Delta\Theta\alpha$; ▷ new parameter estimate
evaluate $F_{new} = f(x, \theta_{new})$;
compute $Q_{new} = \|y - F_{new}\|_2^2$;
$l = 1$;

Algorithm 5 DUD

29: **if** $Q_{new} \geq Q_{p+1}$ and $m \geq l$ **then** ▷ line search (requires $m \neq 0$)
30: $d = -(-1/2)^l$;
31: $\theta_{new} = d\theta_{new} + (1-d)\theta_{p+1}$;
32: perform steps 26 and 27;
33: $l = l + 1$;
34: **end if**
35: $k = 1$;
▷ the algorithm stops now if convergence has been achieved, i.e., if $Q_{new} < tol$
▷ the minimum number of function evaluations at this point is $p + 2$
36: **while** $k \leq maxit$ and $Q_{new} \geq tol$ **do**
▷ update the parameter matrix Θ (columns are numbered by age with $\Theta_{:p+1}$ being the newest member of the parameter set)
37: **if** $|\alpha_1| \geq 10^{-5}$ **then** ▷ replace oldest member of Θ
38: $\Theta_{:1} = \theta_{new}$;
39: $F_{:1} = F_{new}$;
40: $Q_1 = Q_{new}$;
41: **else**
42: find first index s with $|\alpha_s| \geq 10^{-5}$;
43: **if** no such s exists **then** ▷ replace oldest member of Θ
44: compute $\Theta_{:1} = (\Theta_{:1} + \theta_{new})/2$;
45: evaluate $F_{:1} = f(x, \Theta_{:1})$;
46: compute $Q_1 = \|y - F_{:1}\|_2^2$;
47: **else** ▷ replace two members of Θ:
48: $\Theta_{:s} = \theta_{new}$; ▷ first at column s ...
49: $F_{:s} = F_{new}$;
50: $Q_s = Q_{new}$;
51: shift columns as follows: $\Theta_{:p+1} \leftarrow \Theta_{:s}$ and $\Theta_{:i} \leftarrow \Theta_{:i+1}$ for $i = s, \ldots, p$;
52: rearrange F and Q according to the shift;
53: perform steps 44–46; ▷ ... and then the oldest
54: **end if**
55: **end if**
56: shift columns as follows: $\Theta_{:p+1} \leftarrow \Theta_{:1}$ and $\Theta_{:i} \leftarrow \Theta_{:i+1}$ for $i = 1, \ldots, p$;
57: rearrange F and Q according to the shift;
58: perform steps 17–34; ▷ compute estimate θ_{new}
59: $k = k + 1$;
60: **end while**
61: **return** θ_{new};

4.3.1 Gauss-Jordan inversion

Before we introduce the stepwise regression procedure [22] employed by DUD, we need to understand the underlying Gauss-Jordan algorithm for matrix inversion. If we want to invert a square matrix A with the classical Gauss-Jordan elimination method, we have to augment A with the identity matrix I. This is not efficient since additional storage is required and the computations must be performed on both matrices. Here, we present an algorithm that simply overwrites A with A^{-1}, which we call *in place* or *in situ* matrix inversion. Because no augmentation is needed, both memory and arithmetic operations can be saved.

Let $A = (a_{ij}) \in \mathbb{R}^{n\times n}$, $b \in \mathbb{R}^n$ and suppose that we want to solve the linear system

$$Ax = b \tag{4.42}$$

for $x \in \mathbb{R}^n$. If $a_{11} \neq 0$, then we can solve the first equation of (4.42) for x_1 and insert the result into the remaining equations. This yields the linear system

$$\begin{aligned} \tilde{a}_{11}b_1 + \tilde{a}_{12}x_2 + \cdots + \tilde{a}_{1n}x_n &= x_1 \\ \tilde{a}_{21}b_1 + \tilde{a}_{22}x_2 + \cdots + \tilde{a}_{2n}x_n &= b_2 \\ &\vdots \\ \tilde{a}_{n1}b_1 + \tilde{a}_{n2}x_2 + \cdots + \tilde{a}_{nn}x_n &= b_n, \end{aligned} \tag{4.43}$$

where

$$\begin{aligned} &\tilde{a}_{11} = 1/a_{11}, \quad \tilde{a}_{1j} = -a_{1j}/a_{11}, \quad \tilde{a}_{i1} = a_{i1}/a_{11}, \\ &\tilde{a}_{ij} = a_{ij} - a_{i1}a_{1j}/a_{11}, \quad i,j = 2,\ldots,n. \end{aligned} \tag{4.44}$$

In the next step, provided that $\tilde{a}_{22} \neq 0$, we can solve the second equation in (4.43) for x_2, which enables us to exchange the variables x_2 and b_2 in the other equations. If we perform n such steps, we are left with the linear system

$$\bar{A}b = x,$$

where $\bar{A}$ denotes the transformed matrix. But this ultimately means that $\bar{A} = A^{-1}$, i.e., the entries of the inverse matrix can be computed by performing n steps of the form (4.44). This motivates the following definition, where (i,j) refers to the entry in the i-th row and j-th column of some matrix.

Definition 4.8 (Gauss-Jordan pivot)**.** Suppose that $A = (a_{ij})$ is a square matrix with nonzero k-th diagonal element, i.e., $a_{kk} \neq 0$. Then, performing a *Gauss-Jordan pivot*, or simply *pivoting* on a_{kk} or (k,k) results in a matrix $\tilde{A}$ whose ij-th element is given by

$$\tilde{a}_{ij} = \begin{cases} 1/a_{kk} & i = k,\ j = k \\ -a_{ik}/a_{kk} & i \neq k,\ j = k \\ a_{kj}/a_{kk} & i = k,\ j \neq k \\ a_{ij} - a_{ik}a_{kj}/a_{kk} & i \neq k,\ j \neq k \end{cases}.$$

The element a_{kk} of the matrix A before pivoting is referred to as the *pivot*.

Remark. Definition 4.8 can be easily generalized so that performing a Gauss-Jordan pivot on any nonzero entry of some arbitrary matrix is possible, see [2, p. 296]. However, this is not needed for our purpose. The Gauss-Jordan pivot applied to the main diagonal (Definition 4.8) enjoys practicability in regression (see Section 4.3.2) and is commonly known as *sweep operator* [3] in statistics. Pivoting on (k, k) is therefore also referred to as pivoting or *sweeping* row k. However, we note that the definition of the sweep operator is not consistent in the literature. Variations and extensions of it can be found in [16].

The following two properties can be easily verified with Definition 4.8:

- A Gauss-Jordan pivot is its own inverse, i.e., pivoting on (k, k) twice leaves the matrix unchanged.
- Gauss-Jordan pivots commute, i.e., provided that $i \neq j$, pivoting first on (i, i) and then on (j, j) produces the same matrix as pivoting first on (j, j) and then on (i, i).

For further analysis of Gauss-Jordan pivoting, we provide statements that are similar to those of the slightly different exposition of the sweep operator in [26, pp. 96–98].

Proposition 4.9. *Suppose that $V = UA$, where $A \in \mathbb{R}^{n\times n}$ and $U, V \in \mathbb{R}^{m\times n}$. Then*

$$\hat{V} = \hat{U}\tilde{A}$$

after performing a Gauss-Jordan pivot on the k-th diagonal element of A, where

(i) $\hat{U}$ coincides with U except that its k-th column $\hat{U}_{:k}$ is $V_{:k}$,

(ii) $\hat{V}$ coincides with V except that its k-th column $\hat{V}_{:k}$ is $U_{:k}$.

Proof. By assumption, the entries of V have the form

$$v_{jl} = \sum_{i=1}^{n} u_{ji} a_{il},$$

where $j = 1, \ldots, m$ and $l = 1, \ldots, n$. In particular, we find for the k-th column that

$$v_{jk} = u_{jk} a_{kk} + \sum_{i\neq k} u_{ji} a_{ik}.$$

After pivoting on a_{kk},

$$\begin{aligned}
\hat{v}_{jk} &\overset{(ii)}{=} u_{jk} \\
&= \frac{1}{a_{kk}} \left(v_{jk} - \sum_{i\neq k} u_{ji} a_{ik} \right) \\
&\overset{(i)}{=} \hat{u}_{jk} \tilde{a}_{kk} + \sum_{i\neq k} \hat{u}_{ji} \tilde{a}_{ik} \\
&= \sum_{i=1}^{n} \hat{u}_{ji} \tilde{a}_{ik}.
\end{aligned}$$

And for $l \neq k$,

$$\begin{aligned}
\hat{v}_{jl} &\overset{(ii)}{=} v_{jl} \\
&= u_{jk}a_{kl} + \sum_{i\neq k} u_{ji}a_{il} \\
&= \frac{1}{a_{kk}}\left(v_{jk} - \sum_{i\neq k} u_{ji}a_{ik}\right)a_{kl} + \sum_{i\neq k} u_{ji}a_{il} \\
&= v_{jk}\frac{a_{kl}}{a_{kk}} + \sum_{i\neq k} u_{ji}a_{il} - \sum_{i\neq k} u_{ji}a_{ik}\frac{a_{kl}}{a_{kk}} \\
&\overset{(i)}{=} \hat{u}_{jk}\tilde{a}_{kl} + \sum_{i\neq k} \hat{u}_{ji}\tilde{a}_{il} \\
&= \sum_{i=1}^{n} \hat{u}_{ji}\tilde{a}_{il}.
\end{aligned}$$

Hence, it holds that $\hat{V} = \hat{U}\tilde{A}$. □

Remark. We observe that if we pivot twice on (k, k), then again $V = UA$.

In the following, we denote by $\bar{A}$ the matrix which results from pivoting on a number of diagonal entries of some original matrix A.

Proposition 4.10. *Let the matrix $A \in \mathbb{R}^{n\times n}$ be partitioned as*

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix},$$

where $A_{11} \in \mathbb{R}^{m\times m}$, $m < n$. If it is possible to pivot on all diagonal entries of A_{11} (meaning that no zeros are encountered), then the matrix A_{11} is nonsingular and the result of pivoting on all of its diagonal entries is given by

$$\bar{A} = \begin{bmatrix} A_{11}^{-1} & A_{11}^{-1}A_{12} \\ -A_{21}A_{11}^{-1} & A_{22} - A_{21}A_{11}^{-1}A_{12} \end{bmatrix}.$$

In particular, if a matrix is pivoted on all of its diagonal entries, the result is its inverse.

Proof. Pivoting on each diagonal entry of A_{11} once corresponds to repeatedly (in fact m times) applying Proposition 4.9 to the matrix identity

$$\underbrace{\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}}_{V} = \underbrace{\begin{bmatrix} I_{11} & 0_{12} \\ 0_{21} & I_{22} \end{bmatrix}}_{U}\underbrace{\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}}_{A},$$

which leads to

$$\begin{bmatrix} I_{11} & A_{12} \\ 0_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & 0_{12} \\ A_{21} & I_{22} \end{bmatrix}\underbrace{\begin{bmatrix} \bar{A}_{11} & \bar{A}_{12} \\ \bar{A}_{21} & \bar{A}_{22} \end{bmatrix}}_{\bar{A}}.$$

This implies the equations

$$\begin{aligned} I_{11} &= A_{11}\bar{A}_{11}, \\ A_{12} &= A_{11}\bar{A}_{12}, \\ 0_{12} &= A_{21}\bar{A}_{11} + \bar{A}_{21}, \\ A_{22} &= A_{21}\bar{A}_{12} + \bar{A}_{22}. \end{aligned}$$

Now, solving for the submatrices yields the claimed result

$$\begin{bmatrix} \bar{A}_{11} & \bar{A}_{12} \\ \bar{A}_{21} & \bar{A}_{22} \end{bmatrix} = \begin{bmatrix} A_{11}^{-1} & A_{11}^{-1}A_{12} \\ -A_{21}A_{11}^{-1} & A_{22} - A_{21}A_{11}^{-1}A_{12} \end{bmatrix}.$$

□

Remark. Proposition 4.10 shows that pivoting on a matrix in block form follows the same rules as pivoting on the matrix elementwise (cf. Definition 4.8). Furthermore, if a matrix is singular, then for any order of pivoting at least one diagonal entry becomes zero before all diagonal entries have been used as pivots.

The Gauss-Jordan in-place matrix inversion algorithm may be written as follows. Let $A \in \mathbb{R}^{n\times n}$ be the matrix to be inverted. For each pivot row $k = 1, \ldots, n$ repeat:

1. Let $a = a_{kk}$ be the k-th diagonal element.
2. Divide the k-th row by a.
3. For every other row $i \neq k$, let $b = a_{ik}$ be the i-th element of the k-th column. Subtract $b \times$ row k from row i. Then set $a_{ik} = -b/a$.
4. Set $a_{kk} = 1/a$.

An important issue in Gauss-Jordan elimination is the nonzero requirement of the current pivot. If it were zero, carrying out steps 2–4 of the algorithm would cause the procedure to fail. If the pivot is nonzero but very small, then performing the divisions usually results in significant rounding error which prevents the inversion from being successful. In order to cope with this problem, we may use some strategy for changing the pivoting order of the algorithm. Problematic pivots can be detected by checking the diagonal elements of a matrix for their tolerance, see Definition 4.11.

Definition 4.11. Let $A \in \mathbb{R}^{n\times n}$ and denote by $\bar{a}_{kk}$ the value of the k-th diagonal entry of the matrix after pivoting on a number of other diagonal entries. We call the ratio $\bar{a}_{kk}/a_{kk}$ the *tolerance* of the k-th diagonal element at this state.

Remark. The tolerance of the k-th diagonal element, in a sense, measures its precision since $\bar{a}_{kk}$ is obtained from a_{kk} by a sequence of operations. The precision of the entire matrix may be destroyed if an element with low tolerance is chosen as pivot.

Since Gauss-Jordan pivots commute, we can change the order of pivoting to increase the stability of the algorithm. If we select pivots with larger tolerance first, the numerical precision of the matrix may be conserved throughout the process.

Algorithm 6 employs the *partial pivoting* strategy. That is, in each iteration the pivot is selected as the diagonal element that has the largest absolute value among all unpivoted diagonal elements. Even though this increases the numerical stability, the algorithm is clearly not capable of handling matrices whose diagonal is zero. Nevertheless, it performs well for diagonally dominant, positive definite and negative definite matrices.

Algorithm 6 Gauss-Jordan in-place matrix inversion

Input: square matrix A
Output: inverse of A
Require: A is nonsingular and its diagonal contains nonzero entries,
the condition number of A must not be large, e.g., $\kappa(A) < 1/\sqrt{eps}$
Ensure: select the unpivoted diagonal entry that has largest absolute value as pivot

1: $n \leftarrow$ number of rows/columns of A;
2: $\mathcal{R} \leftarrow \{1, \ldots, n\}$ ▷ set of indices for unpivoted rows
3: **for** $j = 1$ to n **do**
4: $k \leftarrow$ index that corresponds to $\max_{r \in \mathcal{R}} |A_{rr}|$;
5: remove index k from $\mathcal{R}$;
6: $a = A_{kk}$;
7: $A_{k:} = A_{k:}/a$; ▷ divide row k by A_{kk}
8: **for** $i = 1$ to n **do**
9: **if** $i \neq k$ **then** ▷ for every row $i \neq k$
10: $b = A_{ik}$;
11: $A_{i:} = A_{i:} - b \cdot A_{k:}$; ▷ subtract $A_{ik} \times$ row k from row i
12: $A_{ik} = -b/a$; ▷ set $A_{ik} = -A_{ik}/A_{kk}$
13: **end if**
14: **end for**
15: $A_{kk} = A_{kk}/a$; ▷ set $A_{kk} = 1/A_{kk}$
16: **end for**
17: **return** A; ▷ A is now A^{-1}

This algorithm performs n Gauss-Jordan pivots, each of which requires $\mathcal{O}(2n^2)$ arithmetic operations. That is, the total associated work is $\mathcal{O}\big(2n^3\big)$. Hence, it performs roughly 4/3 times faster than Gaussian elimination with augmentation and backward substitution, which costs approximately $\mathcal{O}\big(\frac{8}{3}n^3\big)$, see [5, p. 183]. And we recall that it just requires a single matrix for storing both the coefficient matrix and its inverse.

Remark. Bauer and Reinsch [50, p. 45–49] provided an efficient in situ Gauss-Jordan algorithm for the inversion of positive definite matrices. It exploits the symmetry of the matrix by operating only on its lower triangular part, which is stored row by row in a separate working vector. Hence, it requires approximately only one-half of the work of Algorithm 6. However, it fails when it encounters pivot zeros and has a predefined pivoting order. Therefore, it is not suited as basis for stepwise regression.

The matrix $\Delta F^\mathsf{T}\Delta F$ in equation (4.38) of an iteration of DUD is positive semidefinite per construction but may not be positive definite, i.e., not invertible. Furthermore, we recall equation (3.10) and note that the condition number of $\Delta F^\mathsf{T}\Delta F$ is the square of the condition of ΔF, so any ill-conditioning is exacerbated. However, if $\Delta F^\mathsf{T}\Delta F$ is singular or ill-conditioned, we still want to be able to provide an estimate for $\hat{\alpha}$ in (4.38). This can be achieved by modifying Algorithm 6 so that the pivoting follows the order given by stepwise regression and by letting elements with low tolerance remain unpivoted, which consequently prevents the matrix from being fully inverted.

4.3.2 Stepwise regression

In this section, we present stepwise regression based on Gauss-Jordan elimination as proposed in [22]. Suppose that we are given a linear regression model of the form (3.16) with $n > p$. That is, $y \in \mathbb{R}^n$ denotes the dependent variable and $X \in \mathbb{R}^{n\times p}$ is a design matrix whose columns $X_{:i}$ are the independent variables x_i, $i = 1, \ldots, n$. Then, regressing on y on the set of variables $x_1, \ldots, x_p$ refers to the linear relationship

$$y \approx \beta_1 x_1 + \beta_2 x_2 + \ldots \beta_p x_p,$$

where $\beta \in \mathbb{R}^p$ (cf. (3.15)). The goal is to find an estimate for the β which minimizes

$$\left\|y - X\beta\right\|_2^2, \tag{4.45}$$

the residual sum of squares. For this purpose, let us form the $(p+1)\times(p+1)$ matrix

$$C := \begin{bmatrix} X^\mathsf{T}X & X^\mathsf{T}y \\ y^\mathsf{T}X & y^\mathsf{T}y \end{bmatrix}. \tag{4.46}$$

If we assume that X has full column rank, $X^\mathsf{T}X$ is nonsingular and, according to Proposition 4.10, pivoting on each of its diagonal entries results in the matrix

$$\bar{C} = \begin{bmatrix} (X^\mathsf{T}X)^{-1} & (X^\mathsf{T}X)^{-1}X^\mathsf{T}y \\ -y^\mathsf{T}X(X^\mathsf{T}X)^{-1} & y^\mathsf{T}y - y^\mathsf{T}X(X^\mathsf{T}X)^{-1}X^\mathsf{T}y \end{bmatrix}. \tag{4.47}$$

Hence, apart from calculating the inverse of the matrix of cross products $X^\mathsf{T}X$, pivoting allows for the simultaneous computation of the unique OLS estimator of β,

$$\hat{\beta} = (X^\mathsf{T}X)^{-1}X^\mathsf{T}y,$$

and its corresponding RSS, as

$$\begin{aligned} \left\|y - X\hat{\beta}\right\|_2^2 &= (y - X\hat{\beta})^\mathsf{T}(y - X\hat{\beta}) \\ &= (y - X(X^\mathsf{T}X)^{-1}X^\mathsf{T}y)^\mathsf{T}(y - X(X^\mathsf{T}X)^{-1}X^\mathsf{T}y) \\ &= y^\mathsf{T}y - y^\mathsf{T}X(X^\mathsf{T}X)^{-1}X^\mathsf{T}y - y^\mathsf{T}X(X^\mathsf{T}X)^{-1}X^\mathsf{T}y \\ &\quad + y^\mathsf{T}X(X^\mathsf{T}X)^{-1}\underbrace{X^\mathsf{T}X(X^\mathsf{T}X)^{-1}}_{=I}X^\mathsf{T}y \\ &= y^\mathsf{T}y - y^\mathsf{T}X(X^\mathsf{T}X)^{-1}X^\mathsf{T}y. \end{aligned} \tag{4.48}$$

Furthermore, we observe that the term in the lower left part of the matrix $\bar{C}$ is $-\hat{\beta}^{\mathsf{T}}$.

Remark. We emphasize that, after pivoting on all diagonal entries of $X^{\mathsf{T}}X$, the vector $\hat{\beta}$ contains the regression coefficients $\hat{\beta}_i$ for the variables x_i, $i = 1, \ldots, p$. That is, we find the linear regression model

$$\hat{y} = \hat{\beta}_1 x_1 + \cdots + \hat{\beta}_p x_p$$

and

$$\left\|y - \hat{y}\right\|_2^2 = \left\|y - X\hat{\beta}\right\|_2^2$$

is the residual sum of squares which results from regressing y on <u>all</u> the x-variables.

Now, if $X^{\mathsf{T}}X$ is singular (or ill-conditioned), it cannot be converted (with satisfactory numerical accuracy) but we may still be able to find an adequate estimate for β through pivoting. We simply utilize the fact that we do not need to regress y on the whole set of x-variables, namely X, in order to achieve a satisfying regression fit.
In *stepwise regression* we select one variable at a time from the set of x-variables X to add to (or remove from) the regression model. The Gauss-Jordan algorithm is particularly well-suited for this technique since pivoting on the i-th diagonal entry of $X^{\mathsf{T}}X$ corresponds to adding the variable x_i, $i = 1, \ldots, p$, to the regression. Hence, since a Gauss-Jordan pivot is its own inverse, pivoting again on the i-th diagonal entry removes the corresponding predictor from the model. Let us illustrate this principle with the following example.

Example. Suppose that we have 2 independent variables x_1 and x_2, i.e., $X \in \mathbb{R}^{n\times 2}$, and that ${x_1}^{\mathsf{T}}x_1 \neq 0$. We start by forming the matrix

$$\begin{bmatrix} {x_1}^{\mathsf{T}}x_1 & {x_1}^{\mathsf{T}}x_2 & {x_1}^{\mathsf{T}}y \\ {x_2}^{\mathsf{T}}x_1 & {x_2}^{\mathsf{T}}x_2 & {x_2}^{\mathsf{T}}y \\ y^{\mathsf{T}}x_1 & y^{\mathsf{T}}x_2 & y^{\mathsf{T}}y \end{bmatrix}$$

and want to regress the dependent variable y on x_1. This can be achieved by performing a Gauss-Jordan pivot on the first diagonal entry $(1, 1)$, which results in the matrix

$$\begin{bmatrix} ({x_1}^{\mathsf{T}}x_1)^{-1} & ({x_1}^{\mathsf{T}}x_1)^{-1}{x_1}^{\mathsf{T}}x_2 & ({x_1}^{\mathsf{T}}x_1)^{-1}{x_1}^{\mathsf{T}}y \\ -{x_2}^{\mathsf{T}}x_1({x_1}^{\mathsf{T}}x_1)^{-1} & {x_2}^{\mathsf{T}}x_2 - {x_2}^{\mathsf{T}}x_1({x_1}^{\mathsf{T}}x_1)^{-1}{x_1}^{\mathsf{T}}x_2 & {x_2}^{\mathsf{T}}y - {x_2}^{\mathsf{T}}x_1({x_1}^{\mathsf{T}}x_1)^{-1}{x_1}^{\mathsf{T}}y \\ -y^{\mathsf{T}}x_1({x_1}^{\mathsf{T}}x_1)^{-1} & y^{\mathsf{T}}x_2 - y^{\mathsf{T}}x_1({x_1}^{\mathsf{T}}x_1)^{-1}{x_1}^{\mathsf{T}}x_2 & y^{\mathsf{T}}y - y^{\mathsf{T}}x_1({x_1}^{\mathsf{T}}x_1)^{-1}{x_1}^{\mathsf{T}}y \end{bmatrix}.$$

If we treat x_2, the variable that is not added to the regression, as dependent variable, then this matrix may be rewritten as

$$\begin{bmatrix} ({x_1}^{\mathsf{T}}x_1)^{-1} & \hat{\beta}_{x_2|x_1} & \hat{\beta}_{y|x_1} \\ -\hat{\beta}^{\mathsf{T}}_{x_2|x_1} & \|x_2 - \hat{x}_2\|_2^2 & (x_2 - \hat{x}_2)^{\mathsf{T}}(y - \hat{y}) \\ -\hat{\beta}^{\mathsf{T}}_{y|x_1} & (y - \hat{y})^{\mathsf{T}}(x_2 - \hat{x}_2) & \|y - \hat{y}\|_2^2 \end{bmatrix}, \tag{4.49}$$

where, for example, $\hat{\beta}_{y|x_1} = ({x_1}^{\mathsf{T}}x_1)^{-1}{x_1}^{\mathsf{T}}y$ and $\hat{y} = x_1\hat{\beta}_{y|x_1} = x_1({x_1}^{\mathsf{T}}x_1)^{-1}{x_1}^{\mathsf{T}}y$, which shows that we regressed y on the variable x_1. We may now also add x_2 to the model in order to expand the regression to $y \approx x_1\beta_1 + x_2\beta_2$. Provided that X is nonsingular, pivoting on the corresponding entry $(2, 2)$ of the above matrix yields

$$\begin{bmatrix} \begin{bmatrix} {x_1}^\mathsf{T} x_1 & {x_1}^\mathsf{T} x_2 \\ {x_2}^\mathsf{T} x_1 & {x_2}^\mathsf{T} x_2 \end{bmatrix}^{-1} & \hat{\beta}_{y|x_1x_2} \\ -{\hat{\beta}^\mathsf{T}}_{y|x_1x_2} & \|y - \hat{y}\|^2 \end{bmatrix},$$

which is of the form (4.47) and where $\hat{y} = X\hat{\beta}_{y|x_1x_2}$ is now the expanded model. If X were singular, the diagonal entry $(2,2)$ of the matrix (4.49) would be zero and, consequently, we could not add x_2 to the regression.

Remark. This example shows that it makes sense to consider all independent variables that are not currently in the regression equation as dependent variables. Furthermore, at each step, the current regression coefficients and the corresponding residual sum of squares are immediately available. A step is easily reversed by pivoting again.

For our goal, we start with the matrix C (4.46) and proceed by adding variables to the regression only, which is called *forward selection*. In *backward elimination*, one would start with the matrix $\bar{C}$ (4.47), which corresponds to the model with all predictors, and would proceed by successively removing the variables from the regression.

In the first step of the stepwise regression procedure, we choose the independent variable for which we want to start the regression with, say x_k. That is, we pivot on the k-th diagonal element of the matrix C which results in the matrix $\tilde{C}$, where

$$\tilde{c}_{ij} = \begin{cases} 1/c_{kk} & i = k,\ j = k \\ -c_{ik}/c_{kk} & i \neq k,\ j = k \\ c_{kj}/c_{kk} & i = k,\ j \neq k \\ c_{ij} - c_{ik}c_{kj}/c_{kk} & i \neq k,\ j \neq k \end{cases}.$$

The elements $\tilde{c}_{k,p+1}$ and $\tilde{c}_{p+1,p+1}$ are the regression coefficient and the RSS for predicting the dependent variable y from x_k, respectively.
In step two, we choose a different independent variable x_l, $l \neq k$, and pivot on the l-th diagonal element of $\tilde{C}$. At this stage, the entry $(p+1, p+1)$ of the new matrix contains the RSS from regressing y on both the variables x_k and x_l; the entries $(k, p+1)$ and $(l, p+1)$ are the regression coefficients for predicting y from x_k and x_l, respectively.

At an arbitrary step, we suppose that Gauss-Jordan pivots have been performed for the variables $x_1, \ldots, x_r$. That is, according to Proposition 4.10, we find the matrix

$$\bar{C} = \begin{bmatrix} A & B \\ -B^\mathsf{T} & S \end{bmatrix} \tag{4.50}$$

with $A \in \mathbb{R}^{r\times r}$, $B \in \mathbb{R}^{r\times q}$ and $S \in \mathbb{R}^{q\times q}$, where $q = p - r + 1$. More precisely:

- A is the inverse of the cross products matrix of the variables that have been added to the regression.

- B contains the regression coefficients for predicting the variables $x_{r+1}, \ldots, x_p$ and y from the independent variables $x_1, \ldots, x_r$. That is, B has the form

$$B = \begin{bmatrix} \hat{\beta}_{x_{r+1}|x_1 \ldots x_r} & \ldots & \hat{\beta}_{x_p|x_1 \ldots x_r} & \hat{\beta}_{y|x_1 \ldots x_r} \end{bmatrix},$$

  where, for example, the vector $\hat{\beta}_{y|x_1 \ldots x_r}$ contains the r coefficients for regressing the dependent variable y on the first r variables.

- S is the sum of squares and of cross products (SSCP) matrix of residuals for the unpivoted variables regressed on the variables that have been pivoted, i.e.,

$$S = \begin{bmatrix} (x_{r+1} - \hat{x}_{r+1})^\mathsf{T}(x_{r+1} - \hat{x}_{r+1}) & \ldots & (x_{r+1} - \hat{x}_{r+1})^\mathsf{T}(x_p - \hat{x}_p) & (x_{r+1} - \hat{x}_{r+1})^\mathsf{T}(y - \hat{y}) \\ \vdots & \ddots & \vdots & \vdots \\ (x_p - \hat{x}_p)^\mathsf{T}(x_{r+1} - \hat{x}_{r+1}) & \ldots & (x_p - \hat{x}_p)^\mathsf{T}(x_p - \hat{x}_p) & (x_p - \hat{x}_p)^\mathsf{T}(y - \hat{y}) \\ (y - \hat{y})^\mathsf{T}(x_{r+1} - \hat{x}_{r+1}) & \ldots & (y - \hat{y})^\mathsf{T}(x_p - \hat{x}_p) & (y - \hat{y})^\mathsf{T}(y - \hat{y}) \end{bmatrix}.$$

  The diagonal of S contains the current RSS for each such unpivoted variable. In particular, the last diagonal element is the RSS of the dependent variable y with

$$\hat{y} = \begin{bmatrix} x_1 & \ldots & x_r \end{bmatrix} \hat{\beta}_{y|x_1 \ldots x_r}. \tag{4.51}$$

We want to add the independent variable to the regression that provides the most good for the prediction of y. This may be achieved by selecting the variable that provides the greatest possible reduction in the RSS of y at the current stage.
The new regression model that would ensue if we add the independent variable x_k, $k \in \{r+1, \ldots, p\}$, to the current prediction (4.51) of y may be written as

$$\hat{y}_{new} = \hat{y} + (x_k - \hat{x}_k)\hat{\beta}_{new},$$

where $\hat{\beta}_{new} = \big((x_k - \hat{x}_k)^\mathsf{T}(x_k - \hat{x}_k)\big)^{-1}(x_k - \hat{x}_k)^\mathsf{T}(y - \hat{y})$ arises after pivoting and constitutes the last entry of the vector of new regression coefficients $\hat{\beta}_{y|x_1 \ldots x_r x_k} \in \mathbb{R}^{r+1}$. Then

$$\begin{aligned} \big\|y - \hat{y}_{new}\big\|_2^2 &= \big\|y - (\hat{y} + (x_k - \hat{x}_k)\hat{\beta}_{new})\big\|_2^2 \\ &= \big\|(y - \hat{y}) - (x_k - \hat{x}_k)\hat{\beta}_{new}\big\|_2^2 \\ &\overset{\text{cf. (4.48)}}{=} (y - \hat{y})^\mathsf{T}(y - \hat{y}) \\ &\quad - (y - \hat{y})^\mathsf{T}(x_k - \hat{x}_k)\big((x_k - \hat{x}_k)^\mathsf{T}(x_k - \hat{x}_k)\big)^{-1}(x_k - \hat{x}_k)^\mathsf{T}(y - \hat{y}) \\ &= \big\|y - \hat{y}\big\|_2^2 - (x_k - \hat{x}_k)^\mathsf{T}(y - \hat{y})\Big(\big\|x_k - \hat{x}_k\big\|_2^2\Big)^{-1}(x_k - \hat{x}_k)^\mathsf{T}(y - \hat{y}). \end{aligned}$$

Hence, the reduction in the RSS $\big\|y - \hat{y}\big\|_2^2$ from entering the variable x_k is the ratio

$$\frac{\big((x_k - \hat{x}_k)^\mathsf{T}(y - \hat{y})\big)^2}{\big\|x_k - \hat{x}_k\big\|_2^2}.$$

Now, for $i = r+1, \ldots, p$, the diagonal element $\bar{c}_{ii}$ of the matrix $\bar{C}$ (4.50) is the RSS which results from regressing x_i on the variables $x_1, \ldots, x_r$. The RSS for y at this stage is $\bar{c}_{p+1,p+1}$ and we observe that its reduction from entering the variable x_i is

$$\frac{\bar{c}^2_{i,p+1}}{\bar{c}_{ii}}, \quad i = r+1, \ldots, p. \tag{4.52}$$

We choose the independent variable that gives the greatest reduction in the residual sum of squares of the dependent variable y. That is, we pivot on the i-th diagonal element of $\bar{C}$ whose index is the i which maximizes (4.52) among all the unpivoted $\bar{c}_{ii}$. This constitutes the pivoting order for the whole stepwise regression procedure.

In each step of the process, the selected variable is only added to the regression if the tolerance of the corresponding pivot (see Definition 4.11) is above a specified threshold value. If the tolerance is low, it is very unlikely that the independent variable in question contributes significantly to the prediction of y. Hence, diagonal entries that violate the tolerance limit remain unpivoted. The process ends when no pivot among the unpivoted diagonal entries satisfies the tolerance criterion. In effect, this prevents the singular (or ill-conditioned) matrix $X^\mathsf{T}X$ from being fully inverted.

Summarized, adding one independent variable at a time to the regression, the variables are selected in the order that improves the prediction most, provided that their entry does not do too much damage to the numerical precision of the results. The current regression coefficients and the corresponding residual sum of squares are readily available at each stage of the procedure.
Additional information on stepwise regression can be found in [46, p. 413–420].

Remark. Stepwise regression based on the Gauss-Jordan algorithm can also be applied to multivariate regression problems, where the data vector y is replaced by a matrix $Y \in \mathbb{R}^{n \times q}$ whose q columns form a set of dependent variables, see Jennrich [22].

Algorithm 7 is aimed for finding an estimate of the coefficient vector for which (4.45) is minimal. It employs the discussed stepwise regression procedure, i.e., it demands as input some $(p+1) \times (p+1)$ matrix C of the form (4.46). Pivoting on the diagonal entries of $X^\mathsf{T}X$ adds variables to the regression model. Hence, after running the program, the output vector $\beta \in \mathbb{R}^p$ contains the model coefficients of the variables regressed upon. If the k-th diagonal entry ($k \in \{1, \ldots, p\}$) has remained unpivoted due to low tolerance, the independent variable x_k is not in the final regression equation. The algorithm accounts for this fact by setting $\beta_k = 0$, which is realized as follows:

Let $\mathcal{R}$ denote the set of indices of the unpivoted rows and $\mathcal{S}$ the set of indices of the pivoted rows (excluding row $p+1$). At any stage of the process, $\bar{C}$ represents the transformed input matrix C. The reduction in the RSS of y that would ensue if row r were to be pivoted, i.e., if the variable x_r were to be added to the regression, is

$$t_r := \frac{\bar{c}^2_{r,p+1}}{\bar{c}_{rr}}, \quad r \in \mathcal{R}.$$

The tolerance of the pivot $\bar{c}_{rr}$ at this stage is given by

$$tol_r := \frac{\bar{c}_{rr}}{c_{rr}}, \quad r \in \mathcal{R}.$$

The matrix $X^\mathsf{T}X$ is nonnegative, i.e., its diagonal elements are nonnegative and they remain so after any number of pivoting by Definition 4.8. Hence, tol_r is also nonnegative. The output vector β is then obtained by performing the following steps:

1. Choose a small positive number T as threshold for the pivot tolerance.
2. Let $\mathcal{R} = \{1, \ldots, p\}$ and let $\mathcal{S}$ be the empty set.
3. Among all indices $r \in \mathcal{R}$, find the one, say r^*, for which t_r is maximal and $tol_r \geq T$ is satisfied. Pivot on the r^*-th diagonal and transfer r^* from $\mathcal{R}$ to $\mathcal{S}$.
4. Repeat step 3 until there is no index $r \in \mathcal{R}$ with $tol_r \geq T$ left. Finally, set

$$\beta_i = \begin{cases} \bar{c}_{i,p+1} & \text{for } i \in \mathcal{S} \\ 0 & \text{for } i \in \mathcal{R} \end{cases}.$$

DUD employs the stepwise regression procedure in order to directly compute (4.38). Suppose that we are given the assumptions made for DUD. Let us form the matrix

$$C(\theta) := \begin{bmatrix} \Delta F^\mathsf{T}\Delta F & \Delta F^\mathsf{T}\big(y - f(\theta)\big) \\ \big(y - f(\theta)\big)^\mathsf{T}\Delta F & \big(y - f(\theta)\big)^\mathsf{T}\big(y - f(\theta)\big) \end{bmatrix}. \tag{4.53}$$

Pivoting on all p diagonal entries of $\Delta F^\mathsf{T}\Delta F$ transforms the $(p+1)$-th column (excluding the last entry) of $C(\theta)$ into

$$\big(\Delta F^\mathsf{T}\Delta F\big)^{-1}\Delta F^\mathsf{T}\big(y - f(\theta)\big),$$

which, for $\theta = \theta_{p+1}$, is the linear least squares estimator $\hat{\alpha}$, the solution of (4.37). If $\Delta F^\mathsf{T}\Delta F$ is essentially singular, the procedure prevents the matrix from being fully inverted. Let $\bar{C}(\theta)$ denote the matrix which results from performing Gauss-Jordan pivots on the diagonal of $\Delta F^\mathsf{T}\Delta F$ in the order specified by stepwise regression. Then, the program computes the vector $\alpha \in \mathbb{R}^p$ whose components are given by

$$\alpha_i = \begin{cases} \bar{c}_{i,p+1} & \text{if } \Delta F^\mathsf{T}\Delta F \text{ has been pivoted on its } i\text{-th diagonal element} \\ 0 & \text{otherwise} \end{cases}.$$

For $\theta = \theta_{p+1}$, this is an approximation of $\hat{\alpha}$ in (4.38). Thus, the invocation of Algorithm 7 in step 21 of Algorithm 5 is carried out with the input matrix $C(\theta_{p+1})$.

Algorithm 7 Stepwise regression based on Gauss-Jordan pivots

Input: matrix C of the form (4.46), tolerance threshold T
Output: coefficient vector β
Require: $0 < T \ll 1$
Ensure: provided that the tolerance criterion is met, select the unpivoted diagonal element that gives the greatest reduction in the current RSS of y as pivot

$d \leftarrow$ number of rows/columns of C;
$p = d - 1$;
$\beta \leftarrow p$-dim. zero vector ▷ storage for regression coefficients
$v \leftarrow$ first p diagonal values of C; ▷ initial diagonal for pivot tolerance
$\mathcal{R} \leftarrow \{1, \ldots, p\}$
$\mathcal{S} \leftarrow$ empty set
for $j = 1$ to p **do**
    for r in $\mathcal{R}$ **do** ▷ compute the reduction(s) in the RSS of y
        $t_r = C_{rd}^2 / C_{rr}$;
    end for
    $k \leftarrow$ index that corresponds to $\max_{r \in \mathcal{R}}(t_r)$;
    remove index k from $\mathcal{R}$;
    $a = C_{kk}$;
    $tol = a / v_k$;
    if $tol < T$ **then** ▷ tolerance criterion
        continue;
    end if
    $C_{k:} = C_{k:} / a$; ▷ divide row k by C_{kk}
    $C_{kk} = C_{kk} / a$; ▷ set $C_{kk} = 1/C_{kk}$
    for $i = 1$ to d **do**
        if $i \neq k$ **then** ▷ for every row $i \neq k$
            $b = C_{ik}$;
            $C_{i:} = C_{i:} - b \cdot C_{k:}$; ▷ subtract C_{ik} times row k from row i
            $C_{ik} = -b / a$; ▷ set $C_{ik} = -C_{ik}/C_{kk}$
        end if
    end for
    add index k to $\mathcal{S}$; ▷ the k-th row has been pivoted
end for
for s in $\mathcal{S}$ **do**
    $\beta_s = C_{sd}$; ▷ β gets the coefficients of the variables in the regression
end for
return β; ▷ β is approximate OLS solution

4.3.3 Implementation details

This section is concerned with the implementation of DUD (Algorithm 5), which we programmed in MATLAB and can be found as C.5 in Appendix C. The invoked stepwise regression procedure (Algorithm 7) is written as subfunction of DUD and has been placed at the end of C.5.
We have also provided the MATLAB code for the Gauss-Jordan in-place matrix inversion (Algorithm 6), see C.6 in Appendix C.

The implementation details of DUD given by Ralston and Jennrich in [44] are vague. Some information on how the original program was developed can be found in [9]. It was initially written in FORTRAN and has been used for the analysis of insulin data. However, no explicit source code of the program is traceable and we found some inconsistencies between [9] and [44]. For example, the parameter update described in [9, p. 813] allows for the replacement of more than two members of the old parameter set. Also, the authors did not further elaborate on the applied stepwise regression procedure based on the Gauss-Jordan algorithm. So, apart from using a different programming language, it is very likely that our implementation of DUD varies widely from the original version.

Initialization

DUD demands a single user supplied starting parameter $\theta_{p+1} \in \mathbb{R}^p$. The p additional initial parameters θ_i are then generated by displacing the i-th component of θ_{p+1}, $i = 1, \ldots, p$. In [9, p. 812], this displacement is simply described as the sum of the i-th component of the starting vector and some input value. For their numerical tests, Ralston and Jennrich [44] predominantly used some small number times the corresponding entry of θ_{p+1} for that value. However, we observe that if θ_{p+1} had a component that is zero, this would lead to the duplication of the parameter. The authors did not further specify on how they handled occurring zeros in θ_{p+1}.
We generate the additional starting parameters $\theta_i \in \mathbb{R}^p$, $i = 1, \ldots, p$, as follows. Let us consider the user-supplied vectors $\theta_{p+1} \in \mathbb{R}^p$ and $h \in \mathbb{R}^p$ with $h_i \in \mathbb{R} \setminus \{0\}$ for $i = 1, \ldots, p$. Then, the θ_i are obtained from θ_{p+1} by computing their components as

$$(\theta_i)_j = (\theta_{p+1})_j + \delta_{ij} h_i, \quad j = 1, \ldots, p, \tag{4.54}$$

where δ_{ij} denotes the Kronecker delta. That is, the θ_i only differ in their i-th entry from θ_{p+1} and the h_i determine whether the generated set of initial vectors is linearly independent or not. Furthermore, the whole parameter set is actively involved in the minimization process performed by DUD. So, choosing proper input vectors θ_{p+1} and h is crucial for the performance of the algorithm.
Inspired by [44], we found that the following choice for h works exceptionally well for the problems we looked at with different starting vectors. We compute h by setting

$$h_i = \begin{cases} (\theta_{p+1})_i \cdot 0.1 & \text{if } (\theta_{p+1})_i \neq 0 \\ 0.01 & \text{if } (\theta_{p+1})_i = 0 \end{cases}, \quad i = 1, \ldots, p. \tag{4.55}$$

This rule seems to be reasonably robust and we decided to implement it in our MATLAB program C.5. The purely heuristic value 0.01 accounts for the displacement of possible zero components of θ_{p+1}. We observed that significantly smaller values tend to destroy the numerical precision of the matrix $\Delta F^\mathsf{T} \Delta F$. The factor 0.1 in (4.55) is satisfactory for the starting values of most problems. However, we note that the user can almost always find slightly better values for the h_i at a particular problem. This is why it is important to also allow for arbitrary inputs of h in C.5, cf. Algorithm 5.
MATLAB offers an efficient way for the implementation of the displacement (4.54). The for-loop in Algorithm 5 (steps 8–10) can be replaced by creating a logical mask with which we can directly operate on the diagonal of the matrix Θ, see line 182 of C.5.

As an example for the generation of the initial set of parameters in DUD we look at the case $p = 3$. By employing the equations (4.54) and (4.55) we find that

$$\theta_{p+1} := \begin{pmatrix} 0 \\ 10 \\ 1 \end{pmatrix} \implies \theta_1 = \begin{pmatrix} 0.01 \\ 10 \\ 1 \end{pmatrix}, \theta_2 = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} \text{ and } \theta_3 = \begin{pmatrix} 0 \\ 10 \\ 0.1 \end{pmatrix}.$$

DUD requires $p + 1$ function calls for starting. In C.5, the evaluations of the input function f and the computations of the corresponding residual sum of squares are exactly performed as in the steps 11–14 of Algorithm 5, i.e., within a for-loop. The MATLAB routine `sort` [30, p. 11208–11215] is invoked to sort the vector Q of residual sum of squares in descending order and vectorization is used to rearrange the parameters and their function evaluations accordingly. Then, θ_{p+1} is assigned the last column of the matrix Θ, the newly arranged parameter set.

Main program

The MATLAB routine `bsxfun` [30, p. 906–910] allows for elementwise subtraction. That is, the computations of both the matrices $\Delta\Theta$ and ΔF in program C.5 are performed without employing for-loops as in the steps 18–20 and 22–24 of Algorithm 5. Then, the residual $r = y - f(\theta_{p+1})$ is computed and the $(p + 1) \times (p + 1)$ matrix

$$C = \begin{bmatrix} \Delta F^{\mathsf{T}} \\ r^{\mathsf{T}} \end{bmatrix} \begin{bmatrix} \Delta F & r \end{bmatrix} = \begin{bmatrix} \Delta F^{\mathsf{T}} \Delta F & \Delta F^{\mathsf{T}} r \\ r^{\mathsf{T}} \Delta F & r^{\mathsf{T}} r \end{bmatrix} \tag{4.56}$$

(cf. matrix (4.53)) is formed by matrix multiplication, which requires $\mathcal{O}(np^2)$ arithmetic operations. The stepwise regression procedure, which is implemented as subfunction in C.5, is invoked in order to compute α, an approximation of $\hat{\alpha}$ (4.38). Performing Gauss-Jordan pivots on the submatrix $\Delta F^{\mathsf{T}} \Delta F$ of C dominates the computational cost of the stepwise regression procedure, i.e., it needs $\mathcal{O}(2p^3)$ operations.

Remark. For nonsingular ΔF, the normal equations (4.37) could be efficiently solved in MATLAB with the use of the backslash operator \ (see the remark on page 36). However, we chose not to do so because of the following reason. We would need to ensure beforehand that the matrix $\Delta F^{\mathsf{T}} \Delta F$ is well-conditioned. Clearly, matrix inversion has to be avoided in the calculation of the condition number. That is, we would compute the 2-norm condition number of $\Delta F^{\mathsf{T}} \Delta F$, as it is simply the ratio of the largest singular value of the matrix to the smallest singular value, see relationship (A.1.3) in Appendix A. This can be realized by invoking the MATLAB routine `cond` [30, p. 1651–1654], which employs a singular value decomposition for the computation. But just performing the SVD on $\Delta F^{\mathsf{T}} \Delta F$ would already result in higher computational cost than a straightforward application of the stepwise regression procedure on C. Furthermore, we did not observe any difference in the accuracy of the computed solution α when we compared the method for normal equations with the stepwise regression procedure for matrices $\Delta F^{\mathsf{T}} \Delta F$ with condition number

$$\kappa_2\big(\Delta F^{\mathsf{T}} \Delta F\big) < \frac{1}{\sqrt{eps}}.$$

Diagonal elements of $\Delta F^\mathsf{T}\Delta F$ with low tolerance are not used as pivots. Jennrich [22] proposed a value of 10^{-5} for the tolerance threshold T on an 8 decimal place computer. Using a double precision floating-point format, we found that $T = \sqrt{eps}$ seems to be the optimal choice in the sense that smaller threshold values would result in rounding errors that harm the precision of the entire matrix. In program C.5, the two inner for-loops of Algorithm 7 for computing the current reduction in the residual sum of squares and performing the Gauss-Jordan pivots are replaced by vectorization.
After $\theta_{new} = \theta_{p+1} + \Delta\Theta\alpha$, $f(\theta_{new})$ and $Q(\theta_{new})$ have been computed, DUD checks whether the user has entered an integer $m > 0$ and $Q(\theta_{new}) > Q(\theta_{p+1})$ holds. If so, the program attempts to find a point that decreases $Q(\theta)$ by performing the line search (steps 29–34 of Algorithm 5), which is implemented as subfunction in C.5. That is, maximal m line searches with θ_{new} are carried out, each of which requiring a function call. The input m also determines the members of the sequence (4.40). We found that replacing $\frac{1}{2}$ by $\frac{1}{3}$ in the definition of the d_i in (4.40), $i = 1, \dots m$, improves the performance of the step shortening procedure for certain problems. However, overall the program tends to need less line searches for finding an estimate that reduces the RSS when the sequence (4.40) is left unchanged. If no such estimate is found, DUD simply keeps the parameter computed in the last line search.
Let us with θ^*_{new} denote the newly determined parameter in some iteration of DUD. It is possible that the numerical value of $f(\theta^*_{new})$ is larger than $1.797 \cdot 10^{308}$, which cannot be represented in 64 bits on the computer. If the algorithm proceeds with this parameter, it automatically fails since the resulting matrix of function differences ΔF is not representable. In the case that at least one line search was performed, it might be possible to choose a valid estimate instead. This is why we implemented a safeguard (the lines 452–469 in program *C.5*) that aims to prevent DUD from keeping such unacceptable parameters. When it activates, out of the newly found estimates the one with the lowest RSS is kept. If this parameter is also unacceptable, program *C.5* simply keeps the estimate from the linear approximation.
DUD updates the set of parameter vectors at the start of the next iteration as described on page 67. Program C.5 uses the MATLAB routine `find` [30, p. 3522–3530] in order to determine the first subscript i for which $|\alpha_i| > 10^{-5}$. We did not notice an increase in algorithmic performance when using smaller values than 10^{-5}. In fact, quite the opposite occurred, as DUD performed worse when we replaced 10^{-5} by 10^{-8}. The `circshift` routine [30, p. 1334–1337] allows for the rearrangement of the parameters in the updated set according to their age (from oldest to newest), i.e., allows for the implementation of the steps 51–52 and 56–58 of Algorithm 5.
Our implementation of DUD requires $\mathcal{O}(np^2 + 2p^3)$ arithmetic operations and, when the step shortening procedure is not used, one function evaluation per iteration.
Simply reaching the maximum number of iterations (or function evaluations) is not enough as stopping criterion for the algorithm. If the minimum has already been found, DUD tends to over-optimize. Hence, we highly recommend ending the process when a sufficiently good minimization is achieved. Thus, our MATLAB program C.5 terminates when the objective function is minimized or some user-defined maximum number of function evaluations is reached. In addition, as suggested by Ralston and Jennrich in [44], we did account for cases where the relative change in the RSS is small. That is, program C.5 also stops when it holds that for 5 successive iterations

$$\frac{|Q(\theta_{new}) - Q(\theta_{p+1})|}{Q(\theta_{p+1})} \leq 10^{-5}.$$

5 Benchmarking

In this section, we compare the performances of GN, LM, FDGN, FDLM and DUD by using the Moré-Garbow-Hillstrom collection of test functions, see [33]. We wrote MATLAB programs for all 35 test functions from [33] and appended their code in C.7. For each such MATLAB function, we used the recommended standard starting point. Also, we selected 9 randomly generated points from boxes of different size surrounding the respective initial point. Hence, in total 10 starting points were used per function. That is, we tested every algorithm on 350 optimization problems. Since we used DUD both with and without its optional line search, in reality 6 procedures were compared and thus 2100 different algorithmic runs were performed.
The benchmarking is based on the number of equivalent function evaluations and the average wall-clock times of the successful minimizations. In-depth information on how to benchmark optimization algorithms can be found in [4].

5.1 Test Set

The Moré-Garbow-Hillstrom collection [33] is the standard test set for low-dimensional problems in the local optimization community. All of its 35 functions are represented as nonlinear least squares problems of the form (3.3) and include standard starting points that are not too close to the known global minima. Some of these problems also have local solutions. In fact, Kuntsevich [25] pointed out that there appear to be more local minima than originally stated in [33]. Thus, we have added the data collected by Kuntsevich to our test set, which is shown in Table 1.
Using this relatively large number of functions for benchmarking required a lot of coding (see Appendix C.7); however, it makes the whole experiment more meaningful. The test set contains easy to hard to solve problems of various types, which render specific parameter tuning irrelevant and reveal the real strengths and weaknesses of the compared procedures. This means that the algorithms were not only measured for their efficiency, but also for their robustness. The result reliability was further increased by repeating tests on the same functions with a variety of starting points.

5.1.1 Starting points

The choice of the initial guesses is crucial for the success of the optimization methods. It can spell the difference between slow and rapid convergence to the solution. Hillstrom described the need to test optimization software at nonstandard starting points in [21]. Therefore, we created additional points by introducing random perturbations of different magnitude to the given starting points, see page 87.
For a fair and unbiased evaluation of the experiment, each procedure was provided the same ten initial points per optimization problem. The approximate generated starting points used for each test function can be looked up in Table 5 of Appendix B.

Table 1: Used test functions and their known global/local minima.

No.	Function Name	n	m	Global Min.	Local Min.
1	Rosenbrock	2	2	0.00000e+00	-
2	Freudenstein and Roth	2	2	0.00000e+00	4.89843e+01
3	Powell badly scaled	2	2	0.00000e+00	-
4	Brown badly scaled	2	3	0.00000e+00	-
5	Beale	2	3	0.00000e+00	-
6	Jennrich and Sampson	2	10	1.24362e+02	2.59580e+02
7	Helical Valley	3	3	0.00000e+00	-
8	Bard	3	15	8.21487e−03	1.74286e+01
9	Gaussian	3	15	1.12793e−08	-
10	Meyer	3	16	8.79458e+01	-
11	Gulf research and development	3	10	0.00000e+00	3.80000e−02
12	Box three-dimensional	3	10	0.00000e+00	-
13	Powell singular	4	4	0.00000e+00	-
14	Wood	4	6	0.00000e+00	-
15	Kowalik and Osborne	4	11	3.07506e−04	1.02734e−03 1.79454e−03
16	Brown and Dennis	4	20	8.58222e+04	-
17	Osborne 1	5	33	5.46489e−05	-
18	Biggs EXP6	6	13	0.00000e+00	5.65565e−03 3.06367e−01
19	Osborne 2	11	65	4.01377e−02	1.78981e+00 2.63057e+01
20	Watson	9	31	1.39976e−06	-
21	Extended Rosenbrock	10	10	0.00000e+00	-
22	Extended Powell singular	12	12	0.00000e+00	-
23	Penalty I	4	5	2.24997e−05	-
24	Penalty II	4	8	9.37629e−06	-
25	Variably dimensioned	10	12	0.00000e+00	-
26	Trigonometric	10	10	0.00000e+00	2.79506e−05
27	Brown almost-linear	10	10	0.00000e+00	1.00000e+00
28	Discrete boundary value	10	10	0.00000e+00	-
29	Discrete Integral equation	10	10	0.00000e+00	-
30	Broyden tridiagonal	10	10	0.00000e+00	1.36026e+00 1.02865e+00 1.05123e+00 7.12606e−01 3.97373e−01
31	Broyden banded	10	10	0.00000e+00	3.05728e+00 2.68022e+00
32	Linear - full rank	10	20	1.00000e+01	-
33	Linear - rank 1	10	20	4.63415e+00	-
34	Linear - rank 1 with zero columns and rows	10	20	6.13514e+00	-
35	Chebyquad	9	9	0.00000e+00	-

Remark (New starting points). If $x_0 \in \mathbb{R}^n$ denotes the standard starting point of some function from the test set, then another initial point $\hat{x}_0$ was created by setting

$$\hat{x}_0 = x_0 + \alpha p,$$

where $\alpha \in \mathbb{R}_+$ and $p \in \mathbb{R}^n \setminus \{0\}$ is a vector of uniformly distributed random numbers in the interval $(-1, 1)$. That is, the scalar α controls the magnitude of the perturbation. We used the MATLAB routine `rand` [30, p. 10017–10023] to generate 3 such vectors p for each problem dimension n from our test set, see Table 1. Then, 3 different scaling factors α were applied per vector, which ultimately resulted in 9 new starting points for each test function. Typically, $\alpha \in \{1, 10, 100\}$ in order to grasp the ability of the algorithms to reach close, medium and far ahead solutions. However, the nature of some problems does only allow for minor deviations from the standard starting point, so smaller scaling (e.g., $\alpha \in \{0.01, 0.1, 1\}$) was also used, cf. Table 5 in Appendix B.

5.2 Testing Framework

We evaluated all the competing algorithms on the same 350 test problems using the same performance measures, **parameter tuning and stopping conditions**. According to [40, p. 8–9], good algorithms should possess the following properties:

- **Robustness**. They should perform well on a wide variety of problems in their class, for all reasonable values of starting points.
- **Efficiency**. They should not require excessive computer time or storage.
- **Accuracy**. They should be able to identify a solution with precision, without being overly sensitive to errors in the data or to arithmetic rounding errors that occur when the algorithm is implemented on a computer.

Typically, the researcher has to face a trade-off between robustness and efficiency when choosing an optimization method. In the following, we describe how the performance of our procedures was measured with regard to the above attributes.

5.2.1 Robustness

The Moré-Garbow-Hillstrom collection was specifically produced for testing the reliability and robustness of unconstrained optimization software, see [33, p. 1–2]. In Section 5.1.1, we described how we added suitable starting points for its broad range of test functions. Considering each pair consisting of function and initial point as separate optimization problem in algorithmic runs has helped us to track and extract the data. Since we benchmark deterministic methods, each run was only performed once (except for determining the average running times). Hence, the reliability can be easily grasped by counting the number of test problems that were successfully solved by the respective algorithm. That is, we can report reliability as *success rate*.

5.2.2 Efficiency

The efficiency of an optimization method basically refers to the computational effort required to obtain a feasible solution. It can be gauged by several performance measures, with the **number of function evaluations** and the algorithmic **running time** being the most popular ones. A less common measure, which we did not include in our benchmark, is memory usage.

Number of function evaluations. It is not always clear how the total number of function evaluations is obtained in numerical experiments. In our benchmark, every single function evaluation executed by an algorithm was counted, i.e., we also included the calls needed for starting and for the final evaluation at the minimum. This is important for us, as DUD already needs several function calls to generate its initial parameter set. Furthermore, in order to guarantee a fair comparison between gradient and derivative-free methods, we have to account for the user-supplied Jacobian matrices used by GN and LM. This can be achieved by combining the number of function calls and Jacobian matrix computations into one statistic, the *number of equivalent function evaluations*. We followed the exposition of Hillstrom in [21, p. 309], where equivalent function evaluations are determined as follows:

(i) The n-variable objective or composite function evaluation is assigned a weight of 1 per call.

(ii) The gradient computation is assigned a weight of n per call.

(iii) The vector-valued function computation required by least squares solvers is assigned weights of $1/n$ or 1 depending upon whether one component or the total vector is evaluated per call.

(iv) The Jacobian matrix computations are assigned weights of n per call.

In fact, only (iii) and (iv) applied to our algorithmic runs. We recall that the Jacobian of some vector-valued function is the matrix of all its first-order partial derivatives, see (3.4) in Section 3. A typical numerical differentiation procedure would evaluate this vector-valued function n times for the calculation of the partial derivatives. Backward-mode automatic differentiation would only need $\mathcal{O}(1)$ evaluations for the Jacobian computation, but often cannot be applied for problems where derivative-free methods are used, see page 11. Therefore, it is justified that we utilized (iv) for the user-supplied Jacobians.
The advantage of this performance measure is that it indicates the effort on problems for which function and derivative evaluation is expensive. However, it is unreasonable when function calls are cheap or when they do not dominate the internal workings of the respective procedure. This is why we also measured the computation times.

Running time. We determined the wall-clock times of our algorithmic runs. Wall-clock time contains CPU time but is tied a specific hardware and software configuration, with ours being listed at the beginning of Section 5. In addition, it is

heavily influenced by background computer operations, resulting in variable time measurements. Consequently, we shut down all unnecessary processes and programs when performing the experiments.
In order to obtain accurate running time estimates, each optimization problem from our test set (see Section 5.1) was executed 1000 times by the respective procedure. The elapsed times were then used to calculate the *average wall-clock time* of each successful algorithmic run, which is reported in seconds.
So, the average wall-clock time of a run indicates how long it actually takes for an algorithm to solve a problem. The downside of this performance measure is that we cannot compensate for user-supplied Jacobians. Such computationally cheap derivative information is usually not available in practice. That is, the comparison between our gradient and derivative-free methods in terms of running time should be treated with caution. All computations were done in MATLAB R2016a on an AMD Ryzen 5 PRO 3500U CPU with 16 GB of RAM and a 64-bit version of Windows 10.

5.2.3 Accuracy

We already know the minima of the given test functions, see Table 1. Hence, we employed the *fixed-target method* for measuring the quality of our algorithmic outputs. In the fixed-target method, the required time (here function calls or wall-clock time) to find a solution at an accuracy target is evaluated [4, p. 10]. The algorithms may not be able to solve certain test problems, i.e., their termination criterion cannot be solely based on accuracy. Hence, we used additional stopping conditions, given in the upcoming Section 5.2.4, for our runs. We decided to target an accuracy of 10^{-5}. In derivative-free optimization, this level of accuracy is common and considered reasonably high. However, it is mild compared to classical convergence tests based on the gradient [34, p. 15]. In our context, a classification as *medium accuracy* is appropriate, cf. [39, p. 7].

5.2.4 Parameter tuning and stopping conditions

In principle, all of our methods were benchmarked with their default configuration. That is, the specifics can be looked up in Section 4 and in Appendix C. However, since we employed the fixed-target approach, we deactivated all occurring stopping conditions and used (5.1), (5.2) and (5.3) instead. Thus, all test problems were handled by the same set of parameters and termination criteria.

Let $x^* \in \mathbb{R}^n$ be a known minimizer of some NLLS objective function $f(x)$, see (3.3). We decided to terminate the algorithms whenever a point $\bar{x} \in \mathbb{R}^n$ with

$$\left|f(\bar{x}) - f(x^*)\right| < 10^{-5}, \quad \text{for } f(x^*) < eps, \tag{5.1}$$

or

$$\frac{\left|f(\bar{x}) - f(x^*)\right|}{f(x^*)} < 10^{-5}, \quad \text{for } f(x^*) \geq eps, \tag{5.2}$$

where *eps* is the machine epsilon, is found. That is, $\bar{x}$ is an approximation to the solution within the accuracy target and thus acceptable.

Remark (Local solutions). It is important to note that the solution x^* does not have to be global. Some of our test functions have one or more local minima, see Table 1.

The stopping conditions (5.3) have been added as safeguard to end the execution of algorithmic runs that do not convergence within a maximal computational budget. If convergence had not yet occurred, our runs were continued as long as

$$\begin{aligned} nef &< 1000 \\ t &< 0.5\ s, \end{aligned} \tag{5.3}$$

where *nef* denotes the acquired number of equivalent function evaluations and t the elapsed wall-clock time in seconds.
If an algorithm reached the desired accuracy without violating (5.3), then the respective run was counted as *success* and the *nef* and t values were reported. If an algorithm terminated before reaching the desired accuracy, it was considered unsuccessful in solving the respective test problem.

Remark (Upper bounds). Limiting the computational budget was absolutely mandatory, since, including the repetitions, we performed a total of 2.1 million test runs. An upper bound of 1000 function calls is a reasonable value for benchmarking and was also used by the authors of our test set, see [33, p. 31]. The tolerance of 0.5 seconds is generous enough to capture slower runs (with a high *nef* value), which has ultimately led to a better understanding of the robustness of our algorithms. Furthermore, runs with very slow convergence were stopped and thus considered unsuccessful. This is wanted, since they would have greatly affected the average results, leading to wrong conclusions about algorithmic efficiency.

5.3 Numerical Results

We compare the performances of GN, LM, FDGN, FDLM, DUD m=0 and DUD m=5. The results of our numerical experiments are presented in three categories, namely **tables**, **convergence plots** and **performance plots**. Details on the underlying test set and testing framework were given in the previous Sections 5.1 and 5.2.

Remark (DUD setting). The suffixes "m=0" and "m=5" simply indicate the input values for $m \in \mathbb{N}_0$ of Algorithm 5 from Section 4.3. That is, we tested DUD with both disabled and enabled line search option. We have found that values of m greater than 5 typically result in significantly higher numerical cost with little or no improvement in solution quality. The internal initial parameter set was always computed by setting h as in (4.55), see Section 4.3.3.

5.3.1 Tables

Each table presented contains summarized information obtained from the algorithmic runs on the respective 10 optimization problems per test function. More precisely, for each pair consisting of algorithm and test function, Table 2 shows the numbers of successes, Table 3 the average numbers of equivalent function evaluations and Table 4 the average solving times in seconds.

Table 2: Numbers of successful runs.

No.	Function Name	GN	LM	FDGN	FDLM	DUD m = 0	DUD m = 5
1	Rosenbrock	10	9	10	9	10	10
2	Freudenstein and Roth	10	8 (6)†	10	8 (6)	4	3
3	Powell badly scaled	9	10	9	10	8	8
4	Brown badly scaled	10	0	10	0	10	1
5	Beale	9	5	9	4	3	5
6	Jennrich and Sampson	0	3 (3)	0	2 (2)	0	0
7	Helical Valley	6	10	10	10	8	10
8	Bard	4	4	4	4	9 (5)	9 (5)
9	Gaussian	6	6	6	6	6	6
10	Meyer	1	0	0	0	0	2
11	Gulf research and development	0	9	0	9	0	1
12	Box three-dimensional	6	8	6	8	6	6
13	Powell singular	10	8	10	8	10	10
14	Wood	10	9	10	9	10	10
15	Kowalik and Osborne	2 (1)	2	1	3	0	3
16	Brown and Dennis	0	10	0	10	0	0
17	Osborne 1	4	6	4	6	4	9
18	Biggs EXP6	0	9 (1)	0	9	0	0
19	Osborne 2	0	0	0	0	5	8
20	Watson	10	7	10	7	10	10
21	Extended Rosenbrock	10	7	10	7	10	8
22	Extended Powell singular	10	7	10	7	8	7
23	Penalty I	0	0	1	1	10	0
24	Penalty II	0	0	0	10	0	0
25	Variably dimensioned	10	7	10	7	10	9
26	Trigonometric	2	5	2	4	0	0
27	Brown almost-linear	2 (2)	10	2 (2)	10	5 (2)	3 (2)
28	Discrete boundary value	10	10	10	10	9	10
29	Discrete Integral equation	10	7	10	7	7	10
30	Broyden tridiagonal	10	10	10	10	10	10
31	Broyden banded	10	10	10	10	3	3
32	Linear - full rank	10	7	10	7	10	10
33	Linear - rank 1	0	10	10	10	10	10
34	Linear - rank 1 with zero columns and rows	0	10	0	10	10	10
35	Chebyquad	0	7	0	8	0	0
Successes (out of 350 runs)		191 (3)	230 (10)	204 (2)	240 (8)	205 (7)	201 (7)
Success Rate (%)		54.6	65.7	58.3	68.6	58.6	57.4

† Numbers in brackets highlight how often convergence to a local minimum occurred

Table 3: Average numbers of equivalent function evaluations of the successes.

No.	Function Name	GN	LM	FDGN	FDLM	DUD m = 0	DUD m = 5
1	Rosenbrock	7	93	7	93	6	36
2	Freudenstein and Roth	44	65	61	65	20	20
3	Powell badly scaled	87	76	95	76	39	70
4	Brown badly scaled	21	F[†]	21	F	113	742
5	Beale	242	36	105	22	44	284
6	Jennrich and Sampson	F	198	F	174	F	F
7	Helical Valley	21	156	38	156	114	107
8	Bard	15	15	15	15	12	26
9	Gaussian	15	15	15	15	13	19
10	Meyer	61	F	F	F	F	633
11	Gulf research and development	F	58	F	58	F	34
12	Box three-dimensional	11	190	11	190	7	7
13	Powell singular	45	105	45	105	22	37
14	Wood	77	274	77	274	22	90
15	Kowalik and Osborne	234	76	66	151	F	144
16	Brown and Dennis	F	96	F	105	F	F
17	Osborne 1	28	38	28	38	29	141
18	Biggs EXP6	F	55	F	65	F	F
19	Osborne 2	F	F	F	F	181	234
20	Watson	70	364	208	357	65	180
21	Extended Rosenbrock	23	135	23	135	65	168
22	Extended Powell singular	126	166	127	166	79	206
23	Penalty I	F	F	26	26	163	F
24	Penalty II	F	F	F	111	F	F
25	Variably dimensioned	92	78	93	80	110	101
26	Trigonometric	260	353	414	540	F	F
27	Brown almost-linear	23	76	23	76	358	44
28	Discrete boundary value	51	346	51	346	27	118
29	Discrete Integral equation	52	65	52	65	17	79
30	Broyden tridiagonal	37	37	37	37	17	21
31	Broyden banded	251	253	190	131	118	103
32	Linear - full rank	12	43	12	45	12	12
33	Linear - rank 1	F	30	12	30	12	12
34	Linear - rank 1 with zero columns and rows	F	27	F	27	12	12
35	Chebyquad	F	520	F	542	F	F
Average (over all successes)		71	134	67	133	59	96

[†] F means that the algorithm failed to converge for all 10 starting points

Table 4: Average wall-clock times [s] of the successes.

No.	Function Name	GN	LM	FDGN	FDLM	DUD m = 0	DUD m = 5
1	Rosenbrock	1.189e−04	7.128e−04	9.597e−05	8.881e−04	3.352e−04	1.089e−03
2	Freudenstein and Roth	4.762e−04	7.271e−04	7.557e−04	8.543e−04	1.649e−03	1.199e−03
3	Powell badly scaled	1.977e−02	1.576e−02	2.686e−02	1.728e−02	3.572e−03	3.337e−03
4	Brown badly scaled	2.147e−04	F[†]	2.233e−04	F	8.226e−03	2.435e−02
5	Beale	6.062e−03	3.934e−04	1.236e−03	3.313e−04	3.857e−03	7.296e−03
6	Jennrich and Sampson	F	5.016e−02	F	3.705e−02	F	F
7	Helical Valley	1.801e−04	8.934e−04	3.560e−04	1.404e−03	1.060e−02	4.346e−03
8	Bard	2.293e−04	1.999e−04	2.293e−04	2.351e−04	1.147e−03	1.287e−03
9	Gaussian	1.347e−04	1.188e−04	1.615e−04	1.660e−04	1.126e−03	1.169e−03
10	Meyer	4.276e−04	F	F	F	F	2.475e−02
11	Gulf research and development	F	9.413e−04	F	1.531e−03	F	1.321e−03
12	Box three-dimensional	1.399e−04	1.363e−03	1.654e−04	2.722e−03	4.749e−04	4.152e−04
13	Powell singular	2.356e−04	4.965e−04	3.511e−04	8.140e−04	2.187e−03	2.254e−03
14	Wood	4.801e−04	1.380e−03	6.259e−04	2.181e−03	2.476e−03	4.324e−03
15	Kowalik and Osborne	5.175e−02	9.104e−04	8.441e−04	2.161e−03	F	8.290e−03
16	Brown and Dennis	F	1.332e−03	F	1.611e−03	F	F
17	Osborne 1	2.751e−04	3.858e−04	4.136e−04	6.846e−04	4.174e−03	7.518e−03
18	Biggs EXP6	F	6.667e−04	F	1.630e−03	F	F
19	Osborne 2	F	F	F	F	5.752e−02	2.268e−02
20	Watson	1.901e−03	8.661e−03	1.831e−02	3.150e−02	1.945e−02	2.785e−02
21	Extended Rosenbrock	1.391e−04	7.493e−04	2.687e−04	1.418e−03	1.422e−02	1.690e−02
22	Extended Powell singular	3.763e−04	4.423e−04	9.055e−04	1.235e−03	1.615e−02	1.409e−02
23	Penalty I	F	F	3.019e−04	2.989e−04	1.896e−02	F
24	Penalty II	F	F	F	2.321e−03	F	F
25	Variably dimensioned	3.435e−04	2.740e−04	8.140e−04	7.236e−04	1.726e−02	1.227e−02
26	Trigonometric	1.242e−03	2.521e−03	4.942e−03	6.598e−03	F	F
27	Brown almost-linear	1.882e−04	4.120e−04	2.513e−04	9.036e−04	5.450e−02	3.913e−03
28	Discrete boundary value	4.806e−04	2.494e−03	7.279e−04	4.813e−03	4.033e−03	8.177e−03
29	Discrete Integral equation	4.525e−04	5.272e−04	2.544e−03	3.245e−03	2.232e−03	8.699e−03
30	Broyden tridiagonal	4.319e−04	3.955e−04	5.300e−04	5.351e−04	1.755e−03	1.974e−03
31	Broyden banded	2.131e−03	2.204e−03	1.034e−02	7.158e−03	2.851e−02	1.243e−02
32	Linear - full rank	3.550e−04	2.334e−04	1.665e−04	5.591e−04	4.369e−04	3.856e−04
33	Linear - rank 1	F	2.125e−04	8.613e−04	4.250e−04	3.845e−04	3.048e−04
34	Linear - rank 1 with zero columns and rows	F	4.433e−04	F	6.807e−04	5.518e−04	4.591e−04
35	Chebyquad	F	2.703e−03	F	6.712e−03	F	F
Average (over all successes)		2.227e−03	2.440e−03	3.175e−03	3.757e−03	9.059e−03	7.108e−03

[†] F means that the algorithm failed to converge for all 10 starting points

Table 2 gives us a good impression of the robustness of our algorithms. First of all, we can clearly see the difference in difficulty of the test functions. For example, all the procedures were able to solve the 10 problems associated with Broyden's tridiagonal function (30), but only FDLM could handle the penalty function II (24). Furthermore, every tested method severely struggled in minimizing both Jennrich's and Sampson's (6) and Meyer's (10) function. Interestingly, only DUD (and not LM or FDLM) has found solutions to problems associated with Osborne's 2 function (19). We can clearly see that FDLM and LM are the most reliable methods in our benchmark. That outcome was to be expected, as they are aimed to be reliable by design, see the theory from Section 4.2. Actually, FDLM has solved 10 more test problems than LM which shows how well the derivative-free analogue behaves. However, the difference in success rate of just under 3% is marginal and is expected to be even smaller when benchmarking at a higher accuracy level such as 10^{-8}. The success rate of FDGN is also higher than that of its gradient counterpart GN and is similar to that of DUD m=0 and DUD m=5.
To our surprise, the enabled line search does not improve the reliability of DUD. This is in contrast to the findings of Ralston and Jennrich in [44], where the results on Box's two and three-dimensional functions suggest that DUD is more successful with $m = 5$, i.e., when its step shortening procedure is on.
The bracketed numbers in Table 2 indicate that none of our solvers exhibits a noticeable tendency towards convergence to local minima.

Table 3 must be read with caution, as the individual entries depend heavily on the values in Table 2. For example, the 742 function calls used by DUD m=5 for minimizing Brown's badly scaled function correspond to a single problem, whereas the significantly smaller values for GN, FDGN and DUD m=0 associated with this function are averages over 10 test runs. That is, each algorithm requires a significant number of successful runs to enable a somewhat meaningful comparison in terms of efficiency. Hence, the averages over all successes provide the best comparable figures for us. However, it should be clear that these values are still suboptimal for benchmarking. For instance, suppose that the test problems associated with Chebyquad's function (35) were not considered. Then both LM and FDLM would have significantly better numbers on average function calls while still having a higher success rate than the other methods. Then again, removing the results for Brown's and Dennis' (16) or Biggs' EXP6 (18) function from consideration would lead to a worse perception of LM and FDLM. This suggests that all data must be evaluated without artificial changes.
The average numbers of equivalent function evaluations over all successes show that GN and FDGN only need a little more than half of the function calls of LM and FDLM, respectively. Here, the gradient methods are on par with their derivative-free analogues, which is not surprising, since the user-supplied Jacobians have been weighted the same number of function evaluations as the finite difference approximations. DUD m=0 is indeed the best method when it comes to make efficient use of function evaluations. However, DUD m=5 uses additional function calls for the line search in each iteration and is therefore considered to be less efficient than GN and FDGN.

Table 4 is also based on the number of successful runs, see Table 2. Analogous to the previous description of Table 3, the average wall-clock times of all successes provide the best comparable figures for our benchmark.
We observe that the higher reliability of LM and FDLM compared to GN and FDGN, respectively, comes at the expense of slower solving times. The two gradient methods, GN and LM, clearly outperform all the derivative-free competitors in terms of speed. This is due to the fact that they had access to the analytic Jacobians of the test functions, i.e., that outcome was to be expected and is not very representative for practical applications. Although DUD m=0 uses function evaluations efficiently, FDGN performed almost three times as fast as DUD m=0. In fact, this difference in average solving times is so obvious because of the cheap evaluation of our test functions and the rather involved iterations of DUD. We can see that, on average, DUD m=5 was also faster than DUD m=0, even though more function calls were required in most cases, see Table 3. This is because the employed line search usually finds points closer to the solution, which can ultimately result in fewer iterations needed by the procedure.

All of our tables display an abundance of numerical results in a very condensed format. Graphics enable us to incorporate additional information and techniques for an even better understanding of the obtained data.

5.3.2 Convergence plots

The convergence plot is a particularly useful specialized graphic representation of the data for optimization benchmarking. We use it to visualize the performance of the different algorithms by plotting the best objective function values found against either the number of equivalent function evaluations or the wall-clock time in seconds.
A typical convergence plot only represents the results for a single problem, see [4, p. 14]. Since we want to evaluate the overall performance of the procedures, we aggregated the data from all of our runs and created two *average convergence plots*. That is, Figures 1 and 2 are both based on the average performances of each algorithm on the 350 test problems. For plotting, we extracted the currently best objective function value and the corresponding performance measures from every iteration of the successful algorithmic runs. Because of different-sized test problems, these data had to be inter- and extrapolated (with the MATLAB routine `interp1` [30, p. 6186–6200]) in order to calculate averages. This is the reason why the final average values illustrated in Figure 1 and Figure 2 can slightly deviate from the exact results in Table 3 and Table 4, respectively.
Figures 1 and 2 show that all algorithms exhibit quadratic convergence only in the end phase of their solution process. This is in accordance with Theorem 4.6 for FDGN and FDLM, and Corollary 4.7 for the GN and LM methods, see Section 4.2.2. The (arbitrarily slow) linear convergence of these procedures before reaching the final stage agrees with Theorem 4.4 and Corollary 4.5, respectively. Apart from solving speed, the two gradient methods and their derivative-free counterparts exhibit an almost identical convergence behavior. DUD needs several function calls for starting, i.e., its first objective function evaluation is somewhat delayed compared to the other methods. DUD m=5 exhibits (slow) linear convergence and DUD m=0 even shows phases of sublinear convergence rates before the quadratic convergence takes effect.

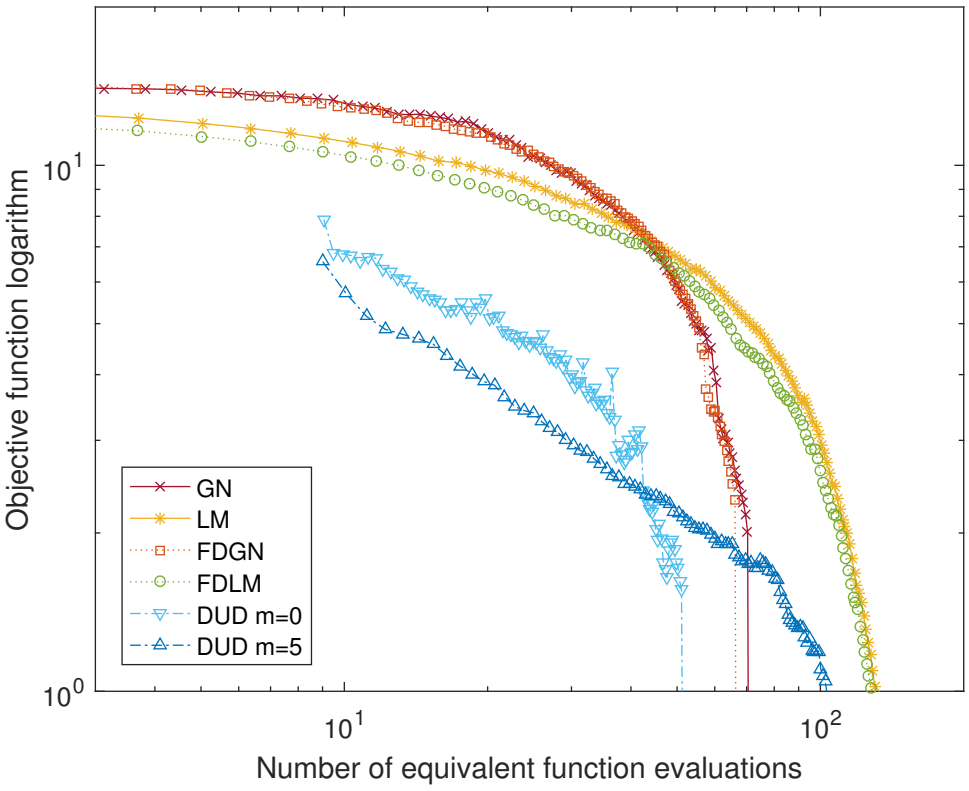


Figure 1: Convergence plot 1 (equivalent function evaluations).

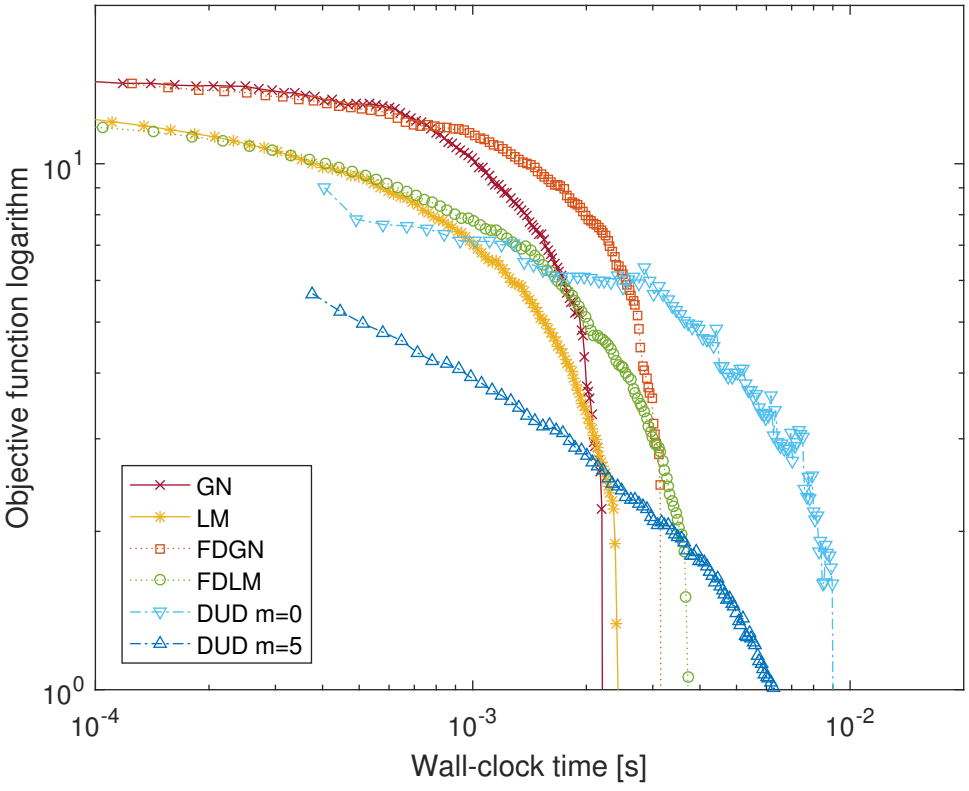


Figure 2: Convergence plot 2 (average wall-clock times).

5.3.3 Performance profiles

Performance profiles were developed by Dolan and Moré [10] and have the primary advantage that they include both convergence speed and success rate in a graphically compact form. Here, the ratio of the running time (number of function evaluations) of the respective algorithm versus the best time (number of function evaluations) of all the procedures in the benchmark is used as performance metric.
Let us denote by $\mathcal{P}$ and $\mathcal{S}$ the sets of our test problems and optimization solvers, respectively. For each problem $p \in \mathcal{P}$ and solver $s \in \mathcal{S}$, we define $t_{p,s}$ as the running time required to solve problem p by solver s. Then $t_{best} := \min\{t_{p,s} \mid s \in \mathcal{S}\}$ is the minimum running time used by all solvers and

$$r_{p,s} := \begin{cases} \frac{t_{p,s}}{t_{best}} & \text{if the run is counted as success} \\ \infty & \text{otherwise} \end{cases}$$

the *performance ratio* of solver s for problem p. The *performance profile* for solver s is then defined as

$$\rho_s(\tau) = \frac{1}{|\mathcal{P}|} \left| \{ p \in \mathcal{P} \mid r_{p,s} \leq \tau \} \right|,$$

where $\tau \in \mathbb{R}$ and $|\cdot|$ denotes the set cardinality (i.e., $|\mathcal{P}| = 350$). That is, $\rho_s(\tau)$ is the probability for solver s that $r_{p,s}$ is within a factor τ of the best possible performance ratio. In fact, the function ρ_s is the cumulative distribution function for the performance ratio [10, p. 203]. The measure is easily changed from running time to the number of function evaluations by setting $t_{p,s}$ accordingly.

Remark. We observe that $\rho_s(1)$ is the percentage of test problems for which solver s has the best performance among all other the procedures; and for sufficiently large τ, $p_s(\tau)$ gives the percentage of problems that can be successfully minimized by solver s, i.e., its overall success rate, cf. Table 2.

The MATLAB file `perf.m` retrieved from [31] was used and customized in order to create the performance profiles. If an algorithm was unsuccessful on a given problem, its corresponding performance measures were simply set to NaN. Figures 3, 4, 5 and 6 were all plotted in logarithmic scale with base 2. Their non-logarithmic versions can be found as Figure 7 in Appendix B. The disadvantage of the profiles is that they only show performance with respect to the best algorithm [4, p. 17]. That is, for a comparison of the other solvers with each other, a different profile without the best method has to be drawn. This is the reason why we only consider four procedures (instead of six) in Figures 4 and 6.

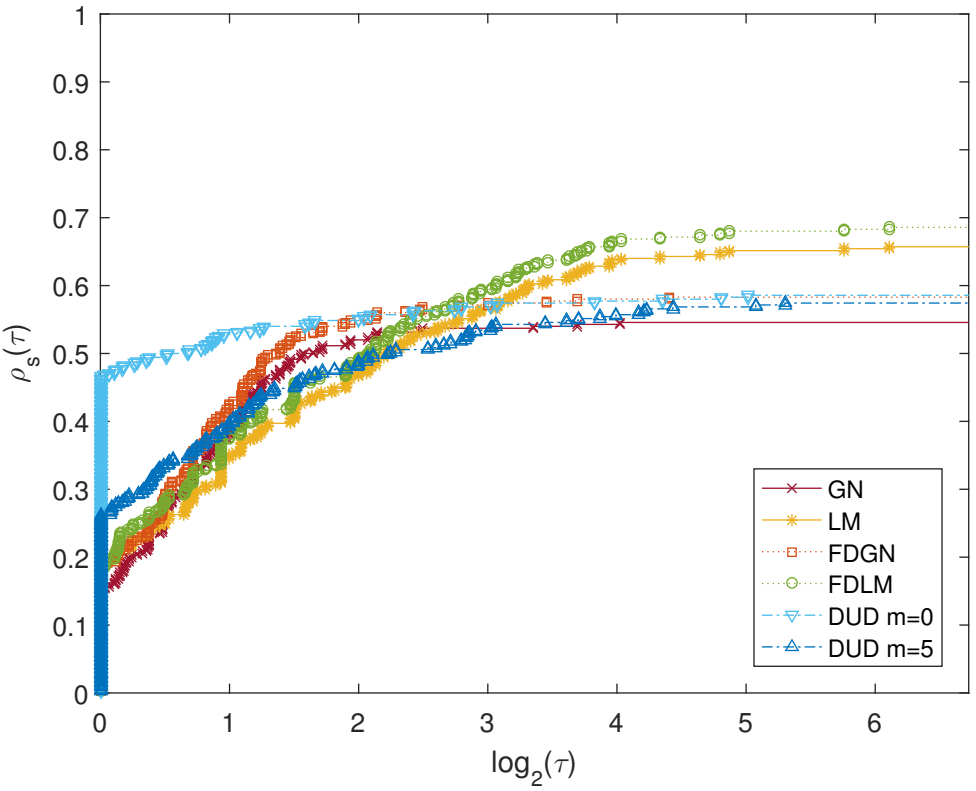


Figure 3: Performance profile 1 (equivalent function evaluations).

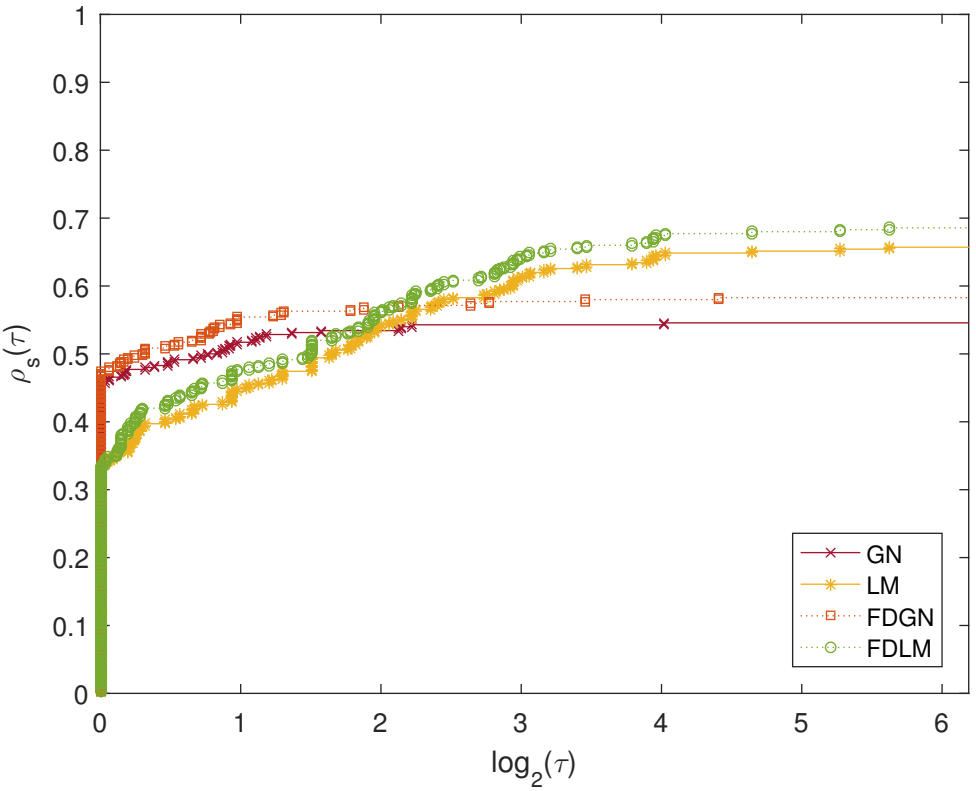


Figure 4: Performance profile 2 (equivalent function evaluations).

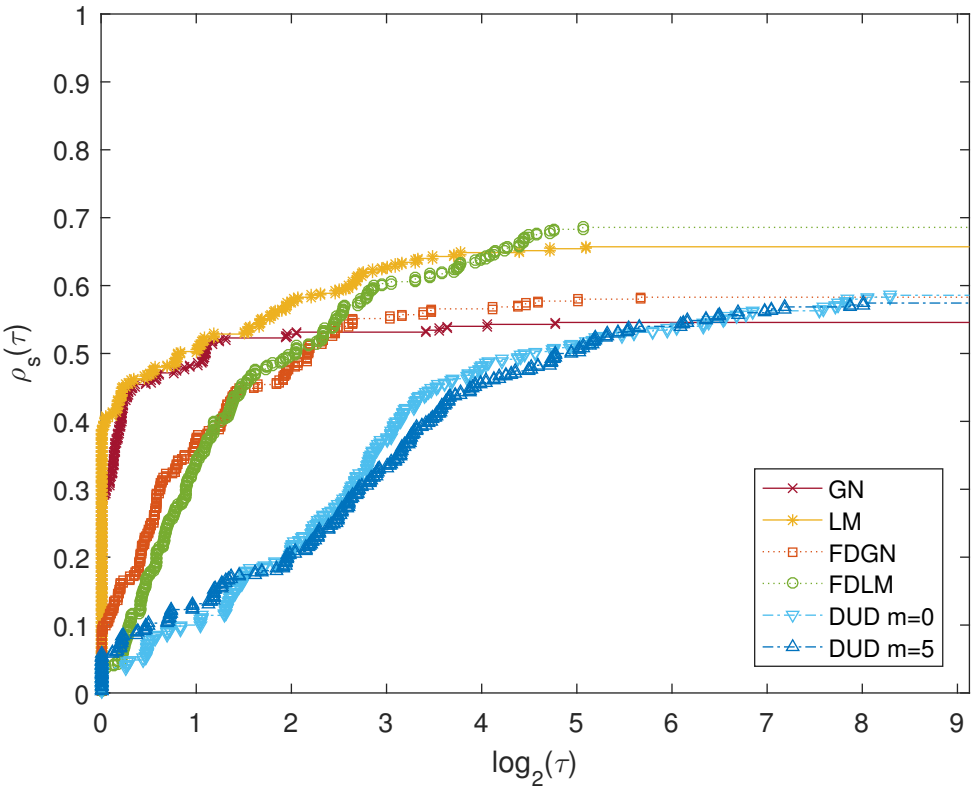


Figure 5: Performance profile 3 (average wall-clock times).

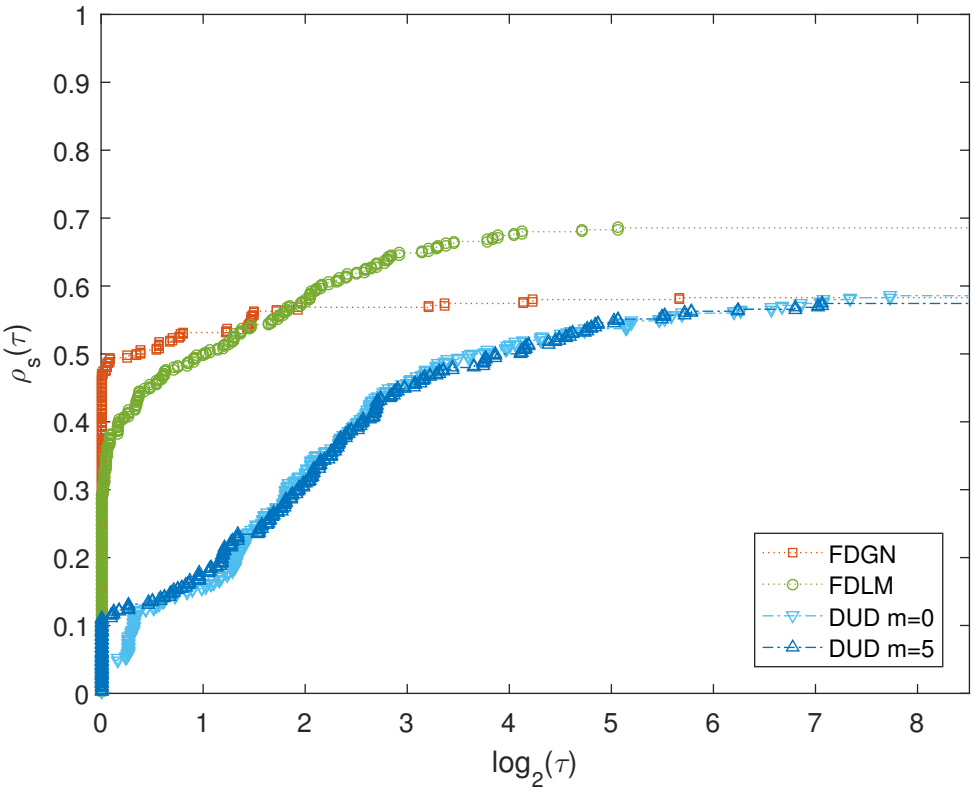


Figure 6: Performance profile 4 (average wall-clock times).

The performance profile 1 (see Figure 3) compares all six methods in terms of numbers of equivalent function evaluations on our test set of 350 problems. We can see that DUD m=0 has the best performance for 47% of the problems. That is, in order to solve 47% of the problems, DUD m=0 needs just as many or fewer function calls than the other five procedures. Although DUD m=5 solves approximately 26% of the problems with just as many or fewer function calls than its competitors, its performance never comes close to DUD m=0 as we increase τ. So, if we are interested in the algorithm that can solve about 50% of the problems with greatest efficiency, then DUD m=0 really stands out. However, if our scope of interest lies within a factor of $\tau = 8$ (i.e., $\log_2(\tau) = 3$) of the best solver, FDGN and FDLM become quite competitive. The method of choice for solving more than 58% of the problems with the least possible amount of function evaluations is FDLM. Given large enough τ, we note that the percentages of the probabilities $\rho_s(\tau)$ coincide with the success rates from Table 2.

Profile 2 (see Figure 4) depicts the differences in performance between the two gradient methods and their derivative-free counterparts. We can observe a slight advantage of FDGN and FDLM over the GN and LM algorithms, respectively. The GN method is able solve about 46% of the problems with just as many or fewer function evaluations than the other procedures, whereas FDGN does even better with 48% wins. As in Figure 3, the FDLM can not compete in terms of wins but dominates the LM algorithm when it comes to reliable solving.

Performance profile 3 (see Figure 5) illustrates the complete comparison in terms running times. Since we provided analytic Jacobians, it is no surprise that the gradient methods are the frontrunners. The LM algorithm is able to solve 41% as fast or faster than the other procedures. Here, its only real competitor is the GN method, which has almost 30% of the wins. However, if the solving time is relaxed to be within a factor of $\tau = 16$ (i.e., $\log_2(\tau) = 4$), we can clearly see that LM performs best. FDLM is slower but can roughly solve 3% more problems than the LM algorithm.

In profile 4 (see Figure 6), the fast gradient-based algorithms are excluded (due to their unfair advantage of user-supplied Jacobians), i.e., only the derivative-free methods are plotted. In terms of running times, both DUD m=0 and DUD m=5 stand no chance versus FDGN or FDLM. The FDGN has the best performance for 48% of the problems. In comparison, the FDLM is only on about 30% of the problems as fast or faster than its competitors but challenges FDGN if performance ratios are allowed up to to a factor of $\tau = 4$, i.e., $\log_2(\tau) = 2$. And if $\tau > 16$, i.e., $\log_2(\tau) > 4$, it can approximately solve 10% more problems from the test set than FDGN.

6 Summary and Conclusion

The benchmark of our procedures reveals that, for low dimensional problems, FDGN and FDLM can be used without hesitation instead of the GN and LM algorithms, respectively. Only a slight difference in the qualitative results is to be expected. In fact, for medium accuracy, both FDGN and FDLM even show better performance than their gradient counterparts in terms of number of function calls and reliability. However, the GN and LM algorithms are expected to be better suited for problems of higher dimensions and for high accuracy levels.
Since we conducted experiments on 350 test problems of varying degrees of difficulty, the acquired success rates are quite representative for the algorithmic reliability. This means that LM and FDLM are the clear choices if the researcher values robust methods the most; FDLM is to be preferred when derivatives are not easily accessible.

When efficiency is valued over robustness and function evaluation is expensive, the DUD algorithm (with turned-off line search option) is the best choice. Ralston and Jennrich [44, p. 13] used only small residual problems for their tests, so they warned us that DUD could look artificially good. But its performance on our relatively large test set has exceeded our expectations. DUD beats both the GN method and FDGN in terms of efficient use of function evaluations and reliability. However, we learned that enabling DUD's optional line search results in additional function calls but does not lead to a higher success rate or faster convergence. Nevertheless, if the step shortening procedure is activated, alternative solution paths may be found. Occasionally, different choices of m lead to different solutions. Since DUD's iteration process is more involved than that of GN or FDGN, it might perform worse than these methods on small residual problems, provided that function evaluation is cheap. While the overall success rate of DUD is slightly better than that of GN or FDGN, the algorithm has similar types of difficulty when minimizing large residual problems.

The stepwise regression procedure employed by DUD performs satisfactorily. However, we note that the underlying Gauss-Jordan algorithm could be more efficient. The number of arithmetic operations can be reduced by almost one half by exploiting the symmetry of the positive definite input matrix. But we decided to go with Algorithm 6 from Section 4.3.1, since MATLAB offers the possibility to vectorize the inner loops of the program, yielding an efficient implementation.

Ralston and Jennrich mentioned that an updating procedure could be used to reduce the number of arithmetic operations of DUD, see [44, p.8].
Brown and Dennis [6, p. 290] pointed out that Powell [42] replaces the classical LM step (4.16) from Section 4.2 with a technique in which Broyden's single rank update [7] is used to approximate the Jacobian at each iteration point. The resulting algorithm is thus very similar to LM and is known as *Powell's hybrid method*.

This page intentionally left blank.

Appendix A Prerequisites

The following mathematical concepts are well known and therefore, for the most part, stated without proofs and further explanation. Full details and extra information can be found in [14, p. 24–29] and [15, p. 63–88, 159–160].

A.1 Matrix Analysis

Definition A.1. The *transpose* of a matrix $A \in \mathbb{R}^{m\times n}$ is the matrix $A^\mathsf{T} \in \mathbb{R}^{n\times m}$ defined by

$$\left(A^\mathsf{T}\right)_{ij} = \left(A\right)_{ji}.$$

Definition A.2. A square matrix $A \in \mathbb{R}^{n\times n}$ is called *symmetric* if $A = A^\mathsf{T}$.

Proposition A.3. *If a matrix $A \in \mathbb{R}^{n\times n}$ is symmetric, then its singular values are the absolute values of its nonzero eigenvalues.*

Definition A.4. A symmetric matrix $A \in \mathbb{R}^{n\times n}$ is called *positive definite* if

$$x^\mathsf{T} Ax > 0 \text{ for all } x \in \mathbb{R}^n,\ x \neq 0,$$

and *negative definite* if

$$x^\mathsf{T} Ax < 0 \text{ for all } x \in \mathbb{R}^n,\ x \neq 0.$$

It is called *positive semidefinite* if

$$x^\mathsf{T} Ax \geq 0 \text{ for all } x \in \mathbb{R}^n$$

and *negative semidefinite* if

$$x^\mathsf{T} Ax \leq 0 \text{ for all } x \in \mathbb{R}^n\,.$$

A matrix which is neither positive or negative semidefinite is called *indefinite*. If the matrix A is positive definite, negative definite, positive semidefinite or negative semidefinite, its eigenvalues of are positive, negative, nonnegative or nonpositive, respectively. An indefinite matrix has both positive and negative eigenvalues.

Definition A.5. A *principal submatrix* of a square matrix $A \in \mathbb{R}^{n\times n}$ is any square submatrix sharing some diagonal elements of A.

Proposition A.6. *If a matrix $A \in \mathbb{R}^{n\times n}$ is positive definite, then all its principal submatrices are positive definite. In particular, all diagonal entries are positive.*

Proposition A.7. *If a matrix $A \in \mathbb{R}^{n\times n}$ is positive semidefinite, then all its principal submatrices are positive semidefinite. In particular, all diagonal entries are nonnegative.*

Definition A.8. An *inverse* of a square matrix $A \in \mathbb{R}^{n\times n}$ is a matrix $A^{-1} \in \mathbb{R}^{n\times n}$ such that

$$AA^{-1} = A^{-1}A = I,$$

where $I \in \mathbb{R}^{n\times n}$ is the *identity matrix*. If a matrix has an inverse it is called *invertible* or *nonsingular*. Otherwise, it is called *singular*.

Proposition A.9. *If a matrix $A \in \mathbb{R}^{n\times n}$ is invertible, all its eigenvalues are nonzero, and the eigenvalues of the inverse A^{-1} are the reciprocals of the eigenvalues of A.*

Theorem A.10. *If a matrix $A \in \mathbb{R}^{n\times n}$ is positive definite, then A is invertible and A^{-1} is positive definite.*

Definition A.11. A *vector norm* on $\mathbb{R}^n$ is a function $\|\cdot\| : \mathbb{R}^n \to \mathbb{R}$ such that

(i) $\|x\| \geq 0$, $x \in \mathbb{R}^n$, and $\|x\| = 0$ if and only if $x = 0$,

(ii) $\|\alpha x\| = |\alpha| \cdot \|x\|$, $\alpha \in \mathbb{R}, x \in \mathbb{R}^n$,

(iii) $\|x + y\| \leq \|x\| + \|y\|$, $x, y \in \mathbb{R}^n$.

Definition A.12. The *p-norm* of $x \in \mathbb{R}^n$ is defined by

$$\|x\|_p = \Big(\sum_{i=1}^{n} |x_i|^p\Big)^{\frac{1}{p}}, \quad p \in \mathbb{R},\ 1 \leq p < \infty,$$

and the *infinity* or *maximum norm* of x is

$$\|x\|_\infty = \max_{1\leq i\leq n} |x_i|.$$

Definition A.13. Let $P \in \mathbb{R}^{n\times n}$ be a symmetric positive definite matrix. For a vector $x \in \mathbb{R}^n$, the *ellipsoidal (vector) norm associated to P* is defined by

$$\|x\|_P := \sqrt{x^\mathsf{T} P x}.$$

Remark. Since $\mathbb{R}^{m\times n}$ is isomorphic to $\mathbb{R}^{mn}$, a *matrix norm* $\|\cdot\| : \mathbb{R}^{m\times n} \to \mathbb{R}$ is defined analogous to Definition A.11. It should be clear within the context if a vector norm or a matrix norm is used.

A matrix norm can also be defined in terms of a vector norm. Let $\|\cdot\|$ be a vector norm and consider $\|Ax\|$ for a matrix $A \in \mathbb{R}^{m\times n}$ and for all vectors $x \in \mathbb{R}^n$ with $\|x\| = 1$. Then the matrix norm *induced* by the vector norm is given by

$$\|A\| = \max_{\|x\|=1} \|Ax\|.$$

Thus, the induced *matrix p-norms* are

$$\|A\|_p = \max_{\|x\|_p=1} \|Ax\|_p.$$

In particular, it is a fact that the matrix 2-norm of $A \in \mathbb{R}^{m\times n}$ can be expressed as

$$\|A\|_2 = \sqrt{\lambda_{\max}(A^\mathsf{T} A)} = \sigma_{\max}(A), \tag{A.1.1}$$

where $\lambda_{\max}$ is the the largest eigenvalue of $A^\mathsf{T}A$ and $\sigma_{max}(A)$ is the largest singular value of A. If $m \geq n$ and A has full column rank n, it can also be shown that

$$\|A^\dagger\|_2 = \frac{1}{\sigma_{\min}(A)}, \tag{A.1.2}$$

where $A^\dagger := (A^\mathsf{T}A)^{-1}A^\mathsf{T}$ denotes the *pseudo-inverse* of A and $\sigma_{\min}(A)$ is the smallest singular value of A. We observe that $A^\dagger = A^{-1}$ for invertible $A \in \mathbb{R}^{n\times n}$.

There exist linear systems $Ax = b$ whose solutions are not stable under small perturbations of either A or b. The *condition number* of A indicates the maximum effect of perturbations in A or b on the exact solution x. For an *ill-conditioned* problem the solution of a perturbed problem may be very different to the exact solution. That is, small changes in the input (the matrix A and b) can cause a large change in the output (the solution of $Ax = b$).

Remark. The solution of a linear system $Ax = b$ may be poor if the coefficient matrix A is close to being singular.

Definition A.14. The *condition number* $\kappa(A)$ of a nonsingular square matrix $A \in \mathbb{R}^{n\times n}$ is defined by

$$\kappa(A) = \|A\| \cdot \|A^{-1}\|,$$

where the convention $\kappa(A) = \infty$ is used for singular A.

Definition A.15. The *condition number* $\kappa(A)$ of a rectangular matrix $A \in \mathbb{R}^{m\times n}$ $(m \geq n)$ of full column rank n, is defined by

$$\kappa(A) = \|A\| \cdot \|A^\dagger\|.$$

Remark. The notation $\kappa_i,\ i \in \{1, \ldots, \infty\}$, is used to stress the associated norm.

Definition A.16. A matrix is said to be *ill-conditioned* if its condition number is large and *well-conditioned* if its condition number is small.

The notion of the terms "large" and "small" strongly depends on the underlying problem. Due to equations (A.1.1) and (A.1.2), the condition number of a matrix $A \in \mathbb{R}^{m\times n}$ $(m \geq n)$ with column rank n is given in terms of the matrix 2-norm by

$$\kappa_2(A) = \|A\|_2 \cdot \|A^\dagger\|_2 = \frac{\sigma_{\max}(A)}{\sigma_{min}(A)}. \tag{A.1.3}$$

Remark. The larger the condition number of a matrix is, the more sensitive the matrix is to inverse calculation.

Proposition A.17. *For any induced matrix norm $\|\cdot\|$ it holds that*

$$\|AB\| \leq \|A\| \cdot \|B\|, \quad A \in \mathbb{R}^{m\times n},\ B \in \mathbb{R}^{n\times q}.$$

Remark. This inequality is known as *submultiplicative property.* It describes in fact a relationship between three different matrix norms since they act on three different spaces.

Corollary A.18. *For any induced matrix norm it holds that*

$$\kappa(A) \geq 1,$$

where $A \in \mathbb{R}^{m\times n}$ $(m \geq n)$ has column rank n.

A.2 Error Analysis

Definition A.19. Suppose that $\hat{x} \in \mathbb{R}^n$ is an approximation to $x \in \mathbb{R}^n$. Then

$$\hat{x} - x$$

is said to be the *error* in $\hat{x}$. For a given vector norm $\|\cdot\|$, the non-negative number

$$\|\hat{x} - x\|$$

is denoted as the *absolute error* in $\hat{x}$. If $x \neq 0$, the *relative error* in $\hat{x}$ is defined as

$$\frac{\|\hat{x} - x\|}{\|x\|}.$$

Remark. The relative error is undefined for $x = 0$. If the exact value x is close to 0, the measure

$$\frac{\|\hat{x} - x\|}{1 + \|x\|}$$

should be used instead of the relative error.

Number representation on computers and error analysis of algorithms are well described in [14, p. 7–14].

A.3 Convergence

Definition A.20. A sequence of vectors $x^l \in \mathbb{R}^n$ *converges* to $x \in \mathbb{R}^n$ if

$$\lim_{l\to\infty} \left\|x^l - x\right\| = 0$$

for a given vector norm $\|\cdot\|$. This is also denoted by

$$\lim_{l\to\infty} x^l = x \quad \text{or} \quad x^l \to x \text{ for } l \to \infty.$$

Remark. All norms on $\mathbb{R}^n$ are equivalent. Therefore, convergence in any particular norm implies convergence in all norms.

Definition A.21. A sequence of vectors $x^l \in \mathbb{R}^n$ converges with *order* q to $x \in \mathbb{R}^n$ if q is the largest number such that

$$0 \leq \lim_{l\to\infty} \frac{\left\|x^{l+1} - x\right\|}{\left\|x^l - x\right\|^q} < \infty.$$

The power q is also known as *asymptotic rate of convergence*. The sequence is said to have *linear convergence* if $q = 1$, *quadratic convergence* if $q = 2$ and *cubic convergence* if $q = 3$. If the sequence has order of convergence q, the limit

$$\gamma = \lim_{l\to\infty} \frac{\left\|x^{l+1} - x\right\|}{\left\|x^l - x\right\|^q}$$

is called the *asymptotic error constant*. Convergence is called *superlinear* when $q = 1$ and $\gamma = 0$ and *sublinear* when $q = 1$ and $\gamma = 1$.

When $q = 1$, convergence only occurs if $\gamma < 1$. Also, if a sequence of vectors converges with order $q > 1$, superlinear convergence is implied.

Remark. An asymptotic rate of convergence q is not necessarily integer.

Definition A.22. A function $F : \mathcal{D} \subset \mathbb{R}^n \to \mathbb{R}^m$ is *Lipschitz continuous* on some set $\mathcal{N} \subset \mathcal{D}$ if there is a constant $L \geq 0$ such that

$$\|F(x) - F(y)\| \leq L\|x - y\|, \quad \forall x, y \in \mathcal{N}. \tag{A.3.1}$$

L is called the *Lipschitz constant*. The function F is locally Lipschitz continuous at a point $\hat{x} \in \operatorname{int} D$ if (A.3.1) holds for some neighborhood $\mathcal{N} \subset \mathcal{D}$ of $\hat{x}$.

The following theorem is crucial for continuous optimization. It provides information about the existence of solutions for problems. Iterative methods construct sequences that are aimed for convergence to such solutions.

Theorem A.23 (Existence of solutions)**.** *Suppose we have a problem of the form*

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & x \in \mathcal{C}, \end{aligned} \tag{A.3.2}$$

where $f : \mathcal{D} \subset \mathbb{R}^n \to \mathbb{R}^n$ is continuous on $\mathcal{C} \subset \mathcal{D}$.

(i) If $\mathcal{C}$ is nonempty and compact, then the optimization problem (A.3.2) has a global solution.

(ii) Let $\mathcal{C}_0 \subset \mathcal{C}$ be compact with nonempty interior. If there is a vector $x^0 \in \mathcal{C}_0$ such that

$$f(x) > f(x^0), \ \forall x \in \partial\mathcal{C}_0, \tag{A.3.3}$$

then (A.3.2) has a local solution.

Proof. This proof is extracted from [38, p. 16]. Statement (i) holds since any continuous function attains its infimum on every nonempty compact set.
(ii) Since $\mathcal{C}_0$ is compact, the function f attains its minimum in $\mathcal{C}_0$ at some $x^* \in \mathcal{C}_0$. Condition (A.3.3) implies that $x^* \in \operatorname{int} \mathcal{C}_0$. Therefore, $\mathcal{C}_0$ contains a neighborhood of x^* such that (A.3.2) holds. □

Remark. A standard requirement for (A.3.3) to hold is that the level set

$$\mathcal{C}_0 := \{x \in \mathcal{C} \mid f(x) \leq f(x^0)\}$$

is in $\operatorname{int} \mathcal{C}$ and bounded (and hence compact).

Definition A.24. An iterative method is called *locally convergent* if it produces a sequence of vectors $x^l \in \mathbb{R}^n$ which converges towards a minimizer $x^* \in \mathbb{R}^n$ provided a *close enough* starting approximation. It is called *globally convergent* if the sequence x^l converges towards x^* provided *any* starting approximation.

Remark. The point x^* may be a local or global minimum. Hence, it is important to note that global convergence does not imply convergence towards a global minimizer.

Having a descent sequence (see (2.7) on page 14) in an iterative method does not need to imply convergence to a local minimizer. In fact, it does not even need to imply convergence to a stationary point. The following two general statements about descent sequences are from [38, p. 74–77].

Proposition A.25. *Let $f : \mathcal{D} \subset \mathbb{R}^n \to \mathbb{R}$ be continuous. Then, for any descent sequence $x^l \in \mathcal{D}$, one of the following holds:*

(i) There is a subsequence x^{l_η} $(l_0 < l_1 < \dots)$ with $\|x^{l_\eta}\| \to \infty$ as $\eta \to \infty$.

(ii) There is a subsequence which converges to a limit $\notin \mathcal{D}$.

(iii) It holds that

$$\inf_{l \geq 0} f(x^l) =: \bar{f} < \infty$$

and the level set $\{x \in \mathcal{D} \mid f(x) = \bar{f}\}$ contains a sequence $\bar{x}^l$ such that

$$\|x^l - \bar{x}^l\| \to 0 \text{ as } l \to \infty.$$

Remark. Conditions (i) and (ii) are not true if the level set $\{x \in \mathcal{D} \mid f(x) \leq f(x^0)\}$ is compact. The infimum in (iii) needs not to be a minimum of f in $\mathcal{D}$. We need to impose stronger conditions in order to guarantee convergence to a stationary point.

Proposition A.26. *Let $f : \mathcal{D} \subset \mathbb{R}^n \to \mathbb{R}$ be continuously differentiable on the compact set $\mathcal{C} \subset \mathcal{D}$ and let x^l be a descent sequence in $\mathcal{C}$.*

(i) If any two stationary points x, $\bar{x} \in \mathcal{C}$ with $x \neq \bar{x}$ satisfy $f(x) \neq f(\bar{x})$, and if for a subsequence x^{l_η}

$$\lim_{\eta \to \infty} \big\|\nabla f(x^{l_\eta})\big\| = 0,$$

then x^{l_η} converges to a stationary point in $\mathcal{C}$.

(ii) If $\bar{f} \leq f(x^0)$, if the set of stationary points $\bar{x} \in \mathcal{C}$ with $f(\bar{x}) = \bar{f}$ is finite, and if

$$\lim_{l \to \infty} \big\|\nabla f(x^l)\big\| = \lim_{l \to \infty} \big\|x^{l+1} - x^l\big\| = 0,$$

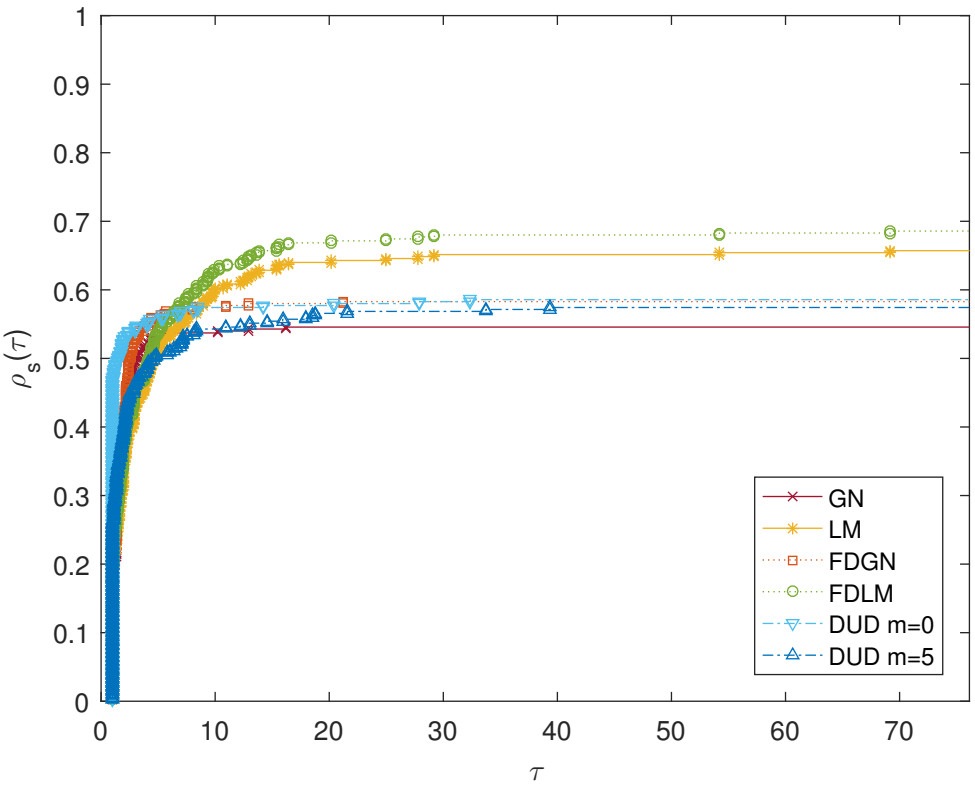
then x^l converges to a stationary point in $\mathcal{C}$.

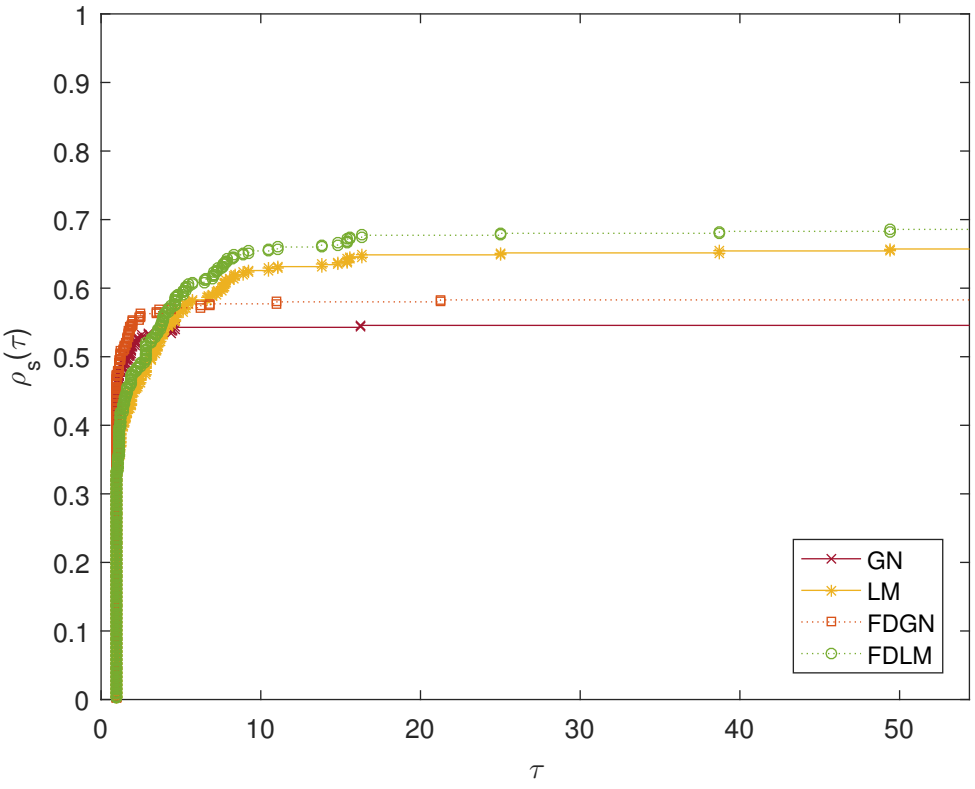
(iii) If the global minimizer x^* is the unique stationary point in $\mathcal{C}$, and if

$$\inf_{l \geq 0} \big\|\nabla f(x^l)\big\| = 0,$$

then x^l converges to x^*.

Appendix B Test Suite





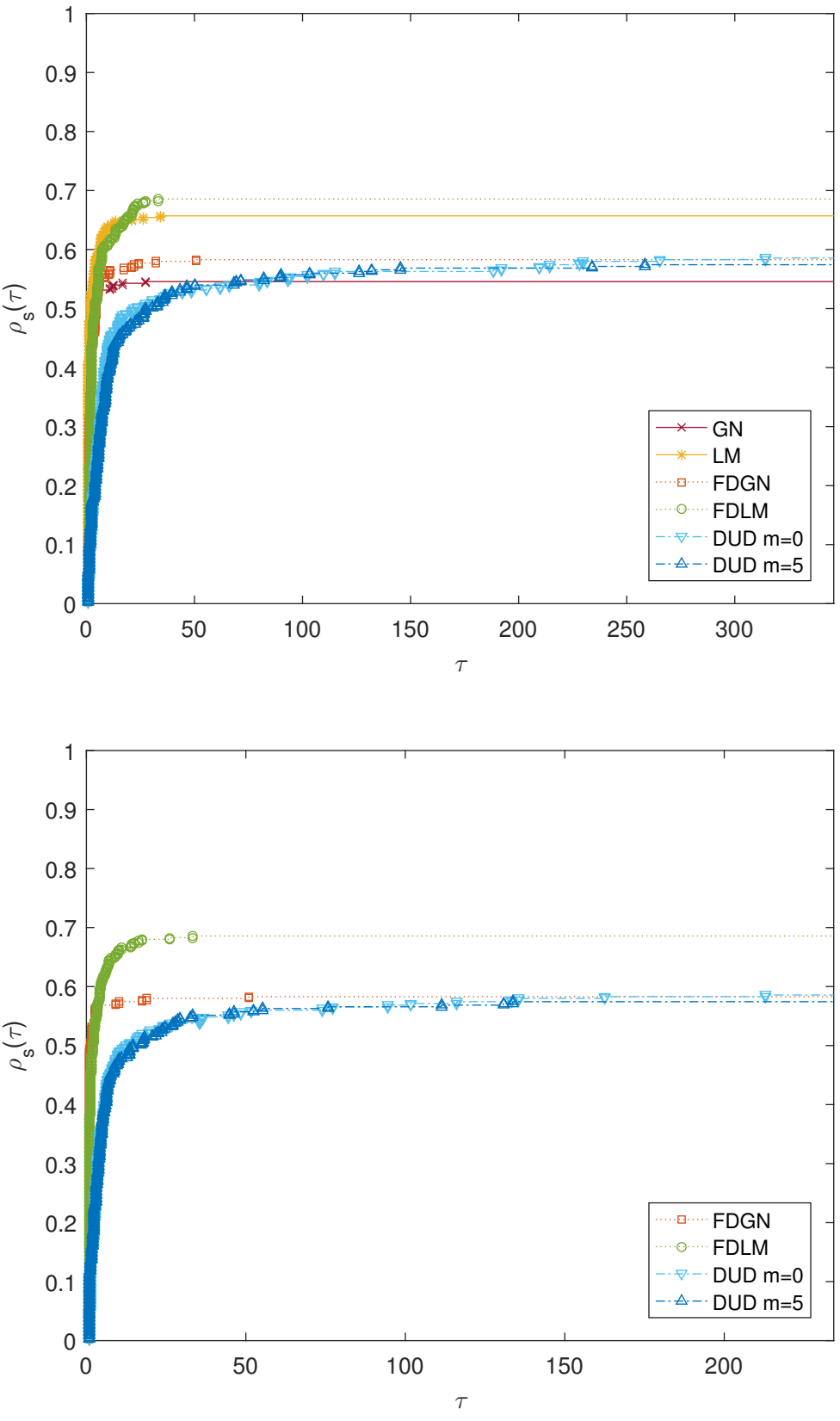


Figure 7: Non-logarithmic performance profiles.
The non-log versions of Figures 3, 4, 5 and 6, respectively.

Table 5: Approximate starting points used for each test function.

1 Rosenbrock

1	2	3	4	5
−1.20000e+00	−5.70553e−01	−1.94603e+00	−9.35282e−01	5.09447e+00
1.00000e+00	1.81158e+00	1.82675e+00	1.95081e−01	9.11584e+00
6	**7**	**8**	**9**	**10**
−8.66026e+00	1.44719e+00	6.17447e+01	−7.58026e+01	2.52719e+01
9.26752e+00	−7.04919e+00	8.21584e+01	8.36752e+01	−7.94919e+01

2 Freudenstein and Roth

1	2	3	4	5
5.00000e−01	1.12945e+00	−2.46026e−01	7.64719e−01	6.79447e+00
−2.00000e+00	−1.18842e+00	−1.17325e+00	−2.80492e+00	6.11584e+00
6	**7**	**8**	**9**	**10**
−6.96026e+00	3.14719e+00	6.34447e+01	−7.41026e+01	2.69719e+01
6.26752e+00	−1.00492e+01	7.91584e+01	8.06752e+01	−8.24919e+01

3 Powell badly scaled

1	2	3	4	5
0.00000e+00	6.29447e−01	−7.46026e−01	2.64719e−01	6.29447e+00
1.00000e+00	1.81158e+00	1.82675e+00	1.95081e−01	9.11584e+00
6	**7**	**8**	**9**	**10**
−7.46026e+00	2.64719e+00	6.29447e+01	−7.46026e+01	2.64719e+01
9.26752e+00	−7.04919e+00	8.21584e+01	8.36752e+01	−7.94919e+01

4 Brown badly scaled

5 Beale

1	2	3	4	5
1.00000e+00	1.62945e+00	2.53974e−01	1.26472e+00	7.29447e+00
1.00000e+00	1.81158e+00	1.82675e+00	1.95081e−01	9.11584e+00
6	**7**	**8**	**9**	**10**
−6.46026e+00	3.64719e+00	6.39447e+01	−7.36026e+01	2.74719e+01
9.26752e+00	−7.04919e+00	8.21584e+01	8.36752e+01	−7.94919e+01

6 Jennrich and Sampson

1	2	3	4	5
3.00000e−01	9.29447e−01	−4.46026e−01	5.64719e−01	6.59447e+00
4.00000e−01	1.21158e+00	1.22675e+00	−4.04919e−01	8.51584e+00
6	**7**	**8**	**9**	**10**
−7.16026e+00	2.94719e+00	6.32447e+01	−7.43026e+01	2.67719e+01
8.66752e+00	−7.64919e+00	8.15584e+01	8.30752e+01	−8.00919e+01

7 Helical Valley

1	2	3	4	5
−1.00000e+00	−1.44300e+00	−7.02229e−02	−8.56661e−02	−5.43004e+00
0.00000e+00	9.37630e−02	−6.84774e−01	−2.92487e−02	9.37630e−01
0.00000e+00	9.15014e−01	9.41186e−01	6.00561e−01	9.15014e+00
6	**7**	**8**	**9**	**10**
8.29777e+00	8.14334e+00	−4.53004e+01	9.19777e+01	9.04334e+01
−6.84774e+00	−2.92487e−01	9.37630e+00	−6.84774e+01	−2.92487e+00
9.41186e+00	6.00561e+00	9.15014e+01	9.41186e+01	6.00561e+01

8 Bard

1	2	3	4	5
1.00000e+00	5.56996e−01	1.92978e+00	1.91433e+00	−3.43004e+00
1.00000e+00	1.09376e+00	3.15226e−01	9.70751e−01	1.93763e+00
1.00000e+00	1.91501e+00	1.94119e+00	1.60056e+00	1.01501e+01
6	**7**	**8**	**9**	**10**
1.02978e+01	1.01433e+01	−4.33004e+01	9.39777e+01	9.24334e+01
−5.84774e+00	7.07513e−01	1.03763e+01	−6.74774e+01	−1.92487e+00
1.04119e+01	7.00561e+00	9.25014e+01	9.51186e+01	6.10561e+01

9 Gaussian

1	2	3	4	5
4.00000e−01	3.55700e−01	4.92978e−01	4.91433e−01	−4.30036e−02
1.00000e+00	1.00938e+00	9.31523e−01	9.97075e−01	1.09376e+00
0.00000e+00	9.15014e−02	9.41186e−02	6.00561e−02	9.15014e−01
6	**7**	**8**	**9**	**10**
1.32978e+00	1.31433e+00	−4.03004e+00	9.69777e+00	9.54334e+00
3.15226e−01	9.70751e−01	1.93763e+00	−5.84774e+00	7.07513e−01
9.41186e−01	6.00561e−01	9.15014e+00	9.41186e+00	6.00561e+00

10 Meyer

1	2	3	4	5
2.00000e−02	−4.23004e−01	9.49777e−01	9.34334e−01	−4.41004e+00
4.00000e+03	4.00009e+03	3.99932e+03	3.99997e+03	4.00094e+03
2.50000e+02	2.50915e+02	2.50941e+02	2.50601e+02	2.59150e+02
6	**7**	**8**	**9**	**10**
9.31777e+00	9.16334e+00	−4.42804e+01	9.29977e+01	9.14534e+01
3.99315e+03	3.99971e+03	4.00938e+03	3.93152e+03	3.99708e+03
2.59412e+02	2.56006e+02	3.41501e+02	3.44119e+02	3.10056e+02

11 Gulf research and development

1	2	3	4	5
5.00000e+00	4.99557e+00	5.00930e+00	5.00914e+00	4.95570e+00
2.50000e+00	2.50094e+00	2.49315e+00	2.49971e+00	2.50938e+00
1.50000e−01	1.59150e−01	1.59412e−01	1.56006e−01	2.41501e−01
6	**7**	**8**	**9**	**10**
5.09298e+00	5.09143e+00	4.55700e+00	5.92978e+00	5.91433e+00
2.43152e+00	2.49708e+00	2.59376e+00	1.81523e+00	2.47075e+00
2.44119e−01	2.10056e−01	1.06501e+00	1.09119e+00	7.50561e−01

12 Box three-dimensional

1	2	3	4	5
0.00000e+00	−4.43004e−01	9.29777e−01	9.14334e−01	−4.43004e+00
1.00000e+01	1.00938e+01	9.31523e+00	9.97075e+00	1.09376e+01
2.00000e+01	2.09150e+01	2.09412e+01	2.06006e+01	2.91501e+01
6	**7**	**8**	**9**	**10**
9.29777e+00	9.14334e+00	−4.43004e+01	9.29777e+01	9.14334e+01
3.15226e+00	9.70751e+00	1.93763e+01	−5.84774e+01	7.07513e+00
2.94119e+01	2.60056e+01	1.11501e+02	1.14119e+02	8.00561e+01

13 Powell singular

1	2	3	4	5
3.00000e+00	2.28377e+00	3.91899e+00	3.86799e+00	−4.16227e+00
−1.00000e+00	−1.15648e+00	−6.88519e−01	−6.42530e−01	−2.56477e+00
0.00000e+00	8.31471e−01	−9.28577e−01	5.15480e−01	8.31471e+00
1.00000e+00	1.58442e+00	1.69826e+00	1.48627e+00	6.84415e+00
6	**7**	**8**	**9**	**10**
1.21899e+01	1.16799e+01	−6.86227e+01	9.48985e+01	8.97987e+01
2.11481e+00	2.57470e+00	−1.66477e+01	3.01481e+01	3.47470e+01
−9.28577e+00	5.15480e+00	8.31471e+01	−9.28577e+01	5.15480e+01
7.98259e+00	5.86265e+00	5.94415e+01	7.08259e+01	4.96265e+01

14 Wood

1	2	3	4	5
−3.00000e+00	−3.71623e+00	−2.08102e+00	−2.13201e+00	−1.01623e+01
−1.00000e+00	−1.15648e+00	−6.88519e−01	−6.42530e−01	−2.56477e+00
−3.00000e+00	−2.16853e+00	−3.92858e+00	−2.48452e+00	5.31471e+00
−1.00000e+00	−4.15585e−01	−3.01741e−01	−5.13735e−01	4.84415e+00
6	**7**	**8**	**9**	**10**
6.18985e+00	5.67987e+00	−7.46227e+01	8.88985e+01	8.37987e+01
2.11481e+00	2.57470e+00	−1.66477e+01	3.01481e+01	3.47470e+01
−1.22858e+01	2.15480e+00	8.01471e+01	−9.58577e+01	4.85480e+01
5.98259e+00	3.86265e+00	5.74415e+01	6.88259e+01	4.76265e+01

15 Kowalik and Osborne

1	2	3	4	5
2.50000e−01	1.78377e−01	3.41899e−01	3.36799e−01	−4.66227e−01
3.90000e−01	3.74352e−01	4.21148e−01	4.25747e−01	2.33523e−01
4.15000e−01	4.98147e−01	3.22142e−01	4.66548e−01	1.24647e+00
3.90000e−01	4.48442e−01	4.59826e−01	4.38627e−01	9.74415e−01
6	**7**	**8**	**9**	**10**
1.16899e+00	1.11799e+00	−6.91227e+00	9.43985e+00	8.92987e+00
7.01481e−01	7.47470e−01	−1.17477e+00	3.50481e+00	3.96470e+00
−5.13577e−01	9.30480e−01	8.72971e+00	−8.87077e+00	5.56980e+00
1.08826e+00	8.76265e−01	6.23415e+00	7.37259e+00	5.25265e+00

16 Brown and Dennis

1	2	3	4	5
2.50000e+01	2.42838e+01	2.59190e+01	2.58680e+01	1.78377e+01
5.00000e+00	4.84352e+00	5.31148e+00	5.35747e+00	3.43523e+00
−5.00000e+00	−4.16853e+00	−5.92858e+00	−4.48452e+00	3.31471e+00
−1.00000e+00	−4.15585e−01	−3.01741e−01	−5.13735e−01	4.84415e+00
6	**7**	**8**	**9**	**10**
3.41899e+01	3.36799e+01	−4.66227e+01	1.16899e+02	1.11799e+02
8.11481e+00	8.57470e+00	−1.06477e+01	3.61481e+01	4.07470e+01
−1.42858e+01	1.54803e−01	7.81471e+01	−9.78577e+01	4.65480e+01
5.98259e+00	3.86265e+00	5.74415e+01	6.88259e+01	4.76265e+01

17 Osborne 1

1	2	3	4	5
5.00000e−01	4.99785e−01	4.99554e−01	4.99634e−01	4.97845e−01
1.50000e+00	1.50031e+00	1.49909e+00	1.50090e+00	1.50311e+00
−1.00000e+00	−1.00066e+00	−1.00081e+00	−1.00093e+00	−1.00658e+00
1.00000e−02	1.04121e−02	1.06469e−02	9.87749e−03	1.41209e−02
2.00000e−02	1.90637e−02	2.03897e−02	1.97631e−02	1.06367e−02
6	**7**	**8**	**9**	**10**
4.95539e−01	4.96342e−01	4.78445e−01	4.55385e−01	4.63420e−01
1.49092e+00	1.50900e+00	1.53110e+00	1.40923e+00	1.59004e+00
−1.00806e+00	−1.00931e+00	−1.06576e+00	−1.08057e+00	−1.09311e+00
1.64692e−02	8.77489e−03	5.12092e−02	7.46916e−02	−2.25113e−03
2.38966e−02	1.76312e−02	−7.36334e−02	5.89657e−02	−3.68831e−03

18 Biggs EXP6

1	2	3	4	5
1.00000e+00	1.05310e+00	1.04187e+00	9.23800e−01	1.53103e+00
2.00000e+00	2.05904e+00	2.05094e+00	1.99967e+00	2.59040e+00
1.00000e+00	9.37375e−01	9.55205e−01	1.09195e+00	3.73745e−01
1.00000e+00	9.97953e−01	1.03594e+00	9.68077e−01	9.79529e−01
1.00000e+00	9.89117e−01	1.03102e+00	1.01705e+00	8.91172e−01
1.00000e+00	1.02926e+00	9.32522e−01	9.44762e−01	1.29263e+00
6	**7**	**8**	**9**	**10**
1.41873e+00	2.37995e−01	6.31034e+00	5.18730e+00	−6.62005e+00
2.50937e+00	1.99673e+00	7.90400e+00	7.09373e+00	1.96728e+00
5.52050e−01	1.91950e+00	−5.26255e+00	−3.47950e+00	1.01949e+01
1.35941e+00	6.80772e−01	7.95288e−01	4.59405e+00	−2.19229e+00
1.31020e+00	1.17054e+00	−8.82760e−02	4.10196e+00	2.70536e+00
3.25224e−01	4.47624e−01	3.92626e+00	−5.74777e+00	−4.52376e+00

19 Osborne 2

1	2	3	4	5
1.30000e+00	1.30992e+00	1.29520e+00	1.29290e+00	1.39923e+00
6.50000e−01	6.41564e−01	6.56001e−01	6.57061e−01	5.65635e−01
6.50000e−01	6.48854e−01	6.48628e−01	6.52441e−01	6.38536e−01
7.00000e−01	6.92133e−01	7.08213e−01	6.97019e−01	6.21331e−01
6.00000e−01	6.09238e−01	5.93637e−01	6.00265e−01	6.92380e−01
3.00000e+00	2.99009e+00	2.99528e+00	2.99804e+00	2.90093e+00
5.00000e+00	5.00550e+00	4.99291e+00	4.99152e+00	5.05498e+00
7.00000e+00	7.00635e+00	6.99272e+00	6.99480e+00	7.06346e+00
2.00000e+00	2.00737e+00	2.00739e+00	1.99247e+00	2.07374e+00
4.50000e+00	4.49169e+00	4.50159e+00	4.49368e+00	4.41689e+00
5.50000e+00	5.49800e+00	5.50100e+00	5.49480e+00	5.47996e+00
6	**7**	**8**	**9**	**10**
1.25197e+00	1.22899e+00	2.29227e+00	8.19741e−01	5.89910e−01
7.10014e−01	7.20606e−01	−1.93649e−01	1.25014e+00	1.35606e+00
6.36283e−01	6.74411e−01	5.35357e−01	5.12828e−01	8.94110e−01
7.82130e−01	6.70191e−01	−8.66945e−02	1.52130e+00	4.01905e−01
5.36369e−01	6.02650e−01	1.52380e+00	−3.63059e−02	6.26499e−01
2.95276e+00	2.98036e+00	2.00927e+00	2.52761e+00	2.80362e+00
4.92911e+00	4.91519e+00	5.54982e+00	4.29108e+00	4.15193e+00
6.92721e+00	6.94798e+00	7.63461e+00	6.27214e+00	6.47983e+00
2.07386e+00	1.92466e+00	2.73739e+00	2.73858e+00	1.24664e+00
4.51594e+00	4.43678e+00	3.66887e+00	4.65941e+00	3.86782e+00
5.50997e+00	5.44799e+00	5.29957e+00	5.59972e+00	4.97991e+00

20 Watson

1	2	3	4	5
0.00000e+00	5.02534e−01	−4.84984e−01	2.32089e−01	5.02534e+00
0.00000e+00	−4.89810e−01	6.81435e−01	−5.34223e−02	−4.89810e+00
0.00000e+00	1.19141e−02	−4.91436e−01	−2.96681e−01	1.19141e−01
0.00000e+00	3.98153e−01	6.28570e−01	6.61657e−01	3.98153e+00
0.00000e+00	7.81807e−01	−5.12950e−01	1.70528e−01	7.81807e+00
0.00000e+00	9.18583e−01	8.58527e−01	9.94472e−02	9.18583e+00
0.00000e+00	9.44311e−02	−3.00033e−01	8.34387e−01	9.44311e−01
0.00000e+00	−7.22751e−01	−6.06810e−01	−4.28322e−01	−7.22751e+00
0.00000e+00	−7.01412e−01	−4.97832e−01	5.14401e−01	−7.01412e+00

6	7	8	9	10
−4.84984e+00	2.32089e+00	5.02534e+01	−4.84984e+01	2.32089e+01
6.81435e+00	−5.34223e−01	−4.89810e+01	6.81435e+01	−5.34223e+00
−4.91436e+00	−2.96681e+00	1.19141e+00	−4.91436e+01	−2.96681e+01
6.28570e+00	6.61657e+00	3.98153e+01	6.28570e+01	6.61657e+01
−5.12950e+00	1.70528e+00	7.81807e+01	−5.12950e+01	1.70528e+01
8.58527e+00	9.94472e−01	9.18583e+01	8.58527e+01	9.94472e+00
−3.00033e+00	8.34387e+00	9.44311e+00	−3.00033e+01	8.34387e+01
−6.06810e+00	−4.28322e+00	−7.22751e+01	−6.06810e+01	−4.28322e+01
−4.97832e+00	5.14401e+00	−7.01412e+01	−4.97832e+01	5.14401e+01

21 Extended Rosenbrock

1	2	3	4	5
−1.20000e+00	−6.92542e−01	−1.26122e+00	−8.91842e−01	3.87458e+00
1.00000e+00	7.60892e−01	2.38041e−02	1.37843e+00	−1.39108e+00
−1.20000e+00	−1.06436e+00	−1.52576e+00	−7.03697e−01	1.56433e−01
1.00000e+00	1.51709e−01	3.24365e−01	9.01083e−01	−7.48291e+00
−1.20000e+00	−2.09210e+00	−6.11431e−01	−2.03236e+00	−1.01210e+01
1.00000e+00	1.06160e+00	6.22430e−01	4.57954e−01	1.61595e+00
−1.20000e+00	−6.41666e−01	−1.14293e+00	−3.73325e−01	4.38335e+00
1.00000e+00	1.86802e+00	3.31298e−01	3.04756e−01	9.68021e+00
−1.20000e+00	−1.94019e+00	−9.96036e−01	−5.48366e−01	−8.60188e+00
1.00000e+00	1.13765e+00	5.25943e−01	1.07669e+00	2.37647e+00

6	7	8	9	10
−1.81219e+00	1.88158e+00	4.95458e+01	−7.32187e+00	2.96158e+01
−8.76196e+00	4.78429e+00	−2.29108e+01	−9.66196e+01	3.88429e+01
−4.45755e+00	3.76303e+00	1.23643e+01	−3.37755e+01	4.84303e+01
−5.75635e+00	1.08320e−02	−8.38291e+01	−6.65635e+01	−8.89168e+00
4.68569e+00	−9.52357e+00	−9.04100e+01	5.76569e+01	−8.44357e+01
−2.77570e+00	−4.42046e+00	7.15951e+00	−3.67570e+01	−5.32046e+01
−6.29337e−01	7.06675e+00	5.46335e+01	4.50663e+00	8.14675e+01
−5.68703e+00	−5.95244e+00	8.78021e+01	−6.58703e+01	−6.85244e+01
8.39639e−01	5.31634e+00	−7.52188e+01	1.91964e+01	6.39634e+01
−3.74057e+00	1.76685e+00	1.47647e+01	−4.64057e+01	8.66849e+00

22 Extended Powell singular

1	2	3	4	5
3.00000e+00	2.83453e+00	2.48338e+00	2.08605e+00	1.34534e+00
−1.00000e+00	−1.90069e+00	−1.19218e+00	−1.66202e+00	−1.00069e+01
0.00000e+00	8.05432e−01	−8.07091e−01	2.98231e−01	8.05432e+00
1.00000e+00	1.88957e+00	2.63947e−01	1.46345e+00	9.89574e+00
3.00000e+00	2.98173e+00	3.88410e+00	3.29549e+00	2.81728e+00
−1.00000e+00	−1.02150e+00	−8.77309e−02	−1.09815e+00	−1.21495e+00
0.00000e+00	−3.24561e−01	1.50417e−01	9.40178e−02	−3.24561e+00
1.00000e+00	1.80011e+00	1.19559e−01	5.92642e−01	9.00108e+00
3.00000e+00	2.73849e+00	2.46956e+00	3.48939e+00	3.84936e−01
−1.00000e+00	−1.77759e+00	−1.29368e+00	−1.62209e+00	−8.77595e+00
0.00000e+00	5.60504e−01	6.42388e−01	3.73551e−01	5.60504e+00
1.00000e+00	7.79478e−01	3.08069e−02	3.67022e−01	−1.20522e+00

6	7	8	9	10
−2.16617e+00	−6.13952e+00	−1.35466e+01	−4.86617e+01	−8.83952e+01
−2.92176e+00	−7.62020e+00	−9.10691e+01	−2.02176e+01	−6.72020e+01
−8.07091e+00	2.98231e+00	8.05432e+01	−8.07091e+01	2.98231e+01
−6.36053e+00	5.63445e+00	8.99574e+01	−7.26053e+01	4.73445e+01
1.18410e+01	5.95492e+00	1.17282e+00	9.14101e+01	3.25492e+01
8.12269e+00	−1.98153e+00	−3.14947e+00	9.02269e+01	−1.08153e+01
1.50417e+00	9.40178e−01	−3.24561e+01	1.50417e+01	9.40178e+00
−7.80441e+00	−3.07358e+00	8.10108e+01	−8.70441e+01	−3.97358e+01
−2.30440e+00	7.89386e+00	−2.31506e+01	−5.00440e+01	5.19386e+01
−3.93683e+00	−7.22090e+00	−7.87595e+01	−3.03683e+01	−6.32090e+01
6.42388e+00	3.73551e+00	5.60504e+01	6.42388e+01	3.73551e+01
−8.69193e+00	−5.32978e+00	−2.10522e+01	−9.59193e+01	−6.22978e+01

23 Penalty I

1	2	3	4	5
1.00000e+00	2.83773e−01	1.91899e+00	1.86799e+00	−6.16227e+00
1.00000e+00	8.43523e−01	1.31148e+00	1.35747e+00	−5.64774e−01
1.00000e+00	1.83147e+00	7.14234e−02	1.51548e+00	9.31471e+00
1.00000e+00	1.58442e+00	1.69826e+00	1.48627e+00	6.84415e+00

6	7	8	9	10
1.01899e+01	9.67987e+00	−7.06227e+01	9.28985e+01	8.77987e+01
4.11481e+00	4.57470e+00	−1.46477e+01	3.21481e+01	3.67470e+01
−8.28577e+00	6.15480e+00	8.41471e+01	−9.18577e+01	5.25480e+01
7.98259e+00	5.86265e+00	5.94415e+01	7.08259e+01	4.96265e+01

24 Penalty II

1	2	3	4	5
5.00000e−01	−2.16227e−01	1.41899e+00	1.36799e+00	−6.66227e+00
5.00000e−01	3.43523e−01	8.11481e−01	8.57470e−01	−1.06477e+00
5.00000e−01	1.33147e+00	−4.28577e−01	1.01548e+00	8.81471e+00
5.00000e−01	1.08442e+00	1.19826e+00	9.86265e−01	6.34415e+00

6	7	8	9	10
9.68985e+00	9.17987e+00	−7.11227e+01	9.23985e+01	8.72987e+01
3.61481e+00	4.07470e+00	−1.51477e+01	3.16481e+01	3.62470e+01
−8.78577e+00	5.65480e+00	8.36471e+01	−9.23577e+01	5.20480e+01
7.48259e+00	5.36265e+00	5.89415e+01	7.03259e+01	4.91265e+01

25 Variably dimensioned

1	2	3	4	5
9.00000e−01	1.40746e+00	8.38781e−01	1.20816e+00	5.97458e+00
9.00000e−01	6.60892e−01	−7.61959e−02	1.27843e+00	−1.49108e+00
9.00000e−01	1.03564e+00	5.74245e−01	1.39630e+00	2.25643e+00
9.00000e−01	5.17086e−02	2.24365e−01	8.01083e−01	−7.58291e+00
9.00000e−01	7.90024e−03	1.48857e+00	6.76428e−02	−8.02100e+00
9.00000e−01	9.61595e−01	5.22430e−01	3.57954e−01	1.51595e+00
9.00000e−01	1.45833e+00	9.57066e−01	1.72668e+00	6.48335e+00
9.00000e−01	1.76802e+00	2.31298e−01	2.04756e−01	9.58021e+00
9.00000e−01	1.59812e−01	1.10396e+00	1.55163e+00	−6.50188e+00
9.00000e−01	1.03765e+00	4.25943e−01	9.76685e−01	2.27647e+00

6	7	8	9	10
2.87813e−01	3.98158e+00	5.16458e+01	−5.22187e+00	3.17158e+01
−8.86196e+00	4.68429e+00	−2.30108e+01	−9.67196e+01	3.87429e+01
−2.35755e+00	5.86303e+00	1.44643e+01	−3.16755e+01	5.05303e+01
−5.85635e+00	−8.91680e−02	−8.39291e+01	−6.66635e+01	−8.99168e+00
6.78569e+00	-7.42357e+00	−8.83100e+01	5.97569e+01	−8.23357e+01
−2.87570e+00	−4.52046e+00	7.05951e+00	−3.68570e+01	−5.33046e+01
1.47066e+00	9.16675e+00	5.67335e+01	6.60663e+00	8.35675e+01
−5.78703e+00	−6.05244e+00	8.77021e+01	−6.59703e+01	−6.86244e+01
2.93964e+00	7.41634e+00	−7.31188e+01	2.12964e+01	6.60634e+01
−3.84057e+00	1.66685e+00	1.46647e+01	−4.65057e+01	8.56849e+00

26 Trigonometric

1	2	3	4	5
1.00000e−01	6.07458e−01	3.87813e−02	4.08158e−01	5.17458e+00
1.00000e−01	−1.39108e−01	−8.76196e−01	4.78429e−01	−2.29108e+00
1.00000e−01	2.35643e−01	−2.25755e−01	5.96303e−01	1.45643e+00
1.00000e−01	−7.48291e−01	−5.75635e−01	1.08320e−03	−8.38291e+00
1.00000e−01	−7.92100e−01	6.88569e−01	−7.32357e−01	−8.82100e+00
1.00000e−01	1.61595e−01	−2.77570e−01	−4.42046e−01	7.15951e−01
1.00000e−01	6.58335e−01	1.57066e−01	9.26675e−01	5.68335e+00
1.00000e−01	9.68021e−01	−5.68703e−01	−5.95244e−01	8.78021e+00
1.00000e−01	−6.40188e−01	3.03964e−01	7.51634e−01	−7.30188e+00
1.00000e−01	2.37647e−01	−3.74057e−01	1.76685e−01	1.47647e+00

6	7	8	9	10
−5.12187e−01	3.18158e+00	5.08458e+01	−6.02187e+00	3.09158e+01
−9.66196e+00	3.88429e+00	−2.38108e+01	−9.75196e+01	3.79429e+01
−3.15755e+00	5.06303e+00	1.36643e+01	−3.24755e+01	4.97303e+01
−6.65635e+00	−8.89168e−01	−8.47291e+01	−6.74635e+01	−9.79168e+00
5.98569e+00	−8.22357e+00	−8.91100e+01	5.89569e+01	−8.31357e+01
−3.67570e+00	−5.32046e+00	6.25951e+00	−3.76570e+01	−5.41046e+01
6.70663e−01	8.36675e+00	5.59335e+01	5.80663e+00	8.27675e+01
−6.58703e+00	−6.85244e+00	8.69021e+01	−6.67703e+01	−6.94244e+01
2.13964e+00	6.61634e+00	−7.39188e+01	2.04964e+01	6.52634e+01
−4.64057e+00	8.66849e−01	1.38647e+01	−4.73057e+01	7.76849e+00

27 Brown almost-linear

1	2	3	4	5
5.00000e−01	5.50746e−01	4.93878e−01	5.30816e−01	1.00746e+00
5.00000e−01	4.76089e−01	4.02380e−01	5.37843e−01	2.60892e−01
5.00000e−01	5.13564e−01	4.67425e−01	5.49630e−01	6.35643e−01
5.00000e−01	4.15171e−01	4.32437e−01	4.90108e−01	−3.48291e−01
5.00000e−01	4.10790e−01	5.58857e−01	4.16764e−01	−3.92100e−01
5.00000e−01	5.06160e−01	4.62243e−01	4.45795e−01	5.61595e−01
5.00000e−01	5.55833e−01	5.05707e−01	5.82668e−01	1.05833e+00
5.00000e−01	5.86802e−01	4.33130e−01	4.30476e−01	1.36802e+00
5.00000e−01	4.25981e−01	5.20396e−01	5.65163e−01	−2.40188e−01
5.00000e−01	5.13765e−01	4.52594e−01	5.07669e−01	6.37647e−01

6	7	8	9	10
4.38781e−01	8.08158e−01	5.57458e+00	−1.12187e−01	3.58158e+00
−4.76196e−01	8.78429e−01	−1.89108e+00	−9.26196e+00	4.28429e+00
1.74245e−01	9.96303e−01	1.85643e+00	−2.75755e+00	5.46303e+00
−1.75635e−01	4.01083e−01	−7.98291e+00	−6.25635e+00	−4.89168e−01
1.08857e+00	−3.32357e−01	−8.42100e+00	6.38569e+00	−7.82357e+00
1.22430e−01	−4.20461e−02	1.11595e+00	−3.27570e+00	−4.92046e+00
5.57066e−01	1.32668e+00	6.08335e+00	1.07066e+00	8.76675e+00
−1.68703e−01	−1.95244e−01	9.18021e+00	−6.18703e+00	−6.45244e+00
7.03964e−01	1.15163e+00	−6.90188e+00	2.53964e+00	7.01634e+00
2.59426e−02	5.76685e−01	1.87647e+00	−4.24057e+00	1.26685e+00

28 Discrete boundary value

29 Discrete Integral equation

1	2	3	4	5
−8.26446e−02	4.24814e−01	−1.43863e−01	2.25514e−01	4.99194e+00
−1.48760e−01	−3.87869e−01	−1.12496e+00	2.29669e−01	−2.53984e+00
−1.98347e−01	−6.27038e−02	−5.24102e−01	2.97956e−01	1.15809e+00
−2.31405e−01	−1.07970e+00	−9.07040e−01	−3.30322e−01	−8.71432e+00
−2.47934e−01	−1.14003e+00	3.40635e−01	−1.08029e+00	−9.16893e+00
−2.47934e−01	−1.86339e−01	−6.25504e−01	−7.89980e−01	3.68017e−01
−2.31405e−01	3.26930e−01	−1.74339e−01	5.95270e−01	5.35194e+00
−1.98347e−01	6.69674e−01	−8.67050e−01	−8.93591e−01	8.48187e+00
−1.48760e−01	−8.88948e−01	5.52036e−02	5.02874e−01	−7.55064e+00
−8.26446e−02	5.50027e−02	−5.56702e−01	−5.95976e−03	1.29383e+00

6	7	8	9	10
−6.94832e−01	2.99894e+00	5.06632e+01	−6.20452e+00	3.07332e+01
−9.91072e+00	3.63553e+00	−2.40596e+01	−9.77684e+01	3.76941e+01
−3.45589e+00	4.76469e+00	1.33660e+01	−3.27738e+01	4.94320e+01
−6.98776e+00	−1.22057e+00	−8.50606e+01	−6.77949e+01	−1.01231e+01
5.63776e+00	−8.57151e+00	−8.94579e+01	5.86090e+01	−8.34837e+01
−4.02363e+00	−5.66840e+00	5.91158e+00	−3.80049e+01	−5.44525e+01
3.39258e−01	8.03534e+00	5.56020e+01	5.47522e+00	8.24361e+01
−6.88537e+00	−7.15079e+00	8.66038e+01	−6.70686e+01	−6.97227e+01
1.89088e+00	6.36758e+00	−7.41675e+01	2.02476e+01	6.50146e+01
−4.82322e+00	6.84204e−01	1.36821e+01	−4.74884e+01	7.58584e+00

30 Broyden tridiagonal

1	2	3	4	5
−1.00000e+00	−9.94925e−01	−1.00061e+00	−9.96918e−01	−9.49254e−01
−1.00000e+00	−1.00239e+00	−1.00976e+00	−9.96216e−01	−1.02391e+00
−1.00000e+00	−9.98644e−01	−1.00326e+00	−9.95037e−01	−9.86436e−01
−1.00000e+00	−1.00848e+00	−1.00676e+00	−1.00099e+00	−1.08483e+00
−1.00000e+00	−1.00892e+00	−9.94114e−01	−1.00832e+00	−1.08921e+00
−1.00000e+00	−9.99384e−01	−1.00378e+00	−1.00542e+00	−9.93841e−01
−1.00000e+00	−9.94417e−01	−9.99429e−01	−9.91733e−01	−9.44167e−01
−1.00000e+00	−9.91320e−01	−1.00669e+00	−1.00695e+00	−9.13198e−01
−1.00000e+00	−1.00740e+00	−9.97960e−01	−9.93484e−01	−1.07402e+00
−1.00000e+00	−9.98624e−01	−1.00474e+00	−9.99233e−01	−9.86235e−01

6	7	8	9	10
−1.00612e+00	−9.69184e−01	−4.92542e−01	−1.06122e+00	−6.91842e−01
−1.09762e+00	−9.62157e−01	−1.23911e+00	−1.97620e+00	−6.21571e−01
−1.03258e+00	−9.50370e−01	−8.64357e−01	−1.32576e+00	−5.03697e−01
−1.06756e+00	−1.00989e+00	−1.84829e+00	−1.67564e+00	−1.09892e+00
−9.41143e−01	−1.08324e+00	−1.89210e+00	−4.11431e−01	−1.83236e+00
−1.03776e+00	−1.05421e+00	−9.38405e−01	−1.37757e+00	−1.54205e+00
−9.94293e−01	−9.17333e−01	−4.41666e−01	−9.42934e−01	−1.73325e−01
−1.06687e+00	−1.06952e+00	−1.31979e−01	−1.66870e+00	−1.69524e+00
−9.79604e−01	−9.34837e−01	−1.74019e+00	−7.96036e−01	−3.48366e−01
−1.04741e+00	−9.92332e−01	−8.62353e−01	−1.47406e+00	−9.23315e−01

31 Broyden banded

1	2	3	4	5
−1.00000e+00	−4.92542e−01	−1.06122e+00	−6.91842e−01	4.07458e+00
−1.00000e+00	−1.23911e+00	−1.97620e+00	−6.21571e−01	−3.39108e+00
−1.00000e+00	−8.64357e−01	−1.32576e+00	−5.03697e−01	3.56433e−01
−1.00000e+00	−1.84829e+00	−1.67564e+00	−1.09892e+00	−9.48291e+00
−1.00000e+00	−1.89210e+00	−4.11431e−01	−1.83236e+00	−9.92100e+00
−1.00000e+00	−9.38405e−01	−1.37757e+00	−1.54205e+00	−3.84049e−01
−1.00000e+00	−4.41666e−01	−9.42934e−01	−1.73325e−01	4.58335e+00
−1.00000e+00	−1.31979e−01	−1.66870e+00	−1.69524e+00	7.68021e+00
−1.00000e+00	−1.74019e+00	−7.96036e−01	−3.48366e−01	−8.40188e+00
−1.00000e+00	−8.62353e−01	−1.47406e+00	−9.23315e−01	3.76473e−01

6	7	8	9	10
−1.61219e+00	2.08158e+00	4.97458e+01	−7.12187e+00	2.98158e+01
−1.07620e+01	2.78429e+00	−2.49108e+01	−9.86196e+01	3.68429e+01
−4.25755e+00	3.96303e+00	1.25643e+01	−3.35755e+01	4.86303e+01
−7.75635e+00	−1.98917e+00	−8.58291e+01	−6.85635e+01	−1.08917e+01
4.88569e+00	−9.32357e+00	−9.02100e+01	5.78569e+01	−8.42357e+01
−4.77570e+00	−6.42046e+00	5.15951e+00	−3.87570e+01	−5.52046e+01
−4.29337e−01	7.26675e+00	5.48335e+01	4.70663e+00	8.16675e+01
−7.68703e+00	−7.95244e+00	8.58021e+01	−6.78703e+01	−7.05244e+01
1.03964e+00	5.51634e+00	−7.50188e+01	1.93964e+01	6.41634e+01
−5.74057e+00	−2.33151e−01	1.27647e+01	−4.84057e+01	6.66849e+00

32 Linear - full rank

33 Linear - rank 1

34 Linear - rank 1 with zero columns and rows

1	2	3	4	5
1.00000e+00	1.50746e+00	9.38781e−01	1.30816e+00	6.07458e+00
1.00000e+00	7.60892e−01	2.38041e−02	1.37843e+00	−1.39108e+00
1.00000e+00	1.13564e+00	6.74245e−01	1.49630e+00	2.35643e+00
1.00000e+00	1.51709e−01	3.24365e−01	9.01083e−01	−7.48291e+00
1.00000e+00	1.07900e−01	1.58857e+00	1.67643e−01	−7.92100e+00
1.00000e+00	1.06160e+00	6.22430e−01	4.57954e−01	1.61595e+00
1.00000e+00	1.55833e+00	1.05707e+00	1.82668e+00	6.58335e+00
1.00000e+00	1.86802e+00	3.31298e−01	3.04756e−01	9.68021e+00
1.00000e+00	2.59812e−01	1.20396e+00	1.65163e+00	−6.40188e+00
1.00000e+00	1.13765e+00	5.25943e−01	1.07669e+00	2.37647e+00

6	7	8	9	10
3.87813e−01	4.08158e+00	5.17458e+01	−5.12187e+00	3.18158e+01
−8.76196e+00	4.78429e+00	−2.29108e+01	−9.66196e+01	3.88429e+01
−2.25755e+00	5.96303e+00	1.45643e+01	−3.15755e+01	5.06303e+01
−5.75635e+00	1.08320e−02	−8.38291e+01	−6.65635e+01	−8.89168e+00
6.88569e+00	−7.32357e+00	−8.82100e+01	5.98569e+01	−8.22357e+01
−2.77570e+00	−4.42046e+00	7.15951e+00	−3.67570e+01	−5.32046e+01
1.57066e+00	9.26675e+00	5.68335e+01	6.70663e+00	8.36675e+01
−5.68703e+00	−5.95244e+00	8.78021e+01	−6.58703e+01	−6.85244e+01
3.03964e+00	7.51634e+00	−7.30188e+01	2.13964e+01	6.61634e+01
−3.74057e+00	1.76685e+00	1.47647e+01	−4.64057e+01	8.66849e+00

35 Chebyquad

1	2	3	4	5
1.00000e−01	1.50253e−01	5.15017e−02	1.23209e−01	6.02534e−01
1.00000e−01	5.10190e−02	1.68144e−01	9.46578e−02	−3.89810e−01
1.00000e−01	1.01191e−01	5.08564e−02	7.03319e−02	1.11914e−01
1.00000e−01	1.39815e−01	1.62857e−01	1.66166e−01	4.98153e−01
1.00000e−01	1.78181e−01	4.87050e−02	1.17053e−01	8.81807e−01
1.00000e−01	1.91858e−01	1.85853e−01	1.09945e−01	1.01858e+00
1.00000e−01	1.09443e−01	6.99968e−02	1.83439e−01	1.94431e−01
1.00000e−01	2.77249e−02	3.93191e−02	5.71678e−02	−6.22751e−01
1.00000e−01	2.98588e−02	5.02168e−02	1.51440e−01	−6.01412e−01

6	7	8	9	10
−3.84984e−01	3.32089e−01	5.12534e+00	−4.74984e+00	2.42089e+00
7.81435e−01	4.65777e−02	−4.79810e+00	6.91435e+00	−4.34223e−01
−3.91436e−01	−1.96681e−01	2.19141e−01	−4.81436e+00	−2.86681e+00
7.28570e−01	7.61657e−01	4.08153e+00	6.38570e+00	6.71657e+00
−4.12950e−01	2.70528e−01	7.91807e+00	−5.02950e+00	1.80528e+00
9.58527e−01	1.99447e−01	9.28583e+00	8.68527e+00	1.09447e+00
−2.00033e−01	9.34387e−01	1.04431e+00	−2.90033e+00	8.44387e+00
−5.06810e−01	−3.28322e−01	−7.12751e+00	−5.96810e+00	−4.18322e+00
−3.97832e−01	6.14401e−01	−6.91412e+00	−4.87832e+00	5.24401e+00

Appendix C MATLAB Source Code

This appendix contains the MATLAB implementations C.1, C.2, C.3, C.4 and C.5 of the algorithms discussed in the thesis. Their benchmarking was performed with test functions from [33]. We provide the MATLAB code for all these functions in C.7. Program C.6 runs on its own but was initially written to understand and to test the stepwise regression procedure in C.5. The M-code LaTeX Package [24] was used for the representation of the MATLAB files.

C.1 Gauss-Newton Algorithm

```
function [ x, phi, fcount, tcount, icount ] = gn( f, v, n, T, ...
    fevals, tol )
%% GAUSS-NEWTON method
%   for nonlinear least squares approximation
%% AUTHOR: Stefan Scheer
%% REFERENCE:
% K.M. Brown and J.E. Dennis. "Derivative Free Analogues of the
% Levenberg-Marquardt and Gauss Algorithms for Nonlinear Least
% Squares Approximation". In: Numer. Math. 18 (1971), pp. 289-297.
%
%% INPUT:
% f - function in the NLLS problem "minimize phi(x) = f(x)'f(x)",
%     [F, J] = f(v) where F is the function evaluation and
%                          J is the Jacobian/gradient of F
% v - vector containing the initial approximation x0
%                       and function parameters par (optional),
%     i.e., v = [x0; par] for column vectors x0 and par7
% n - positive integer, dimension of the domain of the function f
% T - positive scalar for the termination criteria 'phi < T'
%     (optional, default = realmin)
% fevals - nonnegative integer, maximum number of function
%          evaluations (optional, default = 1000)
% tol - positive scalar, step tolerance (optional, default = 1e-5)
%
%% OUTPUT:
% x - n-dimensional vector, approximate solution of the NLLS problem
% phi - minimized squared 2-norm of f (often a residual vector)
% fcount - number of function evaluations
% tcount - elapsed time to minimize NLLS problem
% icount - number of iterations
%
%% EXAMPLE1:
% x0 = [-1.2;1];
% [x, phi, fcount] = gn( @rosenbrock2D, x0, 2 )
%
%% EXAMPLE2:
% v = [[0;20;20]; (0.1:0.1:1)'];
% [x, phi, fcount, tcount, icount] = gn( @box3D, v, 3, 1e-5 )

tic; % start timer


```

```
42 %% Ensuring correct input
43 % ensure that the number of input arguments is correct
44 % and that the arguments 4-6 are scalars/integers
45 if nargin == 3
46     T = realmin;
47     fevals = 1e3;
48     tol = 1e-5;
49 elseif nargin == 4
50     fevals = 1e3;
51     tol = 1e-5;
52     if length(T) ~= 1 || T <= 0
53         error('T must be a positive scalar.')
54     end
55 elseif nargin == 5
56     tol = 1e-5;
57     if length(T) ~= 1 || T <= 0
58         error('T must be a positive scalar.')
59     elseif length(fevals) ~= 1 || mod(fevals,1) ~= 0 || fevals < 0
60         error('maxit must be a nonnegative integer.')
61     end
62 elseif nargin == 6
63     if length(T) ~= 1 || T <= 0
64         error('T must be a positive scalar.')
65     elseif length(fevals) ~= 1 || mod(fevals,1) ~= 0 || fevals < 0
66         error('maxit must be a nonnegative integer.')
67     elseif length(tol) ~= 1 || tol <= 0
68     error('tol must be a positive scalar.')
69     end
70 else
71     error('Not enough input arguments.');
72 end
73 % ensure that the input arguments 1-3 are also correct
74 if ~isa(f,'function_handle')
75     error('Enter function handle.')
76 elseif isvector(v) ~= 1
77     error('v must be a vector.')
78 elseif length(n) ~= 1 || mod(n,1) ~= 0 || n <= 0
79     error('n must be a positive integer.')
80 end
81 % ensure that the program works with a column vector
82 [lv, u] = size(v);
83 if u ~= 1
84     v = v';
85     lv = u; % length of v
86 end
87
88 %% Initialization
89 x = v(1:n);
90 if lv-n == 0 % no additional function parameters given
91     par = [];
92 else
93     par = v(n+1:end);
94 end
95 fcount = 0;
96 [F, J] = f([x; par]); fcount = fcount + 1;
97 phi = norm(F)^2;      % squared 2-norm for inital approximation
98 icount = 0;           % iteration counter
99
```

```
100 %% Main procedure
101 while fcount < fevals && phi >= T
102     icount = icount+1;
103     p = J'*J \ -J'*F; % solve LLS problem norm(J*p+F) = min for p
104     if norm(p) <= tol
105         break             % stop iteration if stepsize is too small
106     end
107     x = x + p;            % new approximation
108     [F, J] = f([x; par]); fcount = fcount + 1 + n;
109     phi = norm(F)^2;      % squared 2-norm of F at the minimum
110 end
111 tcount = toc;             % end timer
112 end
```

C.2 Finite Difference Gauss-Newton Algorithm

```
1  function [ x, phi, fcount, tcount, icount ] = fdgn( f, v, n, T, ...
       fevals, tol )
2  %% FINITE DIFFERENCE GAUSS-NEWTON method
3  %   for nonlinear least squares approximation
4  %% AUTHOR: Stefan Scheer
5  %% REFERENCE:
6  % K.M. Brown and J.E. Dennis. "Derivative Free Analogues of the
7  % Levenberg-Marquardt and Gauss Algorithms for Nonlinear Least
8  % Squares Approximation". In: Numer. Math. 18 (1971), pp. 289-297.
9  %
10 %% INPUT:1
11 % f - function in the NLLS problem "minimize phi(x) = f(x)'f(x)"
12 % v - vector containing the initial approximation x0
13 %                       and function parameters par (optional),
14 %     i.e., v = [x0; par] for column vectors x0 and par
15 % n - positive integer, dimension of the domain of the function f
16 % T - positive scalar for the termination criteria 'phi < T'
17 %     (optional, default = realmin)
18 % fevals - nonnegative integer, maximum number of function
19 %          evaluations (optional, default = 1000)
20 % tol - positive scalar, step tolerance (optional, default = 1e-5)
21 %
22 %% OUTPUT:
23 % x - n-dimensional vector, approximate solution of the NLLS problem
24 % phi - minimized squared 2-norm of f (often a residual vector)
25 % fcount - number of function evaluations
26 % tcount - elapsed time to minimize NLLS problem
27 % icount - number of iterations
28 %
29 %% EXAMPLE1:
30 % x0 = [-1.2;1];
31 % [x, phi, fcount] = fdgn( @rosenbrock2D, x0, 2 )
32 %
33 %% EXAMPLE2:
34 % v = [[0;20;20]; (0.1:0.1:1)'];
35 % [x, phi, fcount, tcount, icount] = fdgn( @box2, v, 3, 1e-5 )
36
37 tic; % start timer
```

```
38 %% Ensuring correct input
39 % ensure that the number of input arguments is correct and
40 % that the arguments 4-6 are scalars/integers
41 if nargin == 3
42     T = realmin;
43     fevals = 1e3;
44     tol = 1e-5;
45 elseif nargin == 4
46     fevals = 1e3;
47     tol = 1e-5;
48     if length(T) ~= 1 || T <= 0
49         error('T must be a positive scalar.')
50     end
51 elseif nargin == 5
52     tol = 1e-5;
53     if length(T) ~= 1 || T <= 0
54         error('T must be a positive scalar.')
55     elseif length(fevals) ~= 1 || mod(fevals,1) ~= 0 || fevals < 0
56         error('maxit must be a nonnegative integer.')
57     end
58 elseif nargin == 6
59     if length(T) ~= 1 || T <= 0
60         error('T must be a positive scalar.')
61     elseif length(fevals) ~= 1 || mod(fevals,1) ~= 0 || fevals < 0
62         error('maxit must be a nonnegative integer.')
63     elseif length(tol) ~= 1 || tol <= 0
64     error('tol must be a positive scalar.')
65     end
66 else
67     error('Not enough input arguments.');
68 end
69 % ensure that the input arguments 1-3 are also correct
70 if ~isa(f,'function_handle')
71     error('Enter function handle.')
72 elseif isvector(v) ~= 1
73     error('v must be a vector.')
74 elseif length(n) ~= 1 || mod(n,1) ~= 0 || n <= 0
75     error('n must be a positive integer.')
76 end
77 % ensure that the program works with a column vector
78 [lv, u] = size(v);
79 if u ~= 1
80     v = v';
81     lv = u; % length of v
82 end
83
84 %% Initialization
85 x = v(1:n);
86 if lv-n == 0 % no additional function parameters given
87     par = [];
88 else
89     par = v(n+1:end);
90 end
91 fcount = 0;
92 F = f([x; par]); fcount = fcount + 1;
93 Fnorm = norm(F);
94 phi = norm(F)^2;     % squared 2-norm for inital approximation
95 n = length(x);       % determine size
```

```
96  m = length(F);       % of the m x n matrix delta_F(x,h)
97  J = zeros(m,n);      % h^{-1}*delta_F(x,h) -> J(x) as h -> 0
98  h = zeros(n,1);      % h << 1 componentwise in the iteration
99  delta = zeros(n,1); % shall guarantee non-uniform h
100 I = eye(n);          % needed for unit vector construction
101 icount = 0;          % iteration counter
102
103 %% Main procedure
104 while fcount < fevals && phi >= T
105     icount = icount + 1;
106     % finite difference approximation of J
107     for j = 1:n
108         % convergence rule for choice of h
109         if abs(x(j)) < 10*sqrt(eps)
110            delta(j) = 1e-2*sqrt(eps);
111         else
112            delta(j) = 1e-3*abs(x(j));
113         end
114         h(j) = min(Fnorm, delta(j));
115         % construction of unit vector in each iteration
116         u = I(:,j);               % j-th unit vector
117         Fh = f([x + h(j)*u; par]); fcount = fcount + 1;
118         J(:,j) = (Fh - F)/h(j);   % difference quotient
119     end
120     p = J'*J \ -J'*F; % solve LLS problem norm(J*p+F) = min for p
121     if norm(p) <= tol
122         break         % stop iteration if stepsize is too small
123     end
124     x = x + p;        % new approximation
125     F = f([x; par]); fcount = fcount + 1;
126     Fnorm = norm(F);
127     phi = Fnorm^2;    % squared 2-norm of F at the minimum
128 end
129 tcount = toc;         % end timer
130 end
```

C.3 Levenberg-Marquardt Algorithm

```
function [ x, phi, fcount, tcount, icount ] = lm( f, v, n, T, ...
    fevals, tol )
%% LEVENBERG-MARQUARDT method
%   for nonlinear least squares approximation
%% AUTHOR: Stefan Scheer
%% REFERENCE:
% K.M. Brown and J.E. Dennis. "Derivative Free Analogues of the
% Levenberg-Marquardt and Gauss Algorithms for Nonlinear Least
% Squares Approximation". In: Numer. Math. 18 (1971), pp. 289-297.
%
%% INPUT:
% f - function in the NLLS problem "minimize phi(x) = f(x)'f(x)",
%     [F, J] = f(v) where F is the function evaluation and
%                           J is the Jacobian/gradient of F
% v - vector containing the initial approximation x0
%                         and function parameters par (optional),
%     i.e., v = [x0; par] for column vectors x0 and par
% n - positive integer, dimension of the domain of the function f
% T - positive scalar for the termination criteria 'phi < T'
%     (optional, default = realmin)
% fevals - nonnegative integer, maximum number of function
%          evaluations (optional, default = 1000)
% tol - positive scalar, step tolerance (optional, default = 1e-5)
%
%% OUTPUT:
% x - n-dimensional vector, approximate solution of the NLLS problem
% phi - minimized squared 2-norm of f (often a residual vector)
% fcount - number of function evaluations
% tcount - elapsed time to minimize NLLS problem
% icount - number of iterations
%
%% EXAMPLE1:
% x0 = [-1.2;1];
% [x, phi, fcount] = lm( @rosenbrock2D, x0, 2 )
%
%% EXAMPLE2:
% v = [[0;20;20]; (0.1:0.1:1)'];
% [x, phi, fcount, tcount, icount] = lm( @box2, v, 3, 1e-5 )

tic; % start timer

%% Ensuring correct input
% ensure that the number of input arguments is correct
% and that the arguments 4-6 are scalars/integers
if nargin == 3
    T = realmin;
    fevals = 1e3;
    tol = 1e-5;
elseif nargin == 4
    fevals = 1e3;
    tol = 1e-5;
    if length(T) ~= 1 || T <= 0
        error('T must be a positive scalar.')
    end
elseif nargin == 5
```

```
55      tol = 1e-5;
56      if length(T) ~= 1 || T <= 0
57          error('T must be a positive scalar.')
58      elseif length(fevals) ~= 1 || mod(fevals,1) ~= 0 || fevals < 0
59          error('maxit must be a nonnegative integer.')
60      end
61  elseif nargin == 6
62      if length(T) ~= 1 || T <= 0
63          error('T must be a positive scalar.')
64      elseif length(fevals) ~= 1 || mod(fevals,1) ~= 0 || fevals < 0
65          error('maxit must be a nonnegative integer.')
66      elseif length(tol) ~= 1 || tol <= 0
67      error('tol must be a positive scalar.')
68      end
69  else
70      error('Not enough input arguments.');
71  end
72  % ensure that the input arguments 1-3 are also correct
73  if ~isa(f,'function_handle')
74      error('Enter function handle.')
75  elseif isvector(v) ~= 1
76      error('v must be a vector.')
77  elseif length(n) ~= 1 || mod(n,1) ~= 0 || n <= 0
78      error('n must be a positive integer.')
79  end
80  % ensure that the program works with a column vector
81  [lv, u] = size(v);
82  if u ~= 1
83      v = v';
84      lv = u; % length of v
85  end
86
87  %% Initialization
88  x = v(1:n);
89  if lv-n == 0 % no additional function parameters given
90      par = [];
91  else
92      par = v(n+1:end);
93  end
94  fcount = 0;
95  [F, J] = f([x; par]); fcount = fcount + 1;
96  phi = norm(F)^2;      % squared 2-norm for inital approximation
97  I = eye(n);           % needed for formula
98  icount = 0;           % iteration counter
99
100 %% Main procedure
101 while fcount < fevals && phi >= T
102     icount = icount + 1;
103     % compute sup norm of F needed for convergence rule
104     Infnorm = norm(F, Inf);
105     % convergence rule for choice of mu
106     if Infnorm >= 10      % mu large -> method of steepest descent
107         c = 10;
108     elseif Infnorm <= 1   % mu = 0 -> Gauss-Newton method
109         c = 1e-3;
110     else
111         c = 1e-1;
112     end
```

```
113     mu = c*Infnorm;
114     p = (mu*I + J'*J) \ -J'*F; % solve the "damped" normal equations
115     if norm(p) <= tol
116         break           % stop iteration if stepsize is too small
117     end
118     x = x + p;          % new approximation
119     [F, J] = f([x; par]); fcount = fcount + 1 + n;
120     phi = norm(F)^2;    % squared 2-norm of F at the minimum
121 end
122 tcount = toc;           % end timer
123 end
```

C.4 Finite Difference Levenberg-Marquardt Algorithm

```
1  function [ x, phi, fcount, tcount, icount ] = fdlm( f, v, n, T, ...
       fevals, tol )
2  %% FINITE DIFFERENCE LEVENBERG-MARQUARDT ALGORITHM
3  %   for nonlinear least squares approximation
4  %% AUTHOR: Stefan Scheer
5  %% REFERENCE:
6  % K.M. Brown and J.E. Dennis. "Derivative Free Analogues of the
7  % Levenberg-Marquardt and Gauss Algorithms for Nonlinear Least
8  % Squares Approximation". In: Numer. Math. 18 (1971), pp. 289-297.
9  %
10 %% INPUT:
11 % f - function in the NLLS problem "minimize phi(x) = f(x)'f(x)"
12 % v - vector containing the initial approximation x0,
13 %                       and function parameters par (optional),
14 %     i.e., v = [x0; par] for column vectors x0 and par
15 % n - positive integer, dimension of the domain of the function f
16 % T -  positive scalar for the termination criteria 'phi < T'
17 %      (optional, default = realmin)
18 % fevals - nonnegative integer, maximum number of function
19 %          evaluations (optional, default = 1000)
20 % tol - positive scalar, step tolerance (optional, default = 1e-5)
21 %
22 %% OUTPUT:
23 % x - n-dimensional vector, approximate solution of the NLLS problem
24 % phi - minimized squared 2-norm of f (often a residual vector)
25 % fcount - number of function evaluations
26 % tcount - elapsed time to minimize NLLS problem
27 % icount - number of iterations
28 %
29 %% EXAMPLE1:
30 % x0 = [-1.2;1];
31 % [x, phi, fcount] = fdlm( @rosenbrock2D, x0, 2 )
32 %
33 %% EXAMPLE2:
34 % v = [[0;20;20]; (0.1:0.1:1)'];
35 % [x, phi, fcount, tcount, icount] = fdlm( @box2, v, 3, 1e-5 )
36
37 tic; % start timer
38
39
```

```
%% Ensuring correct input
% ensure that the number of input arguments is correct
% and that the arguments 4-6 are scalars/integers
if nargin == 3
    T = realmin;
    fevals = 1e3;
    tol = 1e-5;
elseif nargin == 4
    fevals = 1e3;
    tol = 1e-5;
    if length(T) ~= 1 || T <= 0
        error('T must be a positive scalar.')
    end
elseif nargin == 5
    tol = 1e-5;
    if length(T) ~= 1 || T <= 0
        error('T must be a positive scalar.')
    elseif length(fevals) ~= 1 || mod(fevals,1) ~= 0 || fevals < 0
        error('maxit must be a nonnegative integer.')
    end
elseif nargin == 6
    if length(T) ~= 1 || T <= 0
        error('T must be a positive scalar.')
    elseif length(fevals) ~= 1 || mod(fevals,1) ~= 0 || fevals < 0
        error('maxit must be a nonnegative integer.')
    elseif length(tol) ~= 1 || tol <= 0
    error('tol must be a positive scalar.')
    end
else
    error('Not enough input arguments.');
end
% ensure that the input arguments 1-3 are also correct
if ~isa(f,'function_handle')
    error('Enter function handle.')
elseif isvector(v) ~= 1
    error('v must be a vector.')
elseif length(n) ~= 1 || mod(n,1) ~= 0 || n <= 0
    error('n must be a positive integer.')
end
% ensure that the program works with a column vector
[lv, u] = size(v);
if u ~= 1
    v = v';
    lv = u; % length of v
end

%% Initialization
x = v(1:n);
if lv-n == 0 % no additional function parameters given
    par = [];
else
    par = v(n+1:end);
end
fcount = 0;
F = f([x; par]); fcount = fcount + 1;
Fnorm = norm(F);
phi = Fnorm^2;      % squared 2-norm for inital approximation
n = length(x);      % determine size
```

```
98  m = length(F);       % of the m x n matrix delta_F(x,h)
99  J = zeros(m,n);      % h^{-1}*delta_F(x,h) -> J(x) as h -> 0
100 h = zeros(n,1);      % h << 1 componentwise in the iteration
101 delta = zeros(n,1); % shall guarantee non-uniform h
102 I = eye(n);          % needed for formula + unit vector construction
103 icount = 0;          % iteration counter
104
105 %% Main procedure
106 while fcount < fevals && phi >= T
107     icount = icount + 1;
108     % compute sup norm of F needed for convergence rules
109     Infnorm = norm(F, Inf);
110     % convergence rule for choice of mu
111     if Infnorm >= 10    % mu large -> method of steepest descent
112         c = 10;
113     elseif Infnorm <= 1 % mu = 0 -> Gauss-Newton method
114         c = 1e-3;
115     else
116         c = 1e-1;
117     end
118     mu = c*Infnorm;
119     % finite difference approximation of J
120     for j = 1:n
121         % convergence rule for choice of h
122         if abs(x(j)) < 10*sqrt(eps)
123             delta(j) = 1e-2*sqrt(eps);
124         else
125             delta(j) = 1e-3*abs(x(j));
126         end
127         h(j) = min(Fnorm, delta(j));
128         % construction of unit vector in each iteration
129         u = I(:,j);               % j-th unit vector
130         Fh = f([x + h(j)*u; par]); fcount = fcount + 1;
131         J(:,j) = (Fh - F)/h(j);   % difference quotient
132     end
133     p = (mu*I + J'*J) \ -J'*F; % solve the "damped" normal equations
134     if norm(p) <= tol
135        break            % stop iteration if stepsize is too small
136     end
137     x = x + p;           % new approximation
138     F = f([x; par]); fcount = fcount + 1;
139     Fnorm = norm(F);
140     phi = Fnorm^2;       % squared 2-norm of F at the minimum
141 end
142 tcount = toc;           % end timer
143 end
```

C.5 DUD Algorithm

```
function [ theta_new, Qnew, fcount, tcount, icount ] = dud( f, ...
    v, dim, m, T, fevals, h )
%% DUD, a derivative-free algorithm
%   for nonlinear least squares data fitting problems;
%   DUD approximates the p-dimensional manifold spanned by the
%   values of f(theta) by the secant plane through p+1 previous
%   values of f(theta).
%% AUTHOR: Stefan Scheer
%% REFERENCE:
% M.L. Ralston and R.I. Jennrich. "Dud, A Derivative-Free Algorithm
% for Nonlinear Least Squares". In: Technometrics 20 (1978),
% pp. 7-14.
%
% R.I. Jennrich and P.F. Sampson. "Application of Stepwise
% Regression to Non-Linear Estimation". In: Technometrics 10.1
% (1968), pp. 63-72.
%
%% INPUT:
% f - function which takes as input the parameter vector theta
%     and the explanatory vector x
% v - vector which contains the parameter vector theta
%                       and the explanatory vector x
%                       and the observed data vector y,
%     i.e., v = [theta;x;y] for column vectors theta, x and y
%     --> for standard NLLS problem enter v = theta only <--
% dim - two dimensional vector which contains the dimension of the
%       domain and the dimension of the codomain of the function f,
%       i.e., dim = [p;n] for positive integers p and n
%       for data fitting: p is the dimension of the parameter space
%                         n is the sample size
% m - nonnegative integer, threshold for partial stepping procedure
%     (optional, default = 0)
%     handle carefully -> up to m line searches are performed per
%     estimate and each line search requires a function evaluation
% T - positive scalar for the termination criteria 'RSS < T'
%     (optional, default = realmin)
% fevals - nonnegative integer, maximum number of function
%          evaluations (optional, default = 1000)
% h - scalar or p-dimensional vector (optional, default = theta*0.1)
%     h must not be (or contain) zero(s)
%     needed for the generation of p additional parameters theta_i,
%     where the i-th component of theta is displaced by h (or h_i)
%
%% OUTPUT:
% theta_new - p-dimensional vector
%             approximate solution of NLLS (data fitting) problem
% Qnew - minimized residual sum of squares
% fcount - number of function evaluations
% tcount - elapsed time to minimize NLLS (data fitting) problem
% icount - number of iterations
%
%% EXAMPLE1:
% v = [-1.2;1];
% [theta_new, Qnew, fcount] = dud(@rosenbrock2D, v, [2;2], 0, 10)
%
```

```
55 %% EXAMPLE2:
56 % v = [[0;10;20]; (0.1:0.1:1)'; zeros(10,1)];
57 % [theta_new, Qnew, fcount, tcount, icount] = dud(@box3D, v, ...
       [3;10], 5, 1e-5)
58
59 tic; % start timer
60
61 %% Ensuring correct input
62 % ensure that the number of input arguments is correct and that the
63 % arguments 4-6 are scalars/integers
64 if nargin == 3
65     m = 0;
66     T = realmin;
67     fevals = 1e3;
68 elseif nargin == 4
69     T = realmin;
70     fevals = 1e3;
71     if length(m) ~= 1 || mod(m,1) ~= 0 || m < 0
72         error('m must be a nonnegative integer.')
73     end
74 elseif nargin == 5
75     fevals = 1e3;
76     if length(m) ~= 1 || mod(m,1) ~= 0 || m < 0
77         error('m must be a nonnegative integer.')
78     elseif length(T) ~= 1
79         error('T must be scalar.')
80     end
81 elseif nargin == 6
82     if length(m) ~= 1 || mod(m,1) ~= 0 || m < 0
83         error('m must be a nonnegative integer.')
84     elseif length(T) ~= 1
85         error('T must be scalar.')
86     elseif length(fevals) ~= 1 || mod(fevals,1) ~= 0 || fevals < 0
87         error('maxit must be a nonnegative integer.')
88     end
89 elseif nargin == 7
90     if length(m) ~= 1 || mod(m,1) ~= 0 || m < 0
91         error('m must be a nonnegative integer.')
92     elseif length(T) ~= 1
93         error('T must be scalar.')
94     elseif length(fevals) ~= 1 || mod(fevals,1) ~= 0 || fevals < 0
95         error('maxit must be a nonnegative integer.')
96     end
97 else
98     error('Not enough input arguments.');
99 end
100 % ensure that the input arguments 1-3 are correct
101 if ~isa(f,'function_handle')
102     error('Enter function handle.')
103 elseif isvector(v) ~= 1
104     error('v must be a vector.')
105 elseif length(dim) ~= 2 || isvector(dim) ~= 1
106     error('dim must be a two dimensional vector.')
107 end
108 % initialize dimensions
109 p = dim(1);
110 n = dim(2);
111 if mod(p,1) ~= 0 || p <= 0
```

```
112     error('Domain dimension must be a positive integer.')
113 elseif mod(n,1) ~= 0 || n <= 0
114     error('Codomain dimension must be a positive integer.')
115 end
116 % if entered, 'h' must be either scalar or a column vector and must
117 % not be or contain zero(s) in order to avoid parameter duplicates
118 if nargin == 7
119     w = size(h);
120     if isvector(h) ~= 1 || (w(1) ~= 1 && w(1) ~= p) || (w(2) ~= ...
            1 && w(2) ~= p)
121         error('h must be scalar or a p-dimensional vector.')
122     elseif ~all(h) == 1
123         error('h must not be (or contain) zero(s).')
124     elseif w(2) ~= 1
125     h = h';
126     end
127 end
128 % ensure that v is a column vector and determine its length
129 [lv, u] = size(v);
130 if u ~= 1
131     v = v';
132     lv = u; % length of v
133 end
134
135 %% Initialization
136 theta = v(1:p);
137 if lv-p == 0
138     % no data given -> standard nonlinear least squares problem
139     disp('solve STANDARD NLLS problem')
140     x = [];
141     y = zeros(n,1);
142 else
143     % data fitting problem
144     disp('solve DATA FITTING problem')
145     if lv ~= (p + 2*n)
146         error('Wrong parameter dimension or incorrect data set / ...
              sample size.')
147     end
148     x = v(p+1:n+p);
149     y = v(n+p+1:end);
150 end
151 Q = zeros(1,p+1);       % vector for storing residual sums of squares
152 F = zeros(n,p+1);       % matrix for storing function evaluations
153 I = eye(p);             % needed for the inversion of DeltaF'*DeltaF
154 diagonal = logical(I); % logical mask for diagonal of p-dim. matrix
155 fcount = 0;            % counter for function evaluations
156 icount = 0;            % iteration counter for main while loop
157 exitcount = 0;         % counter for successive triggering of the
158                        % exit condition
159
160 %% Start Routine
161 % p+1 starting values are required by DUD
162 % note: 'theta' always stands for theta_{p+1} in this code!
163 %% Construction of initial matrices 'Theta' and 'F'
164 % generate p more estimates from the starting guess 'theta'
165 % (= theta_{p+1}, the other parameters will be theta_1,...,theta_p)
166 % to do this generate px(p+1)-matrix where each column is 'theta'
167 Theta = theta * ones(1,p+1);
```

```
% then displace the diagonal entries of 'Theta' (i.e., the i-th
% component of 'theta') by 'h' to genererate p more parameters
% theta_i, i=1,...,p
if nargin < 7
% if no 'h' has been entered, we compute 'h = theta*0.1'
% however, we need to account for possible zeros in 'theta' which
% would result in zeros in 'h' and thus in duplications of 'theta'
% we do this by simply replacing occuring zeros by a small value
thetaTemp = theta;
thetaZeros = logical(~theta);  % logical mask for occuring zeros
thetaTemp(thetaZeros) = 1e-1;  % replace zeros
h = thetaTemp*0.1;             % 'h' contains no zeros
end
% generate the initial set of p+1 parameters
Theta(diagonal) = diag(Theta) + h;
% compute Q(theta_i) for all i and store function evaluations in 'F'
for i = 1:p+1;
    % Dud requires p+1 function evaluations for starting
    F(:,i) = f([Theta(:,i); x]);  fcount = fcount +1;
    Q(i) = norm( y - F(:,i) )^2;
end;
% we want to reorder the columns of 'Theta' and 'F' such that
% Q(theta_1) >= ... >= Q(theta_{p+1})
% sort Q(theta_i)'s in descending order
[Q,index] = sort(Q,'descend');
% 'index' is a vector containing the column indices of the
% corresponding vectors in 'Theta'
% reallocate vectors in 'Theta' in descending order according to the
% cost, i.e., such that the worst estimate is in the first column
% and the best estimate is in the last column
% consequently, the function evaluations 'F' must be rearranged
TTemp = Theta;
FTemp = F;
t = 1:p+1;
Theta = TTemp(:,index(t));
F = FTemp(:,index(t));
% now Q(theta_1) >= ... >= Q(theta_{p+1}) holds
theta = Theta(:,p+1); % assign theta_{p+1} the best estimate
% 'Theta' consists of p+1 ordered initial parameters
% the parameter yielding the highest cost, i.e., the first column
% of 'Theta' (theta_1) is declared the oldest member of the set of
% parameters 'Theta'
% theta_1 will be replaced by a better estimate 'theta_new' in the
% upcoming iteration if no convergence occurs
% 'theta_new' is the point between the linear approximation l
% and y calculated by the following procedure

%% Linear Approximation procedure
% the linear approximation of f is given by
% l(alpha) = f(theta_{p+1}) + deltaF * alpha
% display the currently best estimate for monitoring the behaviour
% of the upcoming approximation procedure
X = ['before approximation: Qold = ',num2str(Q(p+1))];
disp(X)
%% Generate function difference matrix 'DeltaF'
% 'DeltaF' is a nxp-matrix where the i-th column is given by
% f(theta_i) - f(theta_{p+1}), i = 1,...,p
DeltaF = bsxfun(@minus, F(:,1:p), F(:,p+1));
```

```
226 %% Compute 'alpha'
227 % 'alpha' (a vector) is the minimizer of the LLS problem
228 % Q(alpha) = (y-l(alpha))'(y-l(alpha))
229 % compute residual vector for the currently best estimate 'theta'
230 r = y - F(:,p+1);
231 % generate the (p+1)x(p+1)-dimensional block matrix
232 C = [DeltaF r]' * [DeltaF r];
233 % invoke the stepwise regression procedure, which can compute an
234 % approximate solution of the LLS problem even when the matrix
235 % ' C(1:p,1:p) = DeltaF' * DeltaF ' is essentially singular
236 alpha = stepwise_reg(C, p, sqrt(eps));
237
238 %% Generate parameter difference matrix 'DeltaTheta'
239 % 'DeltaTheta' is a pxp-matrix where the i-th column is given by
240 % theta_i - theta_{p+1}, i = 1,...,p
241 DeltaTheta = bsxfun(@minus, Theta(:,1:p), theta);
242
243 %% Compute 'theta_new'
244 theta_new = theta + DeltaTheta * alpha;
245 f_new = f([theta_new; x]); fcount = fcount + 1;
246 Qnew = norm( y - f_new )^2;
247 X = ['linear approximation: Qnew = ', num2str(Qnew)];
248 disp(X)
249
250 %% Perform line search (optional)
251 % Q(theta_{p+1}) is stored in row vector 'Q'
252 if Qnew >= Q(p+1) && m > 0
253     % invoke partial stepping procedure (subfunction)
254     [theta_new, f_new, Qnew, fcount] = part_step(f, fcount, ...
          theta_new, theta, F(:,p+1), x, y, p, n, Qnew, Q(p+1), m);
255 end
256 % the algorithm stops now if convergence has already been achieved
257 % the minimum number of function evaluations at this point is p+2
258 X = ['------> ',num2str(icount),'. iteration: Qnew = ', ...
      num2str(Qnew)];
259 disp(X)
260
261 %% Main iteration
262 % if convergence has not been achieved the new estimate theta_new
263 % is passed on to the next generation of the parameter set
264 % reiterate until a desired exit criteria is reached
265 while fcount < fevals && Qnew >= T
266     icount = icount + 1;
267     % additional stopping criterion (not mandatory)
268     % relative change in the residual sum of squares
269     if abs(Qnew - Q(p+1)) / Q(p+1) <= 1e-5
270         exitcount = exitcount + 1;
271              if exitcount == 5;
272              disp('STOP: relative change in Qnew is smaller than ...
                    T for 5 successive iterations')
273                 break
274              end
275     else
276         exitcount = 0;
277     end
278     %% Search: replace old parameter value(s) with new estimate(s)
279     % ensure that the search does not collapse into a subplane of
280     % the parameter space
```

```
281     % the component of alpha corresponding to the discarded
282     % parameter must not be zero!
283     % component is nonzero -> replace one member
284     if abs(alpha(1)) >= 1e-5
285     % replace oldest guess theta_1 by the new parameter
286     Theta(:,1) = theta_new;
287     % and replace the corresponding function evaluation
288     F(:,1) = f_new;
289     % and the residual sum of squares
290     Q(1) = Qnew;
291     else % component is zero -> replace two members of the set
292         clear s
293         clear index
294         % first: replace first theta_i with alpha_i >= 1e-5
295         index = find(abs(alpha) >= 1e-5); % index is vector
296         if isempty(index)
297             disp('hint: alpha is or is close to being the zero ...
                    vector')
298             % old parameter values are not retained indefinitely
299             % replace oldest estimate theta_1 by a new parameter
300             theta_newone = (Theta(:,1) + theta_new) / 2;
301             Theta(:,1) = theta_newone;
302             % calculate Q and evaluate f for the new parameter
303             f_newone = f([theta_newone; x]); fcount = fcount + 1;
304             F(:,1) = f_newone;
305             Q(1) = norm( y - f_newone )^2;
306         else
307         s = index(1); % first index where alpha >= 1e-5
308         X = ['update rule: alpha(1) < 1e-5 -> 1. and ', ...
                num2str(s), '. parameters are replaced'];
309         disp(X)
310         Theta(:,s) = theta_new;
311         F(:,s) = f_new;
312         Q(s) = Qnew;
313         % shift columns so that theta_new is the newest member of
314         % the parameter set, i.e., the (p+1)-th entry of 'Theta'
315         Theta(:,s:end) = circshift(Theta(:,s:end),[0 -1]);
316         F(:,s:end) = circshift(F(:,s:end),[0 -1]);
317         Q(s:end) = circshift(Q(s:end),[0 -1]);
318         % second: old parameter values are not retained indefinitely
319         % replace oldest guess theta_1 by a new parameter
320         theta_newone = (Theta(:,1) + theta_new) / 2;
321         Theta(:,1) = theta_newone;
322         % calculate Q and f for the new parameter
323         f_newone = f([theta_newone; x]); fcount = fcount + 1;
324         F(:,1) = f_newone;
325         Q(1) = norm( y - f_newone )^2;
326         end
327     end
328     % shift columns so that the oldest member gets the first entry
329     % and the newest member gets the (p+1)-th entry
330     Theta = circshift(Theta,[0 -1]);
331     F = circshift(F,[0 -1]);
332     Q = circshift(Q,[0 -1]);
333     theta = Theta(:,p+1); % theta_{p+1} is the new estimate
334     %% Linear Approximation procedure
335     % the linear approximation of f is given by
336     % l(alpha) = f(theta_{p+1}) + deltaF * alpha
```

```
337      %% Generate new function difference matrix 'DeltaF'
338      % 'DeltaF' is a nxp-matrix where the i-th column is given by
339      % f(theta_i) - f(theta_{p+1}), i = 1,...,p
340      DeltaF = bsxfun(@minus, F(:,1:p), F(:,p+1));
341
342      %% Compute new minimizer 'alpha'
343      % every iteration consists of minimizing the LLS problem
344      % Q(alpha)= (y-l(alpha))'(y-l(alpha))
345      % residual vector for the currently best estimate 'theta':
346      r = y - F(:,p+1);
347      % generate (p+1)x(p+1)-dimensional block matrix
348      C = [DeltaF r]' * [DeltaF r];
349      % invoke the stepwise regression procedure, which can compute
350      % an approximate solution of the LLS problem even when
351      % ' C(1:p,1:p) = DeltaF' * DeltaF ' is essentially singular
352      alpha = stepwise_reg(C, p, sqrt(eps));
353
354      %% Generate new parameter difference matrix 'DeltaTheta'
355      % 'DeltaTheta' is a pxp-matrix where the i-th column is given
356      %by theta_i - theta_{p+1}, i = 1,...,p
357      DeltaTheta = bsxfun(@minus, Theta(:,1:p), theta);
358
359      %% Compute 'theta_new'
360      theta_new = theta + DeltaTheta * alpha;
361      f_new = f([theta_new; x]); fcount = fcount + 1;
362      Qnew = norm( y - f_new )^2;
363      X = ['linear approximation: Qnew = ', num2str(Qnew)];
364      disp(X)
365
366      %% Perform line search (optional)
367      % Q(theta_{p+1}) is stored in row vector 'Q'
368      if Qnew >= Q(p+1) && m > 0
369      % invoke partial stepping procedure (subfunction)
370      [theta_new, f_new, Qnew, fcount] = part_step(f, fcount, ...
            theta_new, theta, F(:,p+1), x, y, p, n, Qnew, Q(p+1), m);
371      end
372
373      %% Align displayed arrow (unessential)
374      if icount < 1e1
375      X = ['------> ',num2str(icount),'. iteration: Qnew = ', ...
            num2str(Qnew)];
376      disp(X)
377      elseif icount < 1e2
378      X = ['-----> ',num2str(icount),'. iteration: Qnew = ', ...
            num2str(Qnew)];
379      disp(X)
380      else
381      X = ['----> ',num2str(icount),'. iteration: Qnew = ', ...
            num2str(Qnew)];
382      disp(X)
383      end
384  end % end while (main iteration)
385  tcount = toc; % end timer
386  end % end DUD
387
388  %% DUD subfunctions part_step and stepwise_reg
389  %% Line search
```

```
390 function [theta_new, f_new, Qnew, fcount] = part_step(f, fcount, ...
        theta, theta_old, f_old, x, y, p, n, Q, Qold, m)
391 %% PARTIAL STEPPING procedure
392 % for finding better parameters estimates in DUD
393 % -> part_step picks points on the line between theta and theta_old
394 %
395 %% INPUT:
396 % f - response function which takes as input a parameter and the
397 %     explanatory vector x
398 % fcount - current number of function evaluations
399 % theta - current parameter which needs replacement
400 % theta_old - old 'best' parameter
401 % f_old - function evaluation of theta_old
402 % x - the explanatory vector
403 % y - observed data vector
404 % p - dimension of parameter vectors theta and theta_old
405 % n - dimension of x and y
406 % Q - RSS for theta
407 % Qold - RSS for theta_old
408 % m - integer, maximal m line searches (iterations) may be performed
409 %
410 %% OUTPUT:
411 % theta_new - new 'best' parameter or last found parameter from
412 %             iteration m if no improvement in RSS was achieved
413 % f_new - function evaluation at theta_new
414 % Q_new - RSS for theta_new
415
416 %% Initialization
417 l = 1; % iteration counter
418 % store new estimates (and their evaluation) found by the search
419 Tstor = zeros(p,m);
420 Fstor = zeros(n,m);
421 % and store the residual sum of squares for this estimates
422 Qstor = zeros(1,m);
423 Qnew = Q;
424
425 %% Main procedure
426 % find point 'theta_new' on the line between 'theta' and 'theta_old'
427 % that yields a smaller RSS than 'Qold'
428 while Qnew >= Qold && l <= m
429     % choose a point on the line between 'theta' and 'theta_old'
430     d = -(-1/2)^l;
431     theta_new = d * theta + (1 - d) * theta_old;
432     f_new = f([theta_new; x]); fcount = fcount + 1;
433     Qnew = norm(y - f_new)^2;
434     % store findings
435     Tstor(:,l) = theta_new;
436     Fstor(:,l) = f_new;
437     Qstor(l) = Qnew;
438     if l < 10 % display RSS
439         X = ['line search: m = ',num2str(l), '    Qnew = ', ...
                num2str(Qnew)];
440         disp(X)
441     else       % unessential (align displayed spacing)
442         X = ['line search: m = ',num2str(l), '   Qnew = ', ...
                num2str(Qnew)];
443         disp(X)
444     end
```

```
445     l = l + 1;
446 end
447 % if no reduction in the RSS has been achieved, the last found
448 % parameter 'theta_new' is kept
449 % exception: it might happen that the evaluation of this parameter
450 % blows up which would consequently yield NaN in the matrix 'DeltaF'
451 % if so, we choose the best found estimate by the search instead
452 if Qnew == Inf
453     if ismember(Inf,f_new)
454         [Qnew, index] = min(Qstor);
455         theta_new = Tstor(:,index);
456         f_new = Fstor(:,index);
457         if Qnew == Inf
458             if ismember(Inf,f_new) && ~ismember(Inf,f_old)
459                 Qnew = Qold;
460                 theta_new = theta_old;
461                 f_new = f_old;
462                 disp('line search: unsuccessful -> old parameter ...
                        is kept')
463             end
464         else
465             disp('line search: f_new contains Inf -> best found ...
                    parameter is kept')
466         end
467     end
468 end
469 end
470
471 %% Stepwise regression
472 function beta = stepwise_reg(C, p, T)
473 %% STEPWISE REGRESSION procedure using Gauss-Jordan elimination
474 % for approximately solving the normal equations X'X*beta = X'*y
475 % X'X may be ill-conditioned or singular (if X has no full rank)
476 % -> stepwise_reg uses an in-place Gauss-Jordan algorithm for matrix
477 % inversion, i.e., no augmentation with the identity is needed
478 % -> the order of pivoting is determined by stepwise regression
479 % -> problematic diagonal elements remain unpivoted for robusteness
480 %
481 %% INPUT:
482 % C - dxd-dimensional augmented matrix of the form [X y]' * [X y],
483 %     where X is a nxp-matrix and y is a nx1-vector, i.e., d = p+1
484 % p - number of columns of X;
485 %     each column x_i, i = 1,...,p, of X may be interpreted as an
486 %     independent variable entering a linear regression equation
487 % T - threshold for the pivot tolerance 'tol', meaning that some
488 %     diagonal element remains unpivoted if 'tol' < T
489 %
490 %% OUTPUT:
491 % beta - p-dimensional solution vector whose components can be
492 %        understood as regression coefficients which result from
493 %        linear regression of the dependent variable y on the
494 %        columns of X (the independent variables)
495 %        -> components of beta are set to zero if the corresponding
496 %        diagonal elements of C are unpivoted by means of the
497 %        Gauss-Jordan algorithm
498
499 %% Initialization
500 d = p+1;
```

```
501 beta = zeros(p,1);
502 % only components of 'beta' corresponding to pivoted rows in 'C'
503 % get new values
504 initD = diag(C(1:p,1:p)); % initial diagonal values for tolerance
505 unpivoted = 1:p;      % start with set of indices for unpivoted rows
506 pivoted = zeros(1,p); % empty storage for pivoted row indices
507 % pivot on the first p diagonal entries of the matrix 'C'
508 for i = 1:p % add variable x_k, k in {1,...,p}, to the regression
509     % among all the unpivoted diagonal entries pivot on the diagonal
510     % entry (k,k) whos reduction in the RSS of y 'red_RSS' from
511     % adding the variable x_k to the regression is maximal
512     red_RSS = C(unpivoted,d).^2 ./ diag(C(unpivoted,unpivoted));
513     % find the index k which corresponds to the greatest reduction
514     [~,k] = max(red_RSS);
515     k = unpivoted(k);
516     unpivoted(unpivoted==k) = []; % do not use this index again
517     % pivot tolerance is used to prevent a complete inversion of the
518     % matrix X'X if problematic pivots are encountered
519     a = C(k,k);
520     tol = a/initD(k);
521     if tol < T || isnan(tol) % tolerance criterion
522         X = ['GJ pivoting: row ' num2str(k) ' is not swept ...
                (tolerance = ', num2str(tol), ')'];
523         disp(X)
524         continue
525     end
526     % in-place Gauss-Jordan elimination for matrix inversion
527     % computations for the k-th row
528     C(k,:) = C(k,:)/a;
529     C(k,k) = C(k,k)/a; % '1/C(k,k)' at pivot element
530     % computations for the remaining rows
531     t = 1:d;
532     t(k) = [];          % exclude the k-th row from computations
533     b = C(t,k);
534     C(t,t) = C(t,t) - b * C(k,t);
535     C(t,k) = -b./a;
536     pivoted(i) = k;     % add index to regressed rows
537     % ordinary least square estimate beta_k is stored in 'C(k,p+1)'
538 end
539 regressed = nonzeros(pivoted); % indices of variables in the
540 % regression model - 'beta' gets their corresponding coefficients
541 beta(regressed) = C(regressed, d);
542 % the remaining components of 'beta' are zero
543 end
```

C.6 Modified Gauss-Jordan Algorithm

```
function [ Ainv, B, pivoted, tcount ] = gjinv( A )
%% GAUSS-JORDAN ELIMINATION with diagonal pivoting
%       for computing the inverse of a square matrix;
% order of pivoting: picks diagonal entries with larger absolute
%                    value first
% caution: does not work for matrices with zero diagonal but is
%          stable for positive/negative definite matrices
%% AUTHOR: Stefan Scheer
%% REFERENCE:
% R.I. Jennrich and P.F. Sampson. "Application of Stepwise
% Regression to Non-Linear Estimation". In: Technometrics 10.1
% (1968), pp. 63-72.
%
%% INPUT
% A - nonsingular square matrix
%
%% OUTPUT
% Ainv - inverse of A obtained by pivoting on the diagonal elements
% B - test matrix B = Ainv * A, i.e., B should be the identity
% pivoted - row vector which demonstrates the order of pivoting
% tcount - elapsed time
%
%% EXAMPLE
% A = [1 3 -1; 2 -6 7; 5 3 8]; or A = [ 0 1 -3; 2 0 7; 0 3 6];
% [Ainv, used_rows, B, tcount]= gjinv( A )

tic; % start timer

%% Ensuring correct input
[n,m] = size(A);
if n ~= m || n == 1
    error('enter square matrix')
end

%% Initialization
InitA = A;
unpivoted = 1:n;          % indices of unpivoted rows
if nargout >= 3
    pivoted = zeros(1,n); % storage for pivoted indices
end

%% Check for diagonal zeros
if  ~any(diag(A)) == 1
    error('inversion not possible due to zero diagonal')
elseif all(diag(A)) == 0
    disp('caution: diagonal zero(s) encountered')
    disp('-> check if computed inverse is accurate')
end

%% Check if the matrix is ill-conditioned/singular
if cond(A) >= 1/sqrt(eps)
    disp('caution: matrix is ill-conditioned or singular')
    disp('-> the computed inverse is not accurate')
end

```

```
56 %% Main procedure
57 for i = 1:n
58     % among all unpivoted diagonal entries pivot on the diagonal
59     % entry (k,k) that has the largest absolute value
60     D = abs(diag(A(unpivoted,unpivoted)));
61     [~,k] = max(D); % find the index of largest diagonal value
62     k = unpivoted(k);
63     unpivoted(unpivoted==k) = []; % do not use this index again
64     % computations for the k-th row
65     a = A(k,k);
66     A(k,:) = A(k,:)/a;
67     A(k,k) = A(k,k)/a; % '1/A(k,k)' at pivot element
68     % computations for the remaining rows
69     t = 1:n;
70     t(k) = [];          % exclude the k-th row from computations
71     b = A(t,k);
72     A(t,t) = A(t,t) - b*A(k,t);
73     A(t,k) = -b./a;
74     if nargout >= 3
75         pivoted(i) = k;
76     end
77 end
78 Ainv = A;
79 tcount = toc; % end timer
80 % test if Ainv is inverse (do not count time for this test)
81 if nargout >= 2
82     B = Ainv * InitA;
83 end
84 end
```

C.7 Moré, Garbow and Hillstrom Collection

```
function [ F, J ] = rosenbrock2D( x )
%% ROSENBROCK function for testing
%   unconstrained optimization software
%% AUTHOR: Stefan Scheer
%% REFERENCE:
% Test function (1) in
% "Testing unconstrained optimization software"
% by J.J. More, B.S. Garbow, and K.E. Hillstrom,
% ACM Trans. Math. Softw., vol.7, pp. 17-41, Mar. 1981.
%
%% INPUT:
% x - two-dimensional vector
%
%% OUTPUT:
% F - two-dimensional column vector
% J - 2x2-dimensional Jacobian matrix
%
%% EXAMPLE:
% x = [-1.2;1];
% [F, J] = rosenbrock2D(x)

%% Ensuring correct input
% ensure that vector is entered
if isvector(x) ~= 1
    error('x must be a vector.')
% ensure that dimension of vector is correct
elseif length(x) ~= 2
    error('Enter two-dimensional vector.')
end

%% Main procedure
F = [10*(x(2)-x(1)^2); 1-x(1)];
if nargout > 1 % compute the Jacobian
    J = [-20*x(1)  10; -1 0];
end
end
```

```
function [ F, J ] = freuden_roth( x )
%% FREUDENSTEIN AND ROTH function for testing
%   unconstrained optimization software
%% AUTHOR: Stefan Scheer
%% REFERENCE:
% Test function (2) in
% "Testing unconstrained optimization software"
% by J.J. More, B.S. Garbow, and K.E. Hillstrom,
% ACM Trans. Math. Softw., vol.7, pp. 17-41, Mar. 1981.
%
%% INPUT:
% x - two-dimensional vector
%
%% OUTPUT:
% F - two-dimensional column vector
% J - 2x2-dimensional Jacobian matrix
```

```
%% EXAMPLE:
% x = [.5; -2];
% [F, J] = freuden_roth(x)

%% Ensuring correct input
% ensure that vector is entered
if isvector(x) ~= 1
    error('x must be a vector.')
% ensure that dimension of vector is correct
elseif length(x) ~= 2
    error('Enter two-dimensional vector.')
end

%% Main procedure
F = [-13+x(1)+((5-x(2))*x(2)-2)*x(2); ...
    -29+x(1)+((x(2)+1)*x(2)-14)*x(2)];
if nargout > 1 % compute the Jacobian
    J = [1 10*x(2)-3*x(2)^2-2; 1 3*x(2)^2+2*x(2)-14];
end
end
```

```
function [ F, J ] = powell_badly( x )
%% POWELL BADLY SCALED function for testing
%   unconstrained optimization software
%% AUTHOR: Stefan Scheer
%% REFERENCE:
% Test function (3) in
% "Testing unconstrained optimization software"
% by J.J. More, B.S. Garbow, and K.E. Hillstrom,
% ACM Trans. Math. Softw., vol.7, pp. 17-41, Mar. 1981.
%
%% INPUT:
% x - two-dimensional vector
%
%% OUTPUT:
% F - two-dimensional column vector
% J - 2x2-dimensional Jacobian matrix
%
%% EXAMPLE:
% x = [0; 1];
% [F, J] = powell_badly(x)

%% Ensuring correct input
% ensure that vector is entered
if isvector(x) ~= 1
    error('x must be a vector.')
% ensure that dimension of vector is correct
elseif length(x) ~= 2
    error('Enter two-dimensional vector.')
end
%% Main procedure
F = [10^4*x(1)*x(2)-1; exp(-x(1))+exp(-x(2))-1.0001];
if nargout > 1 % compute the Jacobian
    J = [10^4*x(2) 10^4*x(1); -exp(-x(1)) -exp(-x(2))];
end
end
```

```
function [ F, J ] = brown_badly( x )
%% BROWN BADLY SCALED function for testing
%   unconstrained optimization software
%% AUTHOR: Stefan Scheer
%% REFERENCE:
% Test function (4) in
% "Testing unconstrained optimization software"
% by J.J. More, B.S. Garbow, and K.E. Hillstrom,
% ACM Trans. Math. Softw., vol.7, pp. 17-41, Mar. 1981.
%
%% INPUT:
% x - two-dimensional vector
%
%% OUTPUT:
% F - three-dimensional column vector
% J - 3x2-dimensional Jacobian matrix
%
%% EXAMPLE:
% x = [1; 1];
% [F, J] = brown_badly(x)

%% Ensuring correct input
% ensure that vector is entered
if isvector(x) ~= 1
    error('x must be a vector.')
% ensure that dimension of vector is correct
elseif length(x) ~= 2
    error('Enter two-dimensional vector.')
end
%% Main procedure
F = [x(1) - 10^6; x(2) - 2e-6; x(1)*x(2) - 2];
if nargout > 1 % compute the Jacobian
    J = [1 0; 0 1; x(2) x(1)];
end
end
```

```
function [ F, J ] = beale( x )
%% BEALE function for testing
%   unconstrained optimization software
%% AUTHOR: Stefan Scheer
%% REFERENCE:
% Test function (5) in
% "Testing unconstrained optimization software"
% by J.J. More, B.S. Garbow, and K.E. Hillstrom,
% ACM Trans. Math. Softw., vol.7, pp. 17-41, Mar. 1981.
%
%% INPUT:
% x - two-dimensional vector
%
%% OUTPUT:
% F - three-dimensional vector
% J - 3x2-dimensional Jacobian matrix
%
%% EXAMPLE:
% x = [1; 1];
% [y, J] = beale(x)
```

```
21 %% Initialization
22 n = length(x); % input dimension
23
24 %% Ensuring correct input
25 % ensure that vector is entered
26 if isvector(x) ~= 1
27     error('x must be a vector.')
28 % ensure that dimension of vector is correct
29 elseif n ~= 2
30     error('Enter two-dimensional vector.')
31 end
32
33 %% Main procedure
34 z = [1.5, 2.25, 2.625]';
35 t = (1:3)';
36 F = z - x(1)*(1 - x(2).^t);
37 if nargout > 1 % compute the Jacobian
38     J = zeros(3,2);
39     J(:,1) = -1 + x(2).^t;
40     J(:,2) = x(1).*t.*x(2).^(t-1);
41 end
42 end
```

```
1  function [ F, J ] = jennrich_sampson( x )
2  %% JENNRICH AND SAMPSON function for testing
3  %   unconstrained optimization software
4  %% AUTHOR: Stefan Scheer
5  %% REFERENCE:
6  % Test function (6) in
7  % "Testing unconstrained optimization software"
8  % by J.J. More, B.S. Garbow, and K.E. Hillstrom,
9  % ACM Trans. Math. Softw., vol.7, pp. 17-41, Mar. 1981.
10 %
11 %% INPUT:
12 % x - two-dimensional vector
13 %
14 %% OUTPUT:
15 % F - m-dimensional column vector, where m >= 2 (default m = 10)
16 % J - mx2-dimensional Jacobian matrix
17 %
18 %% EXAMPLE:
19 % x = [0.3; 0.4];
20 % [F, J] = jennrich_sampson(x)
21
22 %% Choosing output dimension
23 m = 10;
24
25 %% Ensuring correct input
26 % ensure that vector is entered
27 if isvector(x) ~= 1
28     error('x must be a vector.')
29 % ensure that dimension of vector is correct
30 elseif length(x) ~= 2
31     error('Enter two-dimensional vector.')
32 end
33
34 %% Main procedure
```

```
t = (1:m)';
F = 2 + 2*t - (exp(t*x(1)) + exp(t*x(2)));
if nargout > 1 % compute the Jacobian
    J = zeros(m,2);
    J(:,1) = -t.*exp(t*x(1));
    J(:,2) = -t.*exp(t*x(2));
end
end
```

```
function [ F, J ] = helical( x )
%% HELICAL VALLEY function for testing
%   unconstrained optimization software
%% AUTHOR: Stefan Scheer
%% REFERENCE:
% Test function (7) in
% "Testing unconstrained optimization software"
% by J.J. More, B.S. Garbow, and K.E. Hillstrom,
% ACM Trans. Math. Softw., vol.7, pp. 17-41, Mar. 1981.
%
%% INPUT:
% x - three-dimensional vector
%
%% OUTPUT:
% F - three-dimensional column vector
% J - 3x3-dimensional Jacobian matrix
%
%% EXAMPLE:
% x = [-1; 0; 0];
% [F, J] = helical(x)

%% Ensuring correct input
% ensure that vector is entered
if isvector(x) ~= 1
    error('x must be a vector.')
% ensure that dimension of vector is correct
elseif length(x) ~= 3
    error('Enter three-dimensional vector.')
end

%% Main procedure
F = zeros(3,1);
x2byx1 = x(2)/x(1);
x1squared = x(1)^2;
x2squared = x(2)^2;
if x(1) >= 0
    if x(1) == 0
        disp('caution: singularity in helical valley function ...
            evaluation')
    end
    F(1) = 10*( x(3) - 10*( (1/(2*pi))*atan(x2byx1) ) );
else
    F(1) = 10*( x(3) - 10*( (1/(2*pi))*atan(x2byx1) + .5 ) );
end
lengthx = sqrt(x1squared + x2squared);
F(2) = 10*(lengthx - 1);
F(3) = x(3);
if nargout > 1 % compute the Jacobian
```

```
48      J = zeros(3,3);
49      fiftybypi = 50/pi;
50      oneplussquare = 1 + (x2byx1)^2;
51      tenbylengthx = 10/lengthx;
52      J(1,1) = fiftybypi*x2byx1^2/oneplussquare;
53      J(1,2) = (-fiftybypi/x(1))/oneplussquare;
54      J(1,3) = 10;
55      J(2,1) = x(1)*tenbylengthx;
56      J(2,2) = x(2)*tenbylengthx;
57      J(3,3) = 1;
58  end
59  end
```

```
1   function [ F, J ] = bard( x )
2   %% BARD function for testing
3   %   unconstrained optimization software
4   %% AUTHOR: Stefan Scheer
5   %% REFERENCE:
6   % Test function (8) in
7   % "Testing unconstrained optimization software"
8   % by J.J. More, B.S. Garbow, and K.E. Hillstrom,
9   % ACM Trans. Math. Softw., vol.7, pp. 17-41, Mar. 1981.
10  %
11  %% INPUT:
12  % x - three-dimensional vector
13  %
14  %% OUTPUT:
15  % F - 15-dimensional vector
16  % J - 15x3-dimensional Jacobian matrix
17  %
18  %% EXAMPLE:
19  % x = [1; 1; 1];
20  % [F, J] = bard(x)
21
22  %% Ensuring correct input
23  % ensure that vector is entered
24  if isvector(x) ~= 1
25      error('x must be a vector.')
26  end
27
28  %% Initialization
29  n = length(x);
30  % ensure that dimension of vector is correct
31  if n ~= 3
32      error('Enter three-dimensional vector.')
33  end
34
35  %% Main procedure
36  u = (1:15)';
37  v = 16-u;
38  w = min(u,v);
39  y = [.14, .18, .22, .25, .29, .32, .35, .39, .37, .58, .73, .96, ...
        1.34, 2.1, 4.39]';
40  z = v.*x(2) + w.* x(3);
41  F = y - (x(1) + u./z);
42  if nargout > 1 % compute the Jacobian
43      J = zeros(15,3);
```

```
    ubyzsquared = u.*z.^(-2);
    J(:,1) = -1;
    J(:,2) = v.*ubyzsquared;
    J(:,3) = w.*ubyzsquared;
end
end
```

```
function [ F, J ] = gaussian( x )
%% GAUSSIAN function for testing
%   unconstrained optimization software
%% AUTHOR: Stefan Scheer
%% REFERENCE:
% Test function (9) in
% "Testing unconstrained optimization software"
% by J.J. More, B.S. Garbow, and K.E. Hillstrom,
% ACM Trans. Math. Softw., vol.7, pp. 17-41, Mar. 1981.
%
%% INPUT:
% x - three-dimensional vector
%
%% OUTPUT:
% F - 15-dimensional vector
% J - 15x3-dimensional Jacobian matrix
%
%% EXAMPLE:
% x = [.4; 1; 0];
% [F, J] = gaussian(x)

%% Ensuring correct input
% ensure that vector is entered
if isvector(x) ~= 1
    error('x must be a vector.')
end

%% Initialization
n = length(x);
% ensure that dimension of vector is correct
if n ~= 3
    error('Enter three-dimensional vector.')
end

%% Main procedure
t = (8 - (1:15)')/2;
y = [.0009, .0044, .0175, .0540, .1295, .2420, .3521, .3989, ...
    .3521, .2420, .1295, .0540, .0175, .0044, .0009]';
exponent = exp(-x(2)*(t - x(3)).^2/2);
F = x(1)*exponent - y;
if nargout > 1 % compute the Jacobian
    J = zeros(15,3);
    tminusx3 = t - x(3);
    J(:,1) = exponent;
    J(:,2) = x(1)*-tminusx3.^2/2.*exponent;
    J(:,3) = x(1)*x(2)*tminusx3.*exponent;
end
end
```

```
function [ F, J ] = meyer( x )
%% MEYER function for testing
%   unconstrained optimization software
%% AUTHOR: Stefan Scheer
%% REFERENCE:
% Test function (10) in
% "Testing unconstrained optimization software"
% by J.J. More, B.S. Garbow, and K.E. Hillstrom,
% ACM Trans. Math. Softw., vol.7, pp. 17-41, Mar. 1981.
%
%% INPUT:
% x - three-dimensional vector
%
%% OUTPUT:
% F - 16-dimensional vector
% J - 16x3-dimensional Jacobian matrix
%
%% EXAMPLE:
% x = [.02; 4000; 250];
% [F, J] = meyer(x)

%% Ensuring correct input
% ensure that vector is entered
if isvector(x) ~= 1
    error('x must be a vector.')
end

%% Initialization
n = length(x);
% ensure that dimension of vector is correct
if n ~= 3
    error('Enter three-dimensional vector.')
end

%% Main procedure
s = (1:16)';
t = 45 + 5*s;
y = [34780, 28610, 23650, 19630, 16370, 13720, 11540, 9744, ...
    8261, 7030, 6005, 5147, 4427, 3820, 3307, 2872]';
u = (t + x(3));
v = x(2)./u;
exponent = exp(v);
F = x(1)*exponent - y;
if nargout > 1 % compute the Jacobian
    J = zeros(16,3);
    w = x(1)./u;
    J(:,1) = exponent;
    J(:,2) = w.*exponent;
    J(:,3) = -v.*w.*exponent;
end
end
```

```
function [ F, J ] = gulf( x )
%% GULF RESEARCH AND DEVELOPMENT function for testing
%   unconstrained optimization software
%% AUTHOR: Stefan Scheer
```

```
5  %% REFERENCE:
6  % Test function (11) in
7  % "Testing unconstrained optimization software"
8  % by J.J. More, B.S. Garbow, and K.E. Hillstrom,
9  % ACM Trans. Math. Softw., vol.7, pp. 17-41, Mar. 1981.
10 %
11 %% INPUT:
12 % x - three-dimensional vector
13 %
14 %% OUTPUT:
15 % F - m-dimensional vector, where n <= m =< 100 (default m = 10)
16 % J - mx3-dimensional Jacobian matrix
17 %
18 %% EXAMPLE:
19 % x = [5; 2.5; .15];
20 % [F, J] = gulf(x)
21
22 %% Determining output dimension
23 m = 10;
24
25 %% Ensuring correct input
26 % ensure that vector is entered
27 if isvector(x) ~= 1
28     error('x must be a vector.')
29 end
30 %% Initialization
31 n = length(x);
32 % ensure that dimension of vector is correct
33 if n ~= 3
34     error('Enter three-dimensional vector.')
35 end
36
37 %% Main procedure
38 t = (1:m)'/100;
39 y = 25 + (-50*log(t)).^(2/3);
40 yminusx2 = y - x(2);
41 absyminusx2 = abs(yminusx2);
42 u = absyminusx2.^x(3);
43 v = u/x(1);
44 if x(1) == 0
45     disp('caution: singularity in gulf research and development ...
          function evaluation')
46 end
47 expminusv = exp(-v);
48 F = expminusv - t;
49 if nargout == 2 % compute the Jacobian
50     J = zeros(m,3);
51     w = v.*expminusv;
52     z = (x(3)/x(1))*(absyminusx2.^(x(3) - 1));
53     J(:,1) = w/x(1);
54     if (yminusx2 >= 0)
55         J(:,2) = z.*expminusv;
56     else
57         J(:,2) = -z.*expminusv;
58     end
59     J(:,3) = -log(absyminusx2).*w;
60 end
61 end
```

```
function [ F, J ] = box3D( v )
%% BOX THREE-DIMENSIONAL function for testing
%   unconstrained optimization software
%% AUTHOR: Stefan Scheer
%% REFERENCE:
% Test function (12) in
% "Testing unconstrained optimization software"
% by J.J. More, B.S. Garbow, and K.E. Hillstrom,
% ACM Trans. Math. Softw., vol.7, pp. 17-41, Mar. 1981.
%
%% INPUT:
% v - vector containing a three-dimensional parameter vector theta
%     and a n-dimensional data vector x,
%     i.e., v = [theta;x] for column vectors theta and x
%
%% OUTPUT:
% F - n-dimensional column vector
% J - nx3-dimensional Jacobian matrix
%
%% EXAMPLE:
% v = [[0; 10; 20]; (.1:.1:1)'];
% [F, J ] = box3D( v )

%% Ensuring correct input
% ensure that vector is entered
if isvector(v) ~= 1
    error('v must be a vector.')
% ensure that parameter dimension is correct
elseif length(v) < 3
    error('Enter three-dimensional parameter theta.')
end

%% Initialization
theta = v(1:3);
x = v(4:end);
n = length(x);

%% Ensuring that data is entered
if n == 0
    error('Enter coefficients for Box''s function.')
end

%% Main procedure
u = -theta(1).*x;
v = -theta(2).*x;
w = exp(-x) - exp(-10.*x);
expu = exp(u);
expv = exp(v);
F = expu - expv - theta(3).*w;
if nargout > 1 % compute the Jacobian
    J = zeros(n,3);
    J(:,1) = -x.*expu;
    J(:,2) = x.*expv;
    J(:,3) = -w;
end
```

```
function [ F, J ] = powell_singular( x )
%% POWELL SINGULAR FUNCTION for testing
%   unconstrained optimization software
%% AUTHOR: Stefan Scheer
%% REFERENCE:
% Test function (13) in
% "Testing unconstrained optimization software"
% by J.J. More, B.S. Garbow, and K.E. Hillstrom,
% ACM Trans. Math. Softw., vol.7, pp. 17-41, Mar. 1981.
%
%% INPUT:
% x - four-dimensional vector
%
%% OUTPUT:
% F - four-dimensional vector
% J - 4x4-dimensional Jacobian matrix
%
%% EXAMPLE:
% x = [3; -1; 0; 1];
% [F, J] = powell_singular(x)

%% Ensuring correct input
if isvector(x) ~= 1
    error('x must be a vector.')
end

%% Initialization
n = length(x);
% ensure that dimension of vector is correct
if n ~= 4
    error('Enter four-dimensional vector.')
end

%% Main procedure
a = 5^(1/2);
b = x(2) - 2*x(3);
c = x(1) - x(4);
d = 10^(1/2);
F = [x(1) + 10*x(2); a*(x(3) - x(4)); b^2; d*c^2];
if nargout > 1 % compute the Jacobian
    J = zeros(n,n);
    twodc = 2*d*c;
    J(1,1) = 1;
    J(1,2) = 10;
    J(2,3) = a;
    J(2,4) = -a;
    J(3,2) = 2*b;
    J(3,3) = -4*b;
    J(4,1) = twodc;
    J(4,4) = -twodc;
end
end
```

```
function [ F, J ] = wood( x )
%% WOOD FUNCTION for testing
%   unconstrained optimization software
%% AUTHOR: Stefan Scheer
%% REFERENCE:
% Test function (14) in
% "Testing unconstrained optimization software"
% by J.J. More, B.S. Garbow, and K.E. Hillstrom,
% ACM Trans. Math. Softw., vol.7, pp. 17-41, Mar. 1981.
%
%% INPUT:
% x - four-dimensional vector
%
%% OUTPUT:
% F - six-dimensional vector
% J - 6x4-dimensional Jacobian matrix
%
%% EXAMPLE:
% x = [-3; -1; -3; -1];
% [F, J] = wood(x)

%% Ensuring correct input
if isvector(x) ~= 1
    error('x must be a vector.')
end

%% Initialization
n = length(x);
% ensure that dimension of vector is correct
if n ~= 4
    error('Enter four-dimensional vector.')
end

%% Main procedure
a = sqrt(90);
b = sqrt(10);
c = 1/b;
F = [10*(x(2)-x(1)^2); 1-x(1); a*(x(4)-x(3)^2); 1-x(3); ...
    b*(x(2)+x(4)-2); c*(x(2)-x(4))];
if nargout > 1 % compute the Jacobian
    J = zeros(6,n);
    J(1,1) = -20*x(1);
    J(1,2) = 10;
    J(2,1) = -1;
    J(3,3) = -2*a*x(3);
    J(3,4) = a;
    J(4,3) = -1;
    J(5,2) = b;
    J(5,4) = b;
    J(6,2) = c;
    J(6,4) = -c;
end
end
```

```
function [ F, J ] = kowalik_osborne( x )
%% KOWALIK AND OSBORNE FUNCTION for testing
%   unconstrained optimization software
%% AUTHOR: Stefan Scheer
%% REFERENCE:
% Test function (15) in
% "Testing unconstrained optimization software"
% by J.J. More, B.S. Garbow, and K.E. Hillstrom,
% ACM Trans. Math. Softw., vol.7, pp. 17-41, Mar. 1981.
%
%% INPUT:
% x - four-dimensional vector
%
%% OUTPUT:
% F - 11-dimensional vector
% J - 11x4-dimensional Jacobian matrix
%
%% EXAMPLE:
% x = [.25; .39; .415; .39];
% [F, J] = kowalik_osborne(x)

%% Ensuring correct input
if isvector(x) ~= 1
    error('x must be a vector.')
end

%% Initialization
n = length(x);
% ensure that dimension of vector is correct
if n ~= 4
    error('Enter four-dimensional vector.')
end
y = [.1957; .1947; .1735; .1600; .0844; .0627; .0456; .0342; ...
    .0323; .0235; .0246];
u = [4; 2; 1; .5; .25; .167; .1250; .1; .0833; .0714; .0625];

%% Main procedure
v = u.^2;
a = v + u.*x(2);
b = v + u.*x(3) + x(4);
c = x(1)*a;
F = y - c./b;
if nargout > 1 % compute the Jacobian
    d = b.^(-2);
    J = zeros(11,n);
    J(:,1) = -a./b;
    J(:,2) = -x(1)*u./b;
    J(:,3) = c.*d.*u;
    J(:,4) = c.*d;
end
end
```

```
function [ F, J ] = brown_dennis( x )
%% BROWN AND DENNIS FUNCTION for testing
%   unconstrained optimization software
%% AUTHOR: Stefan Scheer
%% REFERENCE:
% Test function (16) in
% "Testing unconstrained optimization software",
% by J.J. More, B.S. Garbow, and K.E. Hillstrom,
% ACM Trans. Math. Softw., vol.7, pp. 17-41, Mar. 1981.
%
%% INPUT:
% x - four-dimensional vector
%
%% OUTPUT:
% F - m-dimensional vector, where m >= n (default m = 20)
% J - mx4-dimensional Jacobian matrix
%
%% EXAMPLE:
% x = [25; 5; -5; -1];
% [F, J] = brown_dennis(x)

%% Choosing output dimension
m = 20;

%% Ensuring correct input
if isvector(x) ~= 1
    error('x must be a vector.')
end

%% Initialization
n = length(x);
% ensure that dimension of vector is correct
if n ~= 4
    error('Enter four-dimensional vector.')
end

%% Main procedure
t = (1:m)'/5;
expt = exp(t);
sint = sin(t);
cost = cos(t);
u = x(1) + t*x(2) - expt;
v = x(3) + x(4)*sint - cost;
F = u.^2 + v.^2;
if nargout > 1 % compute the Jacobian
    twou = 2*(x(1) + t*x(2) - expt);
    twov = 2*(x(3) + x(4)*sint - cost);
    J = zeros(m,n);
    J(:,1) = twou;
    J(:,2) = twou.*t;
    J(:,3) = twov;
    J(:,4) = twov.*sint;
end
end
```

```
function [ F, J ] = osborne1( x )
%% OSBORNE 1 FUNCTION for testing
%   unconstrained optimization software
%% AUTHOR: Stefan Scheer
%% REFERENCE:
% Test function (17) in
% "Testing unconstrained optimization software"
% by J.J. More, B.S. Garbow, and K.E. Hillstrom,
% ACM Trans. Math. Softw., vol.7, pp. 17-41, Mar. 1981.
%
%% INPUT:
% x - five-dimensional vector
%
%% OUTPUT:
% F - 33-dimensional vector
% J - 33x5-dimensional Jacobian matrix
%
%% EXAMPLE:
% x = [.5; 1.5; -1; .01; .02];
% [F, J] = osborne1(x)

%% Ensuring correct input
if isvector(x) ~= 1
    error('x must be a vector.')
end

%% Initialization
n = length(x);
% ensure that dimension of vector is correct
if n ~= 5
    error('Enter four-dimensional vector.')
end
y = [.844; .908; .932; .936; .925; .908; .881; .850; .818; .784; ...
    .751; .718; .685; .658; .628; .603; .580; .558; .538; .522; ...
    .506; .490; .478; .467; .457; .448; .438; .431; .424; .420; ...
    .414; .411; .406];
s = (1:33)';

%% Main procedure
t = 10*(s - 1);
e4 = exp(-t*x(4));
e5 = exp(-t*x(5));
u = x(2)*e4;
v = x(3)*e5;
F = x(1) + u + v - y;
if nargout > 1 % compute the Jacobian
    J = zeros(33,n);
    J(:,1) = 1;
    J(:,2) = e4;
    J(:,3) = e5;
    J(:,4) = -t.*u;
    J(:,5) = -t.*v;
end
end
```

```
function [ F, J ] = biggs( x )
%% BIGGS EXP6 FUNCTION for testing
%   unconstrained optimization software
%% AUTHOR: Stefan Scheer
%% REFERENCE:
% Test function (18) in
% "Testing unconstrained optimization software",
% by J.J. More, B.S. Garbow, and K.E. Hillstrom,
% ACM Trans. Math. Softw., vol.7, pp. 17-41, Mar. 1981.
%
%% INPUT:
% x - six-dimensional vector
%
%% OUTPUT:
% F - m-dimensional vector, where m >= n (default m = 13)
% J - mx6-dimensional Jacobian matrix
%
%% EXAMPLE:
% x = [1; 2; 1; 1; 1; 1];
% [F, J] = biggs(x)

%% Choosing output dimension
m = 13;

%% Ensuring correct input
if isvector(x) ~= 1
    error('x must be a vector.')
end

%% Initialization
n = length(x);
% ensure that dimension of vector is correct
if n ~= 6
    error('Enter six-dimensional vector.')
end

%% Main procedure
t = (1:m)'/10;
y = exp(-t) - 5*exp(-10*t) + 3*exp(-4*t);
e1 = exp(-t*x(1));
e2 = exp(-t*x(2));
e5 = exp(-t*x(5));
x3e1 = x(3)*e1;
x4e2 = x(4)*e2;
x6e5 = x(6)*e5;
F = x3e1 - x4e2 + x6e5 - y;
if nargout == 2 % compute the Jacobian
    J = zeros(m,n);
    J(:,1)  = -t.*x3e1;
    J(:,2)  = t.*x4e2;
    J(:,3)  = e1;
    J(:,4)  = -e2;
    J(:,5)  = -t.*x6e5;
    J(:,6)  = e5;
end
end
```

```
function [ F, J ] = osborne2( x )
%% OSBORNE 2 FUNCTION for testing
%   unconstrained optimization software
%% AUTHOR: Stefan Scheer
%% REFERENCE:
% Test function (19) in
% "Testing unconstrained optimization software"
% by J.J. More, B.S. Garbow, and K.E. Hillstrom,
% ACM Trans. Math. Softw., vol.7, pp. 17-41, Mar. 1981.
%
%% INPUT:
% x - 11-dimensional vector
%
%% OUTPUT:
% F - 65-dimensional vector
% J - 65x11-dimensional Jacobian matrix
%
%% EXAMPLE:
% x = [1.3; .65; .65; .7; .6; 3; 5; 7; 2; 4.5; 5.5];
% [F, J] = osborne2(x)

%% Ensuring correct input
if isvector(x) ~= 1
    error('x must be a vector.')
end
%% Initialization
n = length(x);
% ensure that dimension of vector is correct
if n ~= 11
    error('Enter 11-dimensional vector.')
end
y  = [1.366; 1.191; 1.112; 1.013; .991; .885; .831; .847; .786; ...
    .725; .746
        .679;  .608;  .655; .616;  .606; .602; .626; .651; .724; ...
            .649; .649
        .694;  .644;  .624; .661;  .612; .558; .533; .495; .500; ...
            .423; .395
        .375;  .372;  .391; .396;  .405; .428; .429; .523; .562; ...
            .607; .653
        .672;  .708;  .633; .668;  .645; .632; .591; .559; .597; ...
            .625; .739
        .710;  .729;  .720; .636;  .581; .428; .292; .162; .098; ...
            .054];
s = (1:65)';
%% Main procedure
t = (s - 1)/10;
u9 = t - x(9);
u10= t - x(10);
u11= t - x(11);
v9= u9.^2;
v10= u10.^2;
v11= u11.^2;
e1 = exp(-t*x(5));
e2 = exp(-v9*x(6));
e3 = exp(-v10*x(7));
e4 = exp(-v11*x(8));
F = x(1)*e1 + x(2)*e2 + x(3)*e3 + x(4)*e4 - y;
```

```
52 if nargout > 1 % compute the Jacobian
53     w2 = x(2)*e2;
54     w3 = x(3)*e3;
55     w4 = x(4)*e4;
56     J = zeros(65,n);
57     J(:,1) = e1;
58     J(:,2) = e2;
59     J(:,3) = e3;
60     J(:,4) = e4;
61     J(:,5) = -t*x(1).*e1;
62     J(:,6) = -v9.*w2;
63     J(:,7) = -v10.*w3;
64     J(:,8) = -v11.*w4;
65     J(:,9) = 2*u9*x(6).*w2;
66     J(:,10) = 2*u10*x(7).*w3;
67     J(:,11) = 2*u11*x(8).*w4;
68 end
69 end
```

```
1  function [ F, J ] = watson( x )
2  %% WATSON FUNCTION for testing
3  %   unconstrained optimization software
4  %% AUTHOR: Stefan Scheer
5  %% REFERENCE:
6  % Test function (20) in
7  % "Testing unconstrained optimization software"
8  % by J.J. More, B.S. Garbow, and K.E. Hillstrom,
9  % ACM Trans. Math. Softw., vol.7, pp. 17-41, Mar. 1981.
10 %
11 %% INPUT:
12 % x - n-dimensional vectorm, where 2 <= n <= 31
13 %
14 %% OUTPUT:
15 % F - 31-dimensional vector
16 % J - 31xn-dimensional Jacobian matrix
17 %
18 %% EXAMPLE:
19 % x = zeros(12,1);
20 % [F, J] = watson(x)
21
22 %% Ensuring correct input
23 if isvector(x) ~= 1
24     error('x must be a vector.')
25 end
26
27 %% Initialization
28 n = length(x);
29 % ensure that dimension of vector is correct
30 if n < 2 || n > 31
31     error('Dimension n of input vector must satisfy 2 <= n <= 31.')
32 end
33 sum1 = 0;
34 sum2 = 0;
35
36 %% Main procedure
37 t = (1:29)'/29;
38 sum2 = sum2 + x(1)*ones(29,1);
```

```
for j = 2:n
    sum1 = sum1 + (j-1)*x(j)*t.^(j-2);
    sum2 = sum2 + x(j)*(t.^(j-1));
end;
F =  sum1 - sum2.^2 - 1;
F(30) =  x(1);
F(31) =  x(2) - x(1)^2 - 1;
if nargout > 1 % compute the Jacobian
    J = zeros(31,n);
    twosum2 = 2*sum2;
    J(1:29,1) = -twosum2;
    for j = 2:n
        J(1:29,j) = (j-1)*(t.^(j-2)) - twosum2.*t.^(j-1);
    end;
    J(30,1) = 1;
    J(31,1) = -2*x(1);
    J(31,2) = 1;
end
end
```

```
function [ F, J ] = rosenbrockXT( x )
%% EXTENDED ROSENBROCK function for testing
%   unconstrained optimization software
%% AUTHOR: Stefan Scheer
%% REFERENCE:
% Test function (21) in
% "Testing unconstrained optimization software"
% by J.J. More, B.S. Garbow, and K.E. Hillstrom,
% ACM Trans. Math. Softw., vol.7, pp. 17-41, Mar. 1981.
%
%% INPUT:
% x - n-dimensional vector, n must be even
%
%% OUTPUT:
% F - n-dimensional vector
% J - nxn-dimensional Jacobian matrix
%
%% EXAMPLE:
% n = 16;
% x = zeros(n,1);
% maxit = n/2;
% for i = 1:maxit
%   x(2*i-1) = -1.2;
%   x(2*i) = 1;
% end
% [F, J] = rosenbrockXT(x)

%% Initialization
n = length(x); % input dimension
% check if dimension is even
if mod(n,2) ~= 0
    error('Input dimension n must be even.')
end
% determine the maximum number of iterations
maxit = n/2;
F = zeros(n,1);

```

```
%% Main procedure
for i = 1:maxit
    F(2*i-1) = 10*(x(2*i) - x(2*i-1)^2);
    F(2*i) = 1 - x(2*i-1);
end
if nargout > 1 % compute the Jacobian
    J = zeros(n,n);
    for i = 1:maxit
        J(2*i-1,2*i-1) = -20*x(2*i-1);
        J(2*i-1,2*i) = 10;
        J(2*i,2*i-1) = -1;
    end
end
end
```

```
function [ F, J ] = powell_singularXT( x )
%% EXTENDED POWELL SINGULAR function for testing
%   unconstrained optimization software
%% AUTHOR: Stefan Scheer
%% REFERENCE:
% Test function (22) in
% "Testing unconstrained optimization software"
% by J.J. More, B.S. Garbow, and K.E. Hillstrom,
% ACM Trans. Math. Softw., vol.7, pp. 17-41, Mar. 1981.
%
%% INPUT:
% x - n-dimensional vector, n must be a multiple of 4
%
%% OUTPUT:
% F - n-dimensional vector
% J - nxn-dimensional Jacobian matrix
%
%% EXAMPLE:
% n = 16;
% x = zeros(n,1);
% maxit = n/4;
% for i = 1:maxit
%   x(4*i-3) = 3;
%   x(4*i-2) = -1;
%   x(4*i-1) = 0;
%   x(4*i) = 1;
% end
% [F, J] = powell_singularXT(x)

%% Initialization
n = length(x);
% check if input dimension is a multiple of 4
if mod(n,4) ~= 0
    error('Input dimension n must be a multiple of 4.')
end
% determine the maximum number of iterations
maxit = n/4;
F = zeros(n,1);

%% Main procedure
sqrt5 = sqrt(5);
sqrt10 = sqrt(10);
```

```
for i = 1:maxit
    F(4*i-3) = x(4*i-3) + 10*x(4*i-2);
    F(4*i-2) = sqrt5*(x(4*i-1) - x(4*i));
    F(4*i-1) = (x(4*i-2) - 2*x(4*i-1))^2;
    F(4*i) = sqrt10*(x(4*i-3) - x(4*i))^2;
end
if nargout > 1 % compute the Jacobian
    J = zeros(n,n);
    for i = 1:maxit
        J(4*i-3,4*i-3) = 1;
        J(4*i-3,4*i-2) = 10;
        J(4*i-2,4*i-1) = sqrt5;
        J(4*i-2,4*i) = -sqrt5;
        J(4*i-1,4*i-2) = 2*(x(4*i-2) - 2*x(4*i-1));
        J(4*i-1,4*i-1) = -4*(x(4*i-2) - 2*x(4*i-1));
        J(4*i,4*i-3) = sqrt10*2*(x(4*i-3) - x(4*i));
        J(4*i,4*i) = sqrt10*(-2)*(x(4*i-3) - x(4*i));
    end
end
end
```

```
function [ F, J ] = penalty1( x )
%% PENALTY I function for testing
%   unconstrained optimization software
%% AUTHOR: Stefan Scheer
%% REFERENCE:
% Test function (23) in
% "Testing unconstrained optimization software"
% by J.J. More, B.S. Garbow, and K.E. Hillstrom,
% ACM Trans. Math. Softw., vol.7, pp. 17-41, Mar. 1981.
%
%% INPUT:
% x - n-dimensional vector
%
%% OUTPUT:
% F - (n+1)-dimensional vector
% J - (n+1)xn-dimensional Jacobian matrix
%
%% EXAMPLE:
% x = ones(10,1);
% [F, J] = penalty1(x)

%% Ensuring correct input
% ensure that vector is entered
if isvector(x) ~= 1
    error('x must be a vector.')
end
%% Initialization
n = length(x);
m = n+1;

%% Main procedure
a = sqrt(1e-5);
F = a*(x - 1);
F(m) = sum(x.^2) - (1/4);
if nargout > 1 % compute the Jacobian
    J = zeros(m,n);
```

```
    for j = 1:n
       J(1:n,j) = a;
    end;
    J(n+1,:) = 2*x';
end
end
```

```
function [ F, J ] = penalty2( x )
%% PENALTY II function for testing
%   unconstrained optimization software
%% AUTHOR: Stefan Scheer
%% REFERENCE:
% Test function (24) in
% "Testing unconstrained optimization software"
% by J.J. More, B.S. Garbow, and K.E. Hillstrom,
% ACM Trans. Math. Softw., vol.7, pp. 17-41, Mar. 1981.
%
%% INPUT:
% x - n-dimensional vector
%
%% OUTPUT:
% F - 2n-dimensional vector
% J - 2nxn-dimensional Jacobian matrix
%
%% EXAMPLE:
% x = ones(10,1)/2;
% [F, J] = penalty2(x)

%% Ensuring correct input
% ensure that vector is entered
if isvector(x) ~= 1
    error('x must be a vector.')
end
%% Initialization
n = length(x);
m = 2*n;
F = zeros(m,1);
t = (1:n)';
s = n-t+1;

%% Main procedure
a = sqrt(1e-5);
F(1)= x(1) - 0.2;
if n > 1
    nplus = n+1;
    mminus = m-1;
    onetenth = 1/10;
    xtenth = x/10;
    y = exp(t(2:n)/10) + exp((t(2:n)-1)/10);
    e1 = exp(xtenth(1:(n-1)));
    e2 = exp(xtenth(2:n));
    F(2:n) = a*(e2 + e1 - y);
    F(nplus:mminus) = a*(e2 - exp(-onetenth));
end
F(m)= sum(s.*x.^2) - 1;
if nargout > 1 % compute the Jacobian
    J = zeros(m,n);
```

```
    J(1,1)= 1;
    if n > 1
        b = a*onetenth;
        c = b*e1;
        d = b*e2;
        for j = 2:n
            J(j,j) = d(j-1);
            J(j,j-1) = c(j-1);
            J(nplus:mminus,j) = d;
        end;
    end
    J(m,:) = s'*2.*x';
end
```

```
function [ F, J ] = vardim( x )
%% VARIABLY DIMENSIONED function for testing
%   unconstrained optimization software
%% AUTHOR: Stefan Scheer
%% REFERENCE:
% Test function (25) in
% "Testing unconstrained optimization software"
% by J.J. More, B.S. Garbow, and K.E. Hillstrom,
% ACM Trans. Math. Softw., vol.7, pp. 17-41, Mar. 1981.
%
%% INPUT:
% x - n-dimensional vector
%
%% OUTPUT:
% F - (n+2)-dimensional vector
% J - (n+2)xn-dimensional Jacobian matrix
%
%% EXAMPLE:
% x = ones(10,1) - 1/10;
% [F, J] = vardim(x)

%% Ensuring correct input
% ensure that vector is entered
if isvector(x) ~= 1
    error('x must be a vector.')
end
%% Initialization
n = length(x);
nplus = n+1;
m = n+2;
t = (1:n)';

%% Main procedure
F = x - 1;
F(nplus) = sum(t.*(x-1));
F(m) = F(nplus)^2;
if nargout > 1 % compute the Jacobian
    J = eye(m,n); % df_i/dx_i = 1 for i = 1,...,n
    J(nplus,:) = t';
    J(m,:) = 2*F(nplus)*t';
end
end
```

```
function [ F, J ] = trigonometric( x )
%% TRIGONOMETRIC function for testing
%   unconstrained optimization software
%% AUTHOR: Stefan Scheer
%% REFERENCE:
% Test function (26) in
% "Testing unconstrained optimization software"
% by J.J. More, B.S. Garbow, and K.E. Hillstrom,
% ACM Trans. Math. Softw., vol.7, pp. 17-41, Mar. 1981.
%
%% INPUT:
% x - n-dimensional vector
%
%% OUTPUT:
% F - n-dimensional vector
% J - nxn-dimensional Jacobian matrix
%
%% EXAMPLE:
% n = 10;
% x = ones(n,1)/n;
% [F, J] = trigonometric(x)

%% Ensuring correct input
% ensure that vector is entered
if isvector(x) ~= 1
    error('x must be a vector.')
end

%% Initialization
n = length(x); % input dimension
t = (1:n)';

%% Main procedure
cosx = cos(x);
sinx = sin(x);
F = n - sum(cosx) + t.*(1 - cosx) - sinx;
if nargout > 1 % compute the Jacobian
    J = zeros(n,n);
    for j = 1:n
        J(:,j) = sinx(j)*ones(n,1);
        J(j,j) = sinx(j) + j*sinx(j) - cosx(j);
    end
end
end
```

```
function [ F, J ] = brownAL( x )
%% BROWN ALMOST-LINEAR function for testing
%   unconstrained optimization software
%% AUTHOR: Stefan Scheer
%% REFERENCE:
% Test function (27) in
% "Testing unconstrained optimization software"
% by J.J. More, B.S. Garbow, and K.E. Hillstrom,
% ACM Trans. Math. Softw., vol.7, pp. 17-41, Mar. 1981.
%
%% INPUT:
% x - n-dimensional vector
%
%% OUTPUT:
% F - n-dimensional vector
% J - nxn-dimensional Jacobian matrix
%
%% EXAMPLE:
% n = 10;
% x = ones(n,1)/2;
% [F, J] = brownAL(x)

%% Ensuring correct input
% ensure that vector is entered
if isvector(x) ~= 1
    error('x must be a vector.')
end

%% Initialization
n = length(x); % input dimension

%% Main procedure
F = x(1:(n-1)) + sum(x) - (n+1);
prodx = prod(x);
F(n) = prodx - 1;
if nargout > 1 % compute the Jacobian
    J = ones(n,n); % df_i/dx_i = 1 except for diagonal and last row
    diagonal = logical(eye(n));
    J(diagonal) = 2;
    J(n,:) = prodx./x';
end
end
```

```
function [ F, J ] = discreteBV( x )
%% DISCRETE BOUNDARY VALUE function for testing
%   unconstrained optimization software
%% AUTHOR: Stefan Scheer
%% REFERENCE:
% Test function (28) in
% "Testing unconstrained optimization software"
% by J.J. More, B.S. Garbow, and K.E. Hillstrom,
% ACM Trans. Math. Softw., vol.7, pp. 17-41, Mar. 1981.
%
%% INPUT:
% x - n-dimensional vector
%
```

```
%% OUTPUT:
% F - n-dimensional vector
% J - nxn-dimensional Jacobian matrix
%
%% EXAMPLE:
% n = 10;
% h = 1/(n+1);
% t = (1:n)'*h;
% x = t.*(t-1);
% [F, J] = discreteBV(x)

%% Ensuring correct input
% ensure that vector is entered
if isvector(x) ~= 1
    error('x must be a vector.')
end

%% Initialization
n = length(x); % input dimension

%% Main procedure
h = 1/(n+1);
t = (1:n)'*h;
F = 2*x - [0; x(1:(n-1))] - [x(2:n); 0] + h^2*((x + t + 1).^3)/2;
if nargout > 1 % compute the Jacobian (here a tridiagonal matrix)
    offdiag = -ones(n,1);
    maindiag = 2 + 3*h^2*((x + t + 1).^2)/2;
    tridiag = spdiags([offdiag maindiag offdiag], -1:1 ,n, n);
    J = full(tridiag);
end
end
```

```
function [ F, J ] = discreteIE( x )
%% DISCRETE INTEGRAL EQUATION function for testing
%   unconstrained optimization software
%% AUTHOR: Stefan Scheer
%% REFERENCE:
% Test function (29) in
% "Testing unconstrained optimization software"
% by J.J. More, B.S. Garbow, and K.E. Hillstrom,
% ACM Trans. Math. Softw., vol.7, pp. 17-41, Mar. 1981.
%
%% INPUT:
% x - n-dimensional vector
%
%% OUTPUT:
% F - n-dimensional vector
% J - nxn-dimensional Jacobian matrix
%
%% EXAMPLE:
% n = 10;
% h = 1/(n+1);
% t = (1:n)'*h;
% x = t.*(t-1);
% [F, J] = discreteIE(x)


```

```
%% Ensuring correct input
% ensure that vector is entered
if isvector(x) ~= 1
    error('x must be a vector.')
end

%% Initialization
n = length(x); % input dimension

%% Main procedure
h = 1/(n+1);
hhalf = h/2;
t = (1:n)'*h;
oneminust = 1-t;
u = x + t + 1;
ucubed = u.^3;
sum1 = zeros(n,1);
sum2 = zeros(n,1);
for i = 1:n
     sum1(i) = sum(t(1:i).*ucubed(1:i));
     sum2(i) = sum(oneminust(i+1:n).*ucubed(i+1:n));
end
F = x + hhalf*(oneminust.*sum1 + t.*sum2);
if nargout > 1 % compute the Jacobian
   J = zeros(n,n);
   temp1 = 3*hhalf*u.^2;
   for i = 1:n
       for j = 1:n
            temp2 = min(t(i),t(j)) - t(i)*t(j);
            J(i,j) = temp2*temp1(j);
       end
       J(i,i) = J(i,i) + 1;
   end
end
end
```

```
function [ F, J ] = broyden_tridiagonal( x )
%% BROYDEN TRIDIAGONAL function for testing
%   unconstrained optimization software
%% AUTHOR: Stefan Scheer
%% REFERENCE:
% Test function (30) in
% "Testing unconstrained optimization software"
% by J.J. More, B.S. Garbow, and K.E. Hillstrom,
% ACM Trans. Math. Softw., vol.7, pp. 17-41, Mar. 1981.
%
%% INPUT:
% x - n-dimensional vector
%
%% OUTPUT:
% F - n-dimensional vector
% J - nxn-dimensional Jacobian matrix
%
%% EXAMPLE:
% x = -ones(10,1);
% [F, J] = broyden_tridiagonal(x)

```

```
22 %% Ensuring correct input
23 % ensure that vector is entered
24 if isvector(x) ~= 1
25     error('x must be a vector.')
26 end
27
28 %% Initialization
29 n = length(x); % input dimension
30
31 %% Main procedure
32 F = (3 - 2*x).*x - [0; x(1:(n-1))] - 2*[x(2:n); 0] + 1;
33 if nargout > 1 % compute the Jacobian (here a tridiagonal matrix)
34     lowerdiag = -ones(n,1);
35     upperdiag = 2*lowerdiag;
36     maindiag = 3 - 4*x;
37     tridiag = spdiags([lowerdiag maindiag upperdiag], -1:1 ,n, n);
38     J = full(tridiag);
39 end
40 end
```

```
1  function [ F, J ] = broyden_banded( x )
2  %% BROYDEN BANDED function for testing
3  %   unconstrained optimization software
4  %% AUTHOR: Stefan Scheer
5  %% REFERENCE:
6  % Test function (31) in
7  % "Testing unconstrained optimization software"
8  % by J.J. More, B.S. Garbow, and K.E. Hillstrom,
9  % ACM Trans. Math. Softw., vol.7, pp. 17-41, Mar. 1981.
10 %
11 %% INPUT:
12 % x - n-dimensional vector
13 %
14 %% OUTPUT:
15 % F - n-dimensional vector
16 % J - nxn-dimensional Jacobian matrix
17 %
18 %% EXAMPLE:
19 % x = -ones(10,1);
20 % [F, J] = broyden_banded(x)
21
22 %% Ensuring correct input
23 % ensure that vector is entered
24 if isvector(x) ~= 1
25     error('x must be a vector.')
26 end
27
28 %% Initialization
29 n = length(x); % input dimension
30
31 %% Main procedure
32 ml = 5;
33 mu = 1;
34 sumJ = zeros(n,1);
35 for i=1:n
36     lb = max(1,i-ml);
37     ub = min(n,i+mu);
```

```
38      for j = 1:n
39          if j ~= i && lb <= j && j <= ub
40              sumJ(j) = sum(x.*(1 + x));
41          end
42      end
43  end
44  F = x.*(2 + 5*x.^2) + 1 - sumJ;
45  if nargout > 1 % compute the Jacobian
46      J = zeros(n,n);
47      for i = 1:n
48          lb = max(1,i-ml);
49          ub = min(n,i+mu);
50          for j = 1:n
51              if i == j
52                  J(i,j)= 2 + 15*x(i)^2;
53              elseif lb <= j && j <= ub
54              J(i,j) = -1 - 2*x(j);
55              end
56          end
57      end
58  end
59  end
```

```
1   function [ F, J ] = linear_fullrk( x )
2   %% LINEAR FULL RANK function for testing
3   %   unconstrained optimization software
4   %% AUTHOR: Stefan Scheer
5   %% REFERENCE:
6   % Test function (32) in
7   % "Testing unconstrained optimization software"
8   % by J.J. More, B.S. Garbow, and K.E. Hillstrom,
9   % ACM Trans. Math. Softw., vol.7, pp. 17-41, Mar. 1981.
10  %
11  %% INPUT:
12  % x - n-dimensional vector
13  %
14  %% OUTPUT:
15  % F - m-dimensional vector, where m => n (default m = 20)
16  % J - mxn-dimensional Jacobian matrix
17  %
18  %% EXAMPLE:
19  % x = ones(10,1);
20  % [F, J] = linear_fullrk(x)
21
22  %% Choosing output dimension
23  m = 20;
24
25  %% Ensuring correct input
26  % ensure that vector is entered
27  if isvector(x) ~= 1
28      error('x must be a vector.')
29  end
30
31  %% Initialization
32  n = length(x); % input dimension
33  F = zeros(m,1);
34
```

```
35 %% Main procedure
36 sumx = sum(x);
37 a = 2/m;
38 F(1:n) = x - a*sumx - 1;
39 if m > n
40 F(n+1:m) = -a*sumx - 1;
41 end
42 if nargout > 1 % compute the Jacobian
43     J = eye(m,n) - a*ones(m,n);
44 end
45 end
```

```
1  function [ F, J ] = linear_rk1( x )
2  %% LINEAR RANK 1 function for testing
3  %   unconstrained optimization software
4  %% AUTHOR: Stefan Scheer
5  %% REFERENCE:
6  % Test function (33) in
7  % "Testing unconstrained optimization software"
8  % by J.J. More, B.S. Garbow, and K.E. Hillstrom,
9  % ACM Trans. Math. Softw., vol.7, pp. 17-41, Mar. 1981.
10 %
11 %% INPUT:
12 % x - n-dimensional vector
13 %
14 %% OUTPUT:
15 % F - m-dimensional vector, where m => n (default m = 20)
16 % J - mxn-dimensional Jacobian matrix
17 %
18 %% EXAMPLE:
19 % x = ones(10,1);
20 % [F, J] = linear_rk1(x)
21
22 %% Choosing output dimension
23 m = 20;
24
25 %% Ensuring correct input
26 % ensure that vector is entered
27 if isvector(x) ~= 1
28     error('x must be a vector.')
29 end
30
31 %% Initialization
32 n = length(x); % input dimension
33
34 %% Main procedure
35 t = (1:m)';
36 sumjx = sum(t(1:n).*x);
37 F = t.*sumjx - 1;
38 if nargout > 1 % compute the Jacobian
39     J=zeros(m,n);
40     for j= 1:n
41         J(:,j) = t*j;
42     end
43 end
44 end
```

```
function [ F, J ] = linear_rk1zero( x )
%% LINEAR RANK 1 function WITH ZERO COLUMNS/ROWS for testing
%   unconstrained optimization software
%% AUTHOR: Stefan Scheer
%% REFERENCE:
% Test function (34) in
% "Testing unconstrained optimization software"
% by J.J. More, B.S. Garbow, and K.E. Hillstrom,
% ACM Trans. Math. Softw., vol.7, pp. 17-41, Mar. 1981.
%
%% INPUT:
% x - n-dimensional vector
%
%% OUTPUT:
% F - m-dimensional vector, where m => n (default m = 20)
% J - mxn-dimensional Jacobian matrix
%
%% EXAMPLE:
% x = ones(10,1);
% [F, J] = linear_rk1zero(x)

%% Choosing output dimension
m = 20;

%% Ensuring correct input
% ensure that vector is entered
if isvector(x) ~= 1
    error('x must be a vector.')
end
%% Initialization
n = length(x); % input dimension
F = zeros(m,1);

%% Main procedure
t = (2:m-1)';
tminusone = t - 1;
F(1) = -1;
F(m) = -1;
F(t) = tminusone*sum(t(1:n-2).*x(2:n-1)) - 1;
if nargout > 1 % compute the Jacobian
    J = zeros(m,n);
    for j = 2:n-1
        J(t,j) = tminusone*j;
    end
end
end
```

```
function [ F, J ] = chebyquad( x )
%% CHEBYQUAD function for testing
%   unconstrained optimization software
%% AUTHOR: Stefan Scheer
%% REFERENCE:
% Test function (35) in
% "Testing unconstrained optimization software"
% by J.J. More, B.S. Garbow, and K.E. Hillstrom,
% ACM Trans. Math. Softw., vol.7, pp. 17-41, Mar. 1981.
```

```
%% INPUT:
% x - n-dimensional vector
%
%% OUTPUT:
% F - m-dimensional vector, where m => n (default m = 9)
% J - mxn-dimensional Jacobian matrix
%
%% EXAMPLE:
% n = 9;
% x = ones(n,1)/(n+1);
% [F, J] = chebyquad(x)

%% Choosing output dimension
m = 9;

%% Initialization
n = length(x); % input dimension
F = zeros(m,1);
if nargout > 1 % compute the Jacobian
    J = zeros(m,n);
end

%% Main procedure
for j = 1:n      % The Chebyshev polynomials are obtained from the
    Tnminus = 1; % recurrence relation  T_0(x) = 1, % T_1(x) = x,
    Tn = x(j);   % T_{n+1}(x) = 2x T_n(x) - T_{n-1}(x).
    if nargout > 1
        dTnminusdx = 0;
        dTndx = 1;
    end
    for i = 1:m
        if nargout > 1
            J(i,j) = dTndx;
            dTnplusdx = 2*Tn + 2*dTndx*x(j) - dTnminusdx;
            dTnminusdx = dTndx;
            dTndx = dTnplusdx;
        end
        F(i) = F(i) + Tn;
        Tnplus = 2*x(j)*Tn - Tnminus;
        Tnminus = Tn;
        Tn = Tnplus;
    end
end
if nargout > 1
    J = J/n;
end
F = F/n;
for i = 1:m
    if mod(i,2) == 0
      F(i) = F(i) + 1/(i^2 - 1);
    end
end
end
```

References

[1] A. Bagirov, N. Karmitsa, and M. Mäkelä. *Introduction to Nonsmooth Optimization. Theory, Practice and Software*. Springer International Publishing, 2014.

[2] Y. Bard. *Nonlinear parameter estimation*. New York: Academic Press, 1974.

[3] A. E. Beaton. "The use of special matrix operators in statistical calculus". In: *ETS Research Bulletin Series* 1964.2 (1964), pp. i–222.

[4] V. Beiranvand, W. Hare, and Y. Lucet. "Best Practices for Comparing Optimization Algorithms". In: *Optimization and Engineering* 18 (Dec. 2017).

[5] T. A. Beu. *Introduction to Numerical Programming. A Practical Guide for Scientists and Engineers Using Python and C/C++*. Boca Raton: CRC Press, 2015.

[6] K. M. Brown and J. E. Dennis. "Derivative Free Analogues of the Levenberg-Marquardt and Gauss Algorithms for Nonlinear Least Squares Approximation". In: *Numer. Math.* 18 (1971), pp. 289–297.

[7] C. G. Broyden. "A Class of Methods for Solving Nonlinear Simultaneous Equations". In: *Mathematics of Computation* 19.92 (1965), pp. 577–593.

[8] J. E. Dennis and R. B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Classics in Applied Mathematics, 16. Philadelphia: SIAM, 1996.

[9] W. J. Dixon and M. B. Brown. *BMDP-77. Biomedical Computer Programs P-Series*. Los Angeles: University of California Press, 1977.

[10] E. Dolan and J. Moré. "Benchmarking Optimization Software with Performance Profiles". In: *Mathematical Programming* 91 (2001).

[11] R. Fletcher. *Practical Methods of Optimization*. Second. New York: Wiley-Interscience, 1987.

[12] R. Fletcher and C. M. Reeves. "Function minimization by conjugate gradients". In: *The Computer Journal* 7 (1964), pp. 149–154.

[13] J. E. Gentle. *Computational Statistics*. First. New York: Springer, 2009.

[14] P. E. Gill, W. Murray, and M. H. Wright. *Practical optimization*. London: Academic Press, 1981.

[15] G. H. Golub and C. F. Van Loan. *Matrix Analysis, Positive Definite Systems*. Fourth. Johns Hopkins University Press, 2013.

[16] J. H. Goodnight. "A Tutorial on the SWEEP Operator". In: *The American Statistician* 33.3 (1979), pp. 149–158.

[17] F. A. Graybill. *An introduction to linear statistical models*. Vol. 1. New York: McGraw-Hill, 1961.

[18] A. Griewank and A. Walther. "Introduction to Automatic Differentiation". In: *PAMM* 2.1 (2003), pp. 45–49.

[19] P. C. Hansen, V. Pereyra, and G. Scherer. *Least Squares Data Fitting with Applications*. Baltimore, Maryland: Johns Hopkins University Press, 2013.

[20] M. R. Hestenes and E. Stiefel. “Methods of conjugate gradients for solving linear systems”. In: *Journal of research of the National Bureau of Standards* 49 (1952), pp. 409–436.

[21] K. E. Hillstrom. “A Simulation Test Approach to the Evaluation of Nonlinear Optimization Algorithms”. In: *ACM Transactions on Mathematical Software* 3.4 (1977), pp. 305–315.

[22] R. I. Jennrich and P. F. Sampson. “Application of Stepwise Regression to Non-Linear Estimation”. In: *Technometrics* 10.1 (1968), pp. 63–72.

[23] M. Kimiaei, A. Neumaier, and B. Azmi. *LMBOPT–a limited memory method for bound-constrained optimization*. 2019. URL: `https://www.mat.univie.ac.at/~neum/ms/lmbopt.pdf` (visited on 07/23/2020).

[24] F. Knorn. *M-code LaTeX Package*. MATLAB Central File Exchange. 2021. URL: `https://www.mathworks.com/matlabcentral/fileexchange/8015-m-code-latex-package` (visited on 02/02/2021).

[25] A. Kuntsevich. *Moré set of test functions*. 1997. URL: `https://imsc.uni-graz.at/kuntsevich/solvopt/results/moreset.html` (visited on 12/17/2020).

[26] K. Lange. “Linear Regression and Matrix Inversion”. In: *Numerical Analysis for Statisticians*. Second. New York: Springer, 2010, pp. 93–111.

[27] K. Levenberg. “A Method for the Solution of Certain Non-Linear Problems in Least Squares”. In: *Quarterly of Applied Mathematics* 2 (1944), pp. 164–168.

[28] D. W. Marquardt. “An Algorithm for Least-Squares Estimation of Nonlinear Parameters”. In: *Journal of the Society for Industrial and Applied Mathematics* 11.2 (1963), pp. 431–441.

[29] The Mathworks. *DSP System Toolbox Reference*. Version R2020a. Natick, Massachusetts, 2020. URL: `https://mathworks.com/help/pdf_doc/dsp/dsp_ref.pdf` (visited on 12/06/2020).

[30] The Mathworks. *MATLAB Function Reference*. Version R2020a. Natick, Massachusetts, 2020. URL: `https://mathworks.com/help/pdf_doc/matlab/matlab_ref.pdf` (visited on 12/06/2020).

[31] J. Moré. *COPS: Large-Scale Optimization Problems*. Argonne National Laboratory. URL: `https://www.mcs.anl.gov/~more/cops/` (visited on 12/18/2020).

[32] J. J. Moré. “The Levenberg-Marquardt algorithm: Implementation and theory”. In: *Numerical Analysis*. Berlin, Heidelberg: Springer, 1978, pp. 105–116.

[33] J. J. Moré, B. S. Garbow, and K. E. Hillstrom. “Testing Unconstrained Optimization Software”. In: *ACM Transactions on Mathematical Software* 7.1 (1981), pp. 17–41.

[34] J. J. Moré and S. Wild. “Benchmarking Derivative-Free Optimization Algorithms”. In: *SIAM Journal on Optimization* 20 (Jan. 2009), pp. 172–191.

[35] J. A. Nelder and R. Mead. “A Simplex Method for Function Minimization”. In: *The Computer Journal* 7.4 (1965), pp. 308–313.

[36] A. Neumaier. “Complete search in continuous global optimization and constraint satisfaction”. In: *Acta Numerica* 13 (2004), pp. 271–369.

[37] A. Neumaier and B. Azmi. *Line search and convergence in bound-constrained optimization*. Tech. rep. Technical report, University of Vienna, 2019.

[38] A. Neumaier and H. Schichl. *Optimization - Theory and Algorithms*. Unpublished lecture notes. 2016.

[39] A. Neumaier et al. "VXQR: derivative-free unconstrained optimiziaton based on QR factorizations". In: *Soft Computing* 15 (2011), pp. 2287–2298.

[40] J. Nocedal and S. J. Wright. *Numerical Optimization*. Second. New York: Springer, 2006.

[41] J. Nocedal and Y. Yuan. "Combining Trust Region and Line Search Techniques". In: *Advances in Nonlinear Programming*. Ed. by Y. Yuan. Boston: Springer US, 1998, pp. 153–175.

[42] M. J. D. Powell. "A hybrid method for nonlinear equations". In: *Numerical Methods for Nonlinear Algebraic Equations*. Ed. by P. Rabinowitz. Gordon and Breach, 1970, pp. 87–114.

[43] W. H. Press et al. *Numerical Recipes. The Art of Scientific Computing*. Third. New York: Cambridge University Press, 2007.

[44] M. L. Ralston and R. I. Jennrich. "Dud, A Derivative-Free Algorithm for Nonlinear Least Squares". In: *Technometrics* 20.1 (1978), pp. 7–14.

[45] A. P. Ruszczyński. *Nonlinear Optimization*. Princeton, New Jersey: Princeton University Press, 2006.

[46] G. A. F. Seber and A. J. Lee. *Linear Regression Analysis*. Second. Wiley Series in Probability and Statistics. Hoboken, New Jersey: Wiley, 2003.

[47] G. A. F. Seber and C. J. Wild. *Nonlinear Regression*. Wiley Series in Probability and Statistics. Hoboken, New Jersey: Wiley, 2003.

[48] J. F. Traub. "A Theorem on the Solutions of Certain Inhomogeneous Difference Equations". In: *Iterative Methods for the Solution of Equations*. Englewood Cliffs, New Jersey: Prentice-Hall, 1964, p. 41.

[49] L. N. Trefethen and D. Bau. *Numerical Linear Algebra*. Philadelphia: SIAM, 1997.

[50] J. H. Wilkinson and C. Reinsch. *Linear Algebra. Handbook for Automatic Computation*. Vol. 2. Berlin, Heidelberg: Springer, 1971.