The Levenberg-Marquardt Method and its Implementation in Python

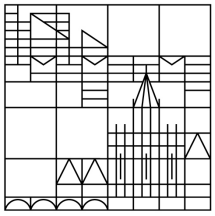
Master Thesis

submitted by

Marius Kaltenbach

at the

Universität Konstanz



Department of Mathematics and Statistics

Konstanz, December 15, 2022

Supervisor and Reviewer: Prof. Dr. Stefan Volkwein, University of Konstanz

Abstract

The Levenberg-Marquardt method is a widely used algorithm applied to solve non-linear least-squares problems. In this thesis, the Levenberg-Marquardt method is implemented in Python as a trust-region approach based on the works of Moré [Mor78] and Nocedal and Wright [NW06]. After a detailed description of the method, the Python code is presented. The Levenberg-Marquardt method is then tested on seven least-squares problems, including small-residual and large-residual problems as well as a poorly scaled one. In particular, the combination of large residuals and poor scaling makes the method slow, as it needs a large number of iterations to converge. Afterwards, the test results are compared to line search methods, such as the Gauss-Newton method. Regarding the test problems, the Levenberg-Marquardt method performed as the most robust algorithm.

Zusammenfassung

Die Levenberg-Marquardt Methode ist ein häufig eingesetztes Verfahren zum Lösen von nichtlinearen Kleinste-Quadrate Problemen. In dieser Arbeit wird die Levenberg-Marquardt Methode aufbauend auf den Arbeiten von Moré [Mor78] und Nocedal und Wright [NW06] als Trust-Region Verfahren in Python implementiert. Nach einer detaillierten Beschreibung der Methode wird der Python Code vorgestellt. Anschließend wird die Levenberg-Marquardt Methode an sieben Problemen getestet, diese beinhalten Probleme mit kleinen und großen Residuen sowie ein Beispiel mit problematischer Skalierungseigenschaft. Besonders die Kombination aus großen Residuen und einer schwierigen Skalierung führt dazu, dass die Methode langsam wird bzw. mehr Iterationen benötigt werden. Abschließend werden die Testergebnisse mit Line Search Methoden, wie der Gauss-Newton Methode verglichen. In Bezug auf die Testprobleme erwies sich die Levenberg-Marquardt Methode als der robusteste Algorithmus.

Contents

1 Introduction

> All models are wrong, but some are useful.
>
> George E. P. Box

Models are typically used in many scientific disciplines to describe complex phenomena [HL19, p. 4]. However, according to the quote above, all models are wrong. This does not come as a surprise, considering that models simplify the real world by focusing on things that are regarded as important to the phenomena, and omitting the rest. As a consequence, models are more predictable but there is usually an unexplained gap between the model and the observed data. To make a given model useful it is a common task to adapt its parameters such that the discrepancy is minimized. A widely used approach to measure the gap is by squaring each error between a data point and its predicted model point and finally sum it up [BF10, p. 3]. The error is usually refered as the residual. If we have m data points and n parameters to estimate, then the problem translates to finding the minimizer of the following function:

$$f(x) = \frac{1}{2}\sum_{j=1}^{m} r_j^2(x), \tag{1}$$

where each $r_j : \mathbb{R}^n \to \mathbb{R}$ stands for a residual. It is further assumed that every r_j is a smooth function and that $m \geq n$ holds, which means that there are more data points than adjustable parameters in the model function. The problem in mimimizing the function (1) is better known as the *least-squares problem* and is the content of this thesis.

When the model function is linear in its parameters x then the least-squares problem becomes especially easy to solve. However, if a nonlinear function is chosen as a model, then the problem becomes more elaborate. This thesis builds upon the master thesis by Albicker [Alb22] who implemented the Gauss-Newton method in order to solve *nonlinear least-squares problems* (1). As described in there, the Gauss-Newton method uses a line search approch to find the minimizer. In the following, the Levenberg-Marquardt method is introduced which can be described as the trust-region counterpart of the Gauss-Newton method.

Originally, the algorithm traces its roots to the influential work of Levenberg [Lev44] and Marquardt [Mar63]. A popular variation of the algorithm which is based on a trust-region strategy is due to Moré [Mor78]. This thesis follows the approach of Moré combined with the theoretical results from the book of Nocedal and Wright [NW06, pp. 258-262].
It is noteworthy that the presented algorithm is relatively old. For example the paper from Moré was published in 1978, the initial publication by Levenberg is even from the year 1944. Nevertheless, the relevance and application of the Levenberg-Marquardt method is still important, which can be illustrated that the paper from Moré peaked with 588 citations in the year 2021 [Sch22].

Before the actual algorithm is presented, some necessary fundamental concepts related to least-squares problems are discussed in Chapter 2. Then, in Chapter 3, the Levenberg-Marquardt method is presented followed directly by its implementation in Python in Chapter 4. The algorithm is also tested here on seven problems and finally, it is compared

to line search methods, like implemented by Albicker [Alb22] in the fifth chapter.
Concerning the notation, $\|\cdot\|$ is used to represent the Euclidian norm or its induced matrix norm, the spectral norm.

2 Fundamentals

> Listen to your teachers, but
> cheat in calculus.
>
> Macklemore, from the song
> Growing up

In this section the special structure of *least-squares problems* (1) is analyzed. What makes them unique is predominantly that there is an inexpensive way to calculate an approximation for the Hessian of the objective function f. This is done by "cheating", in other words by omitting an, often neglectable, part of the Hessian matrix. The resulting approximation is then an important ingredient of the algorithm in Chapter 3. Afterwards linear least-squares problems are examined, in which cases the approximation becomes exact. Finally, this chapter concludes with arguments for the usage of the ℓ_2-norm as a measure of discrepancy, instead of other potential norms. This chapter is largely based on the book from Nocedal and Wright [NW06, pp. 245-254].

2.1 Derivatives of the Objective Function

As a first step, the objective function f from (1) can be simplified by combining all residuals r_j into one residual vector. Define r: $\mathbb{R}^n \to \mathbb{R}^m$ with

$$r(x) := (r_1(x), r_2(x), \ldots, r_m(x))^T.$$

Then f can be written as:

$$f(x) = \frac{1}{2}\sum_{j=1}^{m} r_j^2(x) = \frac{1}{2} r(x)^T r(x) = \frac{1}{2}\|r(x)\|^2.$$

Accordingly, the Jacobian matrix of the residual vector $r(\cdot)$ is an $m \times n$ matrix of the following form:

$$J(x) = \begin{pmatrix} \nabla r_1(x)^T \\ \nabla r_2(x)^T \\ \vdots \\ \nabla r_m(x)^T \end{pmatrix},$$

where the abbreviation $\nabla r_j(x)$ is used as the gradient of $r_j(x)$ for every $j \in \{1, \ldots, m\}$. Note that $\nabla r_j(x)$ is treated as a column vector here.
Furthermore, the gradient and Hessian of the objective function f can be directly computed:

- The gradient can be obtained by using the chain rule:

$$\nabla f(x) = \sum_{j=1}^{m} r_j(x) \cdot \nabla r_j(x) = J(x)^T r(x). \tag{2}$$

- After applying the product rule, the Hessian matrix is obtained:

$$\begin{aligned} \nabla^2 f(x) &= \sum_{j=1}^{m} \nabla r_j(x) \cdot \nabla r_j(x)^T + \sum_{j=1}^{m} r_j(x) \nabla^2 r_j(x) \\ &= J(x)^T J(x) + \sum_{j=1}^{m} r_j(x) \nabla^2 r_j(x). \end{aligned} \tag{3}$$

A closer look now reveals that in order to obtain $\nabla f(x)$, the main task is to calculate $J(x)$. According to Nocedal and Wright [NW06, p. 246] the calculation of $J(x)$ is relatively inexpensive for many applications.
The special feature of least-squares problems is that $J(x)$ can also be used to calculate the first term of the Hessian of f in (3). This can be done without the often expensive evaluations of $\nabla^2 r_j(x)$. As a result we get a cheap approximation for the Hessian by its first term:

$$\nabla^2 f(x) \approx J(x)^T J(x). \tag{4}$$

This approximation is especially useful when the second term in (3) is neglectable, which is the case when $r_j(x)$ or $\nabla^2 r_j(x)$ are sufficiently small for every $j \in \{1, \ldots, m\}$. In other words, when the residuals are small or if the residuals are approximately affine near the solution. An advantage of this approximation of the Hessian lies in the fact that it is always positive semidefinite, because for an arbitrary $y \in \mathbb{R}^n$ it is:

$$y^T (J(x)^T J(x)) y = (J(x)y)^T (J(x)y) = \|J(x)y\|^2 \geq 0.$$

Before these considerations are used to built the Levenberg-Marquardt algorithm, a few remarks concerning the statistical relevance of the ℓ_2-norm and linear least-squares problems in general are given.

2.2 A Linear Least-Squares Problem

As mentioned in Chapter 1, squared residuals can be used to observe the fit between a postulated model and the observed data. The minimization task (1) becomes particularly simple when a linear function is used as a model. This procedure will now be deepened by solving (1) for a general linear least-squares problem in combination with a simple example.
When a linear model function is chosen, the corresponding residual vector has the following form:

$$r(x) = Jx - y, \qquad \text{where } J \in \mathbb{R}^{m \times n} \text{ and } y \in \mathbb{R}^m. \tag{5}$$

Thus, the objective function is given by:

$$f(x) = \frac{1}{2}\|Jx - y\|^2. \tag{6}$$

Since the residual vector is also an affine-linear function, the approximation of the Hessian used in (4) is exact.
By applying (2) and (3), we have then:

$$\nabla f(x) = J^T(Jx - y), \qquad \nabla^2 f(x) = J^T J.$$

Thus, the Hessian of f is positive semidefinit and therefore the function is convex on its domain in the linear case. Consequently, a minimizer is the vector x^* that solves

$$\nabla f(x^*) = 0,$$

which is equivalent to:

$$J^T J x^* = J^T y. \tag{7}$$

This system of linear equations is better known as the *normal equations* for (6).
With (7) it is easy to argue that there is only one unique solution vector x^* when $J^T J$ is

nonsingular. This is exactly the case when J has full rank n.
There is not one method that best computes the solution of (6) in all cases, rather there are several methods, which have advantages and disadvantages in different situations. For a concise overview, see [NW06, pp. 251-254].
For the following Levenberg-Marquardt method, the choice will fall on an approach based on a QR decomposition of J. Therefore, this procedure is now presented in more detail.
A QR decomposition of J with column pivoting gives the following matrices:

$$J\Pi = Q \begin{bmatrix} R \\ 0 \end{bmatrix} = [Q_1 \quad Q_2] \begin{bmatrix} R \\ 0 \end{bmatrix} = Q_1 R \tag{8}$$

where Π is an $n \times n$ permutation matrix and Q is an $m \times m$ orthogonal matrix. Furthermore, Q_1 is defined as the first n colums of Q, and Q_2 as its last $(m-n)$ colums, R is an $n \times n$ upper triangular matrix with positive diagonal elements.
With the QR decomposition of J it is already possible to compute a minimizer of f. This is the result of the following lemma.

Lemma 2.1. *If J has full rank n, then the unique minimizer of f defined in* (6) *is given by*

$$x^* = \Pi R^{-1} Q_1^T y,$$

where Π, R and Q_1 are defined as above.

Proof. Following [NW06, p. 252], it is:

$$\begin{aligned} \|Jx - y\|^2 &= \left\| \begin{bmatrix} Q_1^T \\ Q_2^T \end{bmatrix} (J\Pi\Pi^T x - y) \right\|^2 \\ &= \left\| \begin{bmatrix} R \\ 0 \end{bmatrix} (\Pi^T x) - \begin{bmatrix} Q_1^T y \\ Q_2^T y \end{bmatrix} \right\|^2 \\ &= \|R(\Pi^T x) - Q_1^T y\|^2 + \|Q_2^T y\|^2 \end{aligned}$$

The first equation is true, because an orthogonal matrix does not affect the Euclidian norm and for the permutation matrix Π it holds $\Pi\Pi^T = I$. In the second equation the result of the QR decomposition is applied.
Because $\|Q_2^T y\|^2$ does not depend on x, $\|Jx - y\|^2$ is minimized, if $R(\Pi^T x) = Q_1^T y$. Consequently we have a minimum at x^* for

$$x^* = \Pi R^{-1} Q_1^T y.$$

□

If J has no full rank $r < n$, then R is singular and there is not a unique solution to the linear least-squares problem in (6). Instead there is an infinite number of solutions. The next lemma provides one possible solution, it is based on [GMW21, pp. 227–229].

Lemma 2.2. *If J has rank $r < n$, then a minimizer of f defined in* (6) *is given by*

$$x^* = \Pi z,$$

where

$$z = \begin{bmatrix} z_B \\ 0 \end{bmatrix} \text{ and } z_B \text{ is defined by } R_1 z_B = (Q_1^T y)_r.$$

Π and Q_1 are defined as above and R_1 represents the first r columns of R and $(Q_1^T y)_r$ the first r elements of the vector $Q_1^T y$.

Proof. It can be argued analogously like in the proof of Lemma 2.1, that $\|Jx - y\|^2$ is minimized, if $R(\Pi^T x) = Q_1^T y$. Because R is singular there is an infinite number of solutions and one of those can be computed like described above [GMW21, p. 228]. □

It is important to note that we used column pivoting in the calculation of the QR decomposition of J in (8). This step is necessary in order to obtain (8) if J has no full rank. In the following Levenberg-Marquardt method, we will have to solve a linear least-squares problem too. Accordingly, we will use also column pivoting so that we can obtain (8) for both cases, whether the respective matrix J has full rank or not.

With this, all the building blocks are already together to solve a given linear least-squares problem. Linear least-squares problems are widely applied (see for an overview for linear models in statistics for example Bingham and Fry [BF10]).
To illustrate the procedure, an example from the social sciences is used. The example and the data are taken from the book [LL15, p. 10]. In Figure 1 a scatterplot is presented which illustrates the bivariate relationship between education in years and the income per year in dollars. The sample is based on 32 participants. For every participant j there is a point (t_j, y_j) in the scatterplot in Figure 1, where t_j denotes the years of education and y_j the income in dollars.

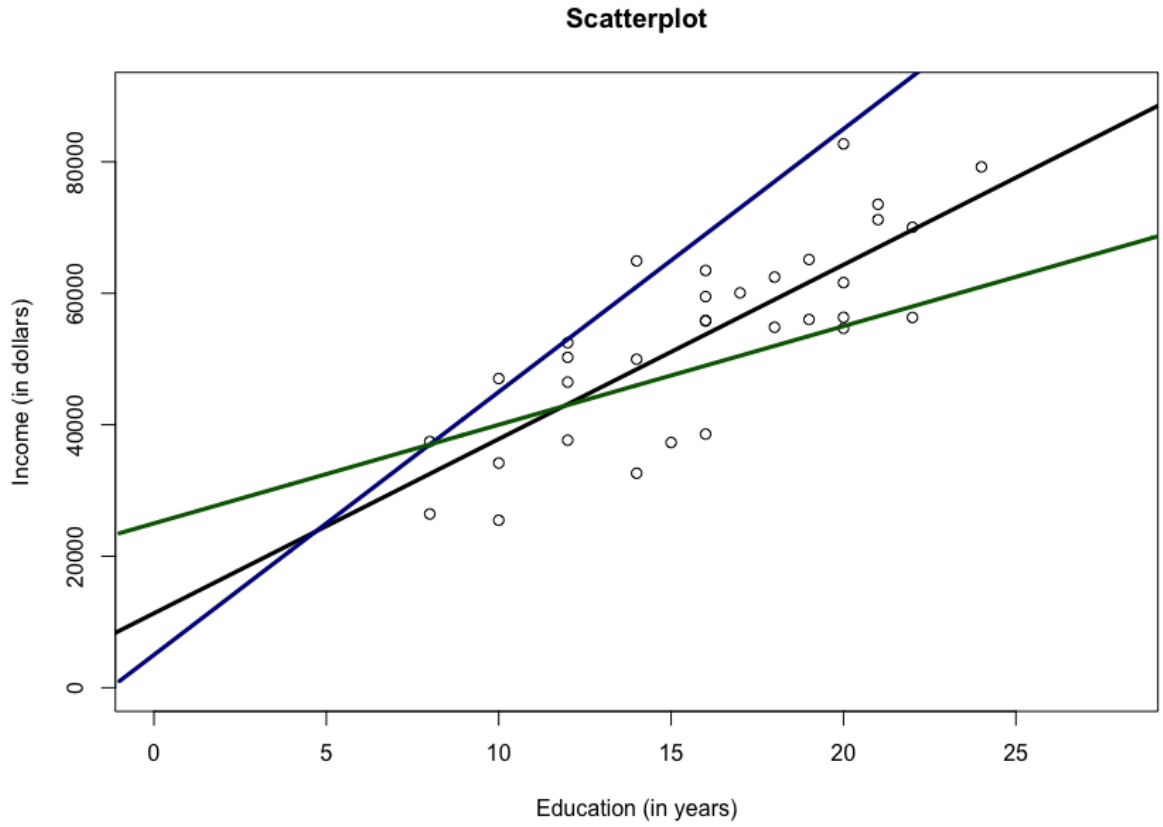


Figure 1: Relationship between education and income.

In this example, theoretical considerations lead to the use of a linear model function. In other words, it is assumed that the income of a respective person can be modeled as an affine-linear function of the given years of education:

$$\phi(x;t) = x_1 + tx_2.$$

The goal is then to find the parameter vector $x = (x_1, x_2)^T$ such that the discrepancy between the actual data and the predicted model values is minimized. Graphically, the problem is to choose the function graph that best fits the data. Therefore, three example functions are added to the scatterplot in Figure 1 to illustrate the problem.
A common approach to measuring the discrepancy, and thus deciding which parameters are better, is to use the sum of the squared residuals. The question of why this approach is particulary useful will be discussed in more detail in the next section.

To summarise, the according problem which has to be solved is given by:

$$\min_{x\in\mathbb{R}^2} \frac{1}{2}\sum_{j=1}^{32}[\phi(x;t_j) - y_j]^2 = \frac{1}{2}\|r(x)\|^2,$$

where $r(x) := (\phi(x;t_1) - y_1, \dots, \phi(x;t_{32}) - y_{32})^T$.
In order to use matrix notation, we can write the residual vector $r(x)$ in the form (5) by defining:

$$J = \begin{pmatrix} 1 & t_1 \\ \vdots & \vdots \\ 1 & t_{32} \end{pmatrix} \in \mathbb{R}^{32\times 2}.$$

In fact, it is $r(x) = Jx - y$, where $y = (y_1, \dots, y_{32})^T$ is the vector which stores the income in dollar for each participant.
The application of Lemma 2.1 gives the parameter vector $x^* \approx (11.321, 2.651)^T$ as the solution for our model. In the scatterplot shown in Figure 1, this result is ilustrated by the black function graph.

At this point, neither the meaning of this result nor the limitations of this bivariate model are discussed. More information about this can be found in the book by Lewis-Beck and Lewis-Beck [LL15, pp. 8-14], where this example was taken from. The goal of this chapter was rather to simply illustrate the general procedure on how to construct a least-squares problem. This will be now followed by a discussion of the question on which norm should be used to measure the discrepancy between the model and the data.

2.3 Measurement of Discrepancy

To begin, it is not clear how the length of the residuals must be measured. In the aforementioned example the sum of squares produced a useful result. However, there are other reasonable measures available. Nocedal and Wright [NW06, p. 249] mention alternatively the maximum absolute value and the sum of absolute values to measure the discrepancy between the data and the model. In the previous example this would result in the following:

$$\max_{j=1,2,\dots,32} |\phi(x;t_j) - y_j| \qquad \text{and} \qquad \sum_{j=1}^{32} |\phi(x;t_j) - y_j|.$$

Consequently, the objective function f, which must be minimized, is given for the corresponding cases by the following:

$$f(x) = \|r(x)\|_\infty \qquad \text{and} \qquad f(x) = \|r(x)\|_1. \tag{9}$$

There is no objective criteria that dictates that a certain norm must always be chosen. Nevertheless, the usage of the ℓ_2-norm has several positive features. First of all, the objective function f is already a smooth function. In contrast, the functions f in (9) are not differentiable in the classical sense. There are also differences in dealing with very large residuals. Compared to the ℓ_1-norm it values large residuals higher, whereby the maximum norm only takes the biggest residual into account.
In statistical applications there is an important connection to the calculation of the maximum likelihood estimate. In fact, if the residuals are independent, identically distributed and follow a normal distribution, the minimizer of the sum-of-squares function is identical to the maximum likelihood estimate [NW06, p. 250].

Suppose that a set of observations $\{(t_j, y_j) \,|\, j = 1, \ldots, m\}$ is given. Furthermore, a model function $\phi(x; t)$ is chosen such that the remaining task is to identify the appropriate parameter vector $x = (x_1, \ldots, x_n)^T$. The objective function, which has to be minimized, is then given by:

$$f(x) = \frac{1}{2} \sum_{j=1}^{m} \left[\phi(x, t_j) - y_j\right]^2. \tag{10}$$

According to Nocedal and Wright [NW06, p. 249], the notation of the residuals should be slightly altered within a statistical application, instead of r_j we use ε_j:

$$\varepsilon_j = \phi(x; t_j) - y_j \qquad \text{for } j \in \{1, \ldots, m\}.$$

The concerned ε_j are assumed to be indepenedent and identically distributed. Furthermore, they follow a normal distribution with an arbitrary variance σ^2 and a mean $\mu = 0$. The condition that $\mu = 0$ is important because it makes sure that y_j is not inflected by any systematic bias. Thus it is $\varepsilon_j \sim \mathcal{N}(0, \sigma^2)$ for every $j \in \{1, \ldots, m\}$.
As a consequence, the probability density function $g_\sigma(\varepsilon)$, which is the same for each ε_j, can be already stated:

$$g_\sigma(\varepsilon) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp(-\frac{\varepsilon^2}{2\sigma^2}). \tag{11}$$

The general idea behind a maximum likelihood approach is to find the parameter vector x for which the likelihood that the obsevations y_j have occured given x is the true parameter, is maximized. See for a more detailed description of this approach the introductory book by Hogg, McKean and Craig [HCM19].
Because of the independence of the residuals, we directly know the joint distribution of the ε_j and can then state the likelihood function as:

$$p(y; x) = \prod_{j=1}^{m} g_\sigma(\varepsilon_j) = \prod_{j=1}^{m} g_\sigma(\phi(x; t_j) - y_i).$$

Now (11) can be inserted instead of g_σ, thus it follows after a few rearrangements:

$$p(y; x) = \frac{1}{(2\pi\sigma^2)^{\frac{m}{2}}} \exp\left(-\frac{1}{2\sigma^2} \sum_{j=1}^{m} [\phi(x; t_j) - y_i]^2\right). \tag{12}$$

As a direct result of (12) it is clear that the minimizer x of (10) is also the minimizer of $p(y; x)$.

3 The Levenberg-Marquardt Method

> Play the opening like a book, the middle game like a magician, and the endgame like a machine.
>
> Rudolf Spielmann

The quote at the beginning of this chapter is from the Austrian chess player Rudolf Spielmann and describes quite well the structure of the following chapter about the derivation of the Levenberg-Marquardt method. More precisely, the first section describes the idea and the general structure of the algorithm. This part will be played like a book, because it refers to a standard trust-region approach, which results in a first, general algorithm. Then, to find the minimum of our problem (1), the special properties are exploited. This is, where the "magic" comes into play, a few tricks which simplify a lot due to the approximation of the Hessian matrix by a positive semidefinite matrix. As a result the so-called "hard case" is avoided [CGT00, p.180]. The final section is concerned with convergence properties and improvements, such that the algorithm will work well on a (computer) machine.

3.1 Idea

The Levenberg-Marquardt method presented here is based on the paper by Moré [Mor78] in combination with the book of Nocedal and Wright [NW06] and uses a trust-region approach. In general, the goal is to provide an iterative scheme to find a minimizer for the nonlinear least-squares problem in (1). The main idea behind a trust-region approach is to focus on a region around the current iterate x_k, for which the objective function f can be approximated by an easier, often quadratic, model function m_k. The next iterate can then be calculated by $x_{k+1} = x_k + p$, such that it is the minimizer of m_k and also lies within the trust-region.
In order that the new iterate results in a sufficient decrease, the trust-region has to be chosen appropriately. In the following section, these two topics, the model function and the choice of an appropriate trust-region, are discussed. Finally this results in a first, general algorithm.

3.1.1 Model Function m_k

To obtain a model function m_k, which approximates the objective function f, a linearization of the residual vector around the current iterate x_k can be used. It is:

$$r(x_k + p) \approx r(x_k) + r'(x_k)p = r(x_k) + J(x_k)p.$$

Substitution results in the model function:

$$m_k(p) = \frac{1}{2}\|J(x_k)p + r(x_k)\|^2. \tag{13}$$

Obviously, this linearization can only be trusted for a p whose norm is small enough. Therefore it is important to define an appropriate trust-region for which the approximation

can be trusted. Before doing so, a closer look on the model function m_k reveals an interesting connection to the approximation of the Hessian matrix of f in (4). Written out, the model function (13) is:

$$m_k(p) = f(x_k) + J(x_k)^T r(x_k)p + \frac{1}{2}p^T J(x_k)^T J(x_k)p. \tag{14}$$

In fact, this looks quite similar to the second-order Taylor-series expansion of f around the point x_k:

$$f(x_k + p) = f(x_k) + \nabla f(x_k)^T p + \frac{1}{2}p^T \nabla^2 f(x_k + tp)p \qquad \text{for some } t \in (0, 1). \tag{15}$$

The only difference between the model function (14) and (15) is the approximation of the Hessian $\nabla^2 f(x_k + tp) \approx J(x_k)^T J(x_k)$. As stated in the previous section, the usage of this nonexpensive approximation is a specialty for our present problem. The approximation is especially appropriate when we deal with a small-residual problem, such that the neglected second-order term in the Hessian (3) is small.

3.1.2 Trust-Region

The easiest way to constrain the step length is to consider only vectors p such that $\|p\| \leq \Delta_k$ holds for a given constant $\Delta_k > 0$. As a result we have a spherical trust-region, and the following subproblem has to be solved, in order to get a new iterate $x_{k+1} = x_k + p_k$:

$$\min_p m_k(p) \qquad \text{such that } \|p\| \leq \Delta_k.$$

The size of the trust-region, or similarly the value of its radius Δ_k is crucial for the success of the algorithm [NW06, p.67]. On the one hand, if it is too large, then the model function may not be an appropriate approximation anymore and thus, the function value may not even be reduced at the new iterate x_{k+1}. On the other hand, if Δ_k is chosen too small, then only small steps are allowed, although larger steps would be reasonable and would save computational effort.
Therefore ρ_k, a measure of fit between the model function and the objective function, is introduced. In the following algorithm this value can then be used to update the radius Δ_k accordingly. It is

$$\rho_k = \frac{f(x_k) - f(x_k + p_k)}{m_k(0) - m_k(p_k)}. \tag{16}$$

The numerator in the above definition is named the *actual reduction* in f by the step p_k. The denominator is the reduction in the model function, which should predict the reduction in f well. Therefore the denominator is usually called the *predicted reduction*. On the one hand, if $\rho_k \approx 1$, then the model function predicts approximately the actual reduction. Thus, the model function m_k fits well to the objective function f and it is reasonable to expand the trust-region. On the other hand, if ρ_k is close to zero than the model function is rather not a good approximation within the trust region and Δ_k must be shrunk. The choice of the exact thresholds to guide the decision in the algorithm is somewhat arbitrary. Nocedal and Wright [NW06, p. 69] shrink the trust-region whenever $\rho_k < \frac{1}{4}$ and expand it whenever $\rho_k > \frac{3}{4}$, this approach will be followed here.
It is important to mention, that the predicted reduction is nonnegative, because p_k is chosen as the minimizer of m_k within a region where $p = 0$ is included. Thus $\rho_k < 0$ is equivalent to $f(x_k + p_k) > f(x_k)$ and the step p_k does not result in an improvement. It

is therefore more efficient to set $\rho_k = 0$, if the case $f(x_k + p_k) > f(x_k)$ occurs and shrink the trust-region respectively.
Note, that ρ_k might be bigger than 1, which means, that the actual reduction is bigger than the predicted reduction. Because we get a good next iteration step, the model function is also seen as being suitable and the trust-region radius Δ_k can be increased as well.

With further knowledge about the minimizer p_k of the subproblem, a more robust calculation method can be applied, as discussed in Section 3.5.2. For now the general trust-region algorithm can be given in Algorithm 1, which is based on [NW06, p. 69].

Algorithm 1 General Trust-Region Algorithm

Given f, $\Delta_{max} > 0, \Delta_0 \in (0, \Delta_{max}), \eta \in [0, \frac{1}{4})$;
for $k = 0, 1, 2, \ldots$ **do**
Obtain p_k by solving the subproblem:

$$\min_p \frac{1}{2}\|J(x_k)p + r(x_k)\|^2 \qquad \text{subject to } \|p\| \leq \Delta_k$$

Calculate ρ_k

if $\rho_k < \frac{1}{4}$ **then**
$\Delta_{k+1} = \frac{1}{4}\Delta_k$
else
if $\rho_k > \frac{3}{4}$ and $\|p_k\| = \Delta_k$ **then**
$\Delta_{k+1} = \min(2\Delta_k, \hat{\Delta})$
else
$\Delta_{k+1} = \Delta_k$
end if
end if

if $\rho_k > \eta$ **then**
$x_{k+1} = x_k + p_k$
else
$x_{k+1} = x_k$
end if

end for

In the algorithm, a few new features are introduced. These are now briefly discussed:

- a maximum trust-region radius $\Delta_{max} > 0$ is used to bound the step length.
- $\eta \in [0, \frac{1}{4})$ is the threshold which determines whether a step p_k is accecpted.
- An additional condition has to be fulfilled in order to increase the trust-region. More precisely, the radius is only increased when the constraint becomes active. The idea behind this is that otherwise it can be assumed that an increased trust-region might not lead to an advantage [NW06, p. 69].
- Previously, it was only stated to increase or decrease Δ_k, but not how this is done exactly. The easiest way to decrease is to multiply Δ_k by a positive factor less

than one. Similarly, to increase Δ_k, a factor bigger than one is used. According to Nocedal and Wright [NW06, p.69] the factors $\frac{1}{4}$ and 2 are used.

Obviously, Algorithm 1 needs a few updates in order to be useful. For example a suitable stopping criteria has to be added. However, the big question remains as to how to get the minimizer of the subproblem. The answer is presented in the next section.

3.2 Solving the Subproblem

The missing link in the previous algorithm is to solve the subproblem in order to obtain the step p_k. A fundamental characterization of p_k is given in the next theorem. For ease of reading, this section drops the index k and also uses two abbreviations in reference to the model function m_k.
The goal is to solve the following general subproblem:

$$\min_{p\in\mathbb{R}^n} m(p) = f + g^T p + \frac{1}{2} p^T B p \qquad \text{such that } \|p\| \leq \Delta \tag{17}$$

where $f = f(x)$, $g = \nabla f(x)$, and B is a symmetric matrix, which in our special case is $B = J^T J$.
As it turns out the minimizer p^* of (17) fulfills the following equation:

$$(B + \lambda I)p^* = -g$$

where $\lambda \geq 0$ is a Lagrange multiplier associated with the constraint $\|p\| \leq \Delta$. This is the content of the following fundamental theorem, as presented in [NW06, p. 70].

Theorem 3.1. *The vector p^* is a global solution of the trust-region subproblem* (17) *if and only if p^* is feasible and there is a scalar $\lambda \geq 0$ such that the following conditions are satisfied:*

$$\begin{aligned} (B + \lambda I)p^* &= -g, \\ \lambda(\Delta - ||p^*||) &= 0, \\ (B + \lambda I) \text{ is positive semidefinite.} & \end{aligned}$$

Proof. A proof is presented in [NW06, pp. 89-91] □

Note that λ is the Lagrange multiplier for our trust-region constraint. The characterization provided in Theorem 3.1 will play a central role for the calculation of the minimizer of our subproblem. As a first step it can be applied to a least-squares problem (1), in which case B is substituted by $J^T J$.

Corollary 3.2. *The vector p^{LM} is a global solution of the trust-region subproblem*

$$\min_p \|Jp + r\|^2 \qquad \textit{subject to } \|p\| \leq \Delta \tag{19}$$

if and only if p^{LM} is feasible (i.e. $\|p^{LM}\| \leq \Delta$) and there is a scalar $\lambda \geq 0$ such that

$$(J^T J + \lambda I)p^{LM} = -J^T r, \tag{20a}$$

$$\lambda(\Delta - \|p^{LM}\|) = 0. \tag{20b}$$

Proof. This is a direct consequence of Theorem 3.1 for the case $B = J^T J$ [NW06, p. 262]. □

Corollary 3.2 has some important practical implications. In fact, there are two possible situations depending on whether the step p^{GN}, which solves

$$J^T J p^{GN} = -J^T r \tag{21}$$

lies within the trust-region or not. If J has full rank than the solution p^{GN} is unique and is usually called the *Gauss-Newton step*.

- Case 1: $p^{GN} \leq \Delta$.
  Then p^{GN} is already our solution p^{LM} of the subproblem (19) and $\lambda = 0$.
- Case 2: $p^{GN} > \Delta$.
  Then $\lambda = 0$ cannot be chosen. This is, because it would imply that $p^{LM} = p^{GN}$ and thus p^{LM} is not feasible as well. Consequently, there has to be found a $\lambda > 0$ such that p^{LM} solves equation (20a) and because of (20b) it must hold $\|p^{LM}\| = \Delta$.

When J has no full rank, then $J^T J$ is singular and there is not *the* Gauss-Newton step p^{GN}, but an infinite number of possible solutions to (21). In the following we will show how to calculate one possible solution. We will treat then this solution as the Gauss-Newton step p^{GN} so that the above distinction of cases is applicable in the same way.

Whenever we are in the first case, it is relatively easy to calculate the step p^{LM}. The only difficulty is to solve equation (21). However, in the second case an appropriate $\lambda > 0$ has to be found. These issues will be addressed in the next sections. First the question about how equation (20a) can be practically solved for any given $\lambda \geq 0$ is addressed. Afterwards it is focussed on the calculation of λ for the second case.

3.2.1 Solution of a Linear Least-Squares Problem

An essential step in order to solve our subproblem (19) is to solve for p:

$$(J^T J + \lambda I)p = -J^T r, \tag{22}$$

for a given $\lambda \geq 0$. These equations can be recognized as the normal equations of the following linear least-squares problem:

$$\min_p \frac{1}{2} \left\| \begin{bmatrix} J \\ \sqrt{\lambda} I \end{bmatrix} p + \begin{bmatrix} r \\ 0 \end{bmatrix} \right\|^2. \tag{23}$$

Consequently, it is possible to solve for p within the normal equations (22), or to solve the linear-least squares problem (23) and obtain p in this way.
Both approaches have advantages and disadvantages. According to Moré [Mor78, p. 107] the first one can be twice as fast as the latter approach. However, the latter one is usually chosen because it is more robust, especially as the calculation of the matrix product $J^T J$ can be avoided. Furthermore, when J is nearly rank deficient and $\lambda = 0$ then (22) is unreliable.
According to these considerations, one has to weigh whether to use a faster algorithm or a more robust and reliable one. Usually, the advantages of robustness and reliability are

valued higher and as a consequence (23) is chosen for the further procedure [Mor78, p. 107; NW06, p. 259].

In the following implementation of the Levenberg-Marquardt method, (23) has often to be solved several times for varying λ. It will be started by calculating the Gauss-Newton step for $\lambda = 0$ and checking whether it already fullfils the trust-region constraint. If not, further computational effort is needed to solve (23) again, this time for some $\lambda > 0$, as long as $\|p\| = \Delta$ holds.
In Chapter 2 it was already shown how to solve a linear least-squares problem using a QR decomposition. In general, the procedure presented there can be used directly to solve the problem (23). But in order to make multiple calculations of (23) for varying λ more efficient, a slightly altered scheme is presented which consists of two parts. The special feature is that only the second part must be recalculated for changing λ. The following procedure is due to Moré [Mor78, pp. 107-108].

Part 1
The first step consists of the calculation of a QR decomposition of $J \in \mathbb{R}^{m\times n}$. Householder transformations can be applied to get the following decomposition of J. As discussed in Section 2.2 column pivoting is used because J might be rank deficient. It is

$$J\Pi = Q \begin{bmatrix} R \\ 0 \end{bmatrix} = [Q_1 \quad Q_2] \begin{bmatrix} R \\ 0 \end{bmatrix} = Q_1 R, \tag{24}$$

where Π is an $n \times n$ permutation matrix, Q is an $m \times m$ orthogonal matrix. Furthermore, Q_1 is defined as the first n colums of Q, and Q_2 as its last $(m-n)$ colums, R is an $n \times n$ upper triangular matrix with positive diagonal elements.
For $\lambda = 0$ a solution p of the problem (23) can be calculated directly. It depends whether J has full rank n or not. Accordingly, either Lemma 2.1 or Lemma 2.2 can be applied to obtain a solution p.

Part 2
When $\lambda > 0$ is given, then the QR decomposition in Part 1 is not sufficient to calculate the solution of (23), but it is still a useful tool for the further process. Note that this time there exists a unique p, because $J^T J + \lambda I$ is always positive definite and thus invertible. The main procedure in order to get the step p is to calculate another QR decomposition, but this time for $\begin{bmatrix} J \\ \sqrt{\lambda} I \end{bmatrix}$.
Before doing so, the first QR decomposition (24) implies that the following statement is true:

$$\begin{bmatrix} R \\ 0 \\ \sqrt{\lambda} I \end{bmatrix} = \begin{bmatrix} Q^T & 0 \\ 0 & \Pi^T \end{bmatrix} \begin{bmatrix} J \\ \sqrt{\lambda} I \end{bmatrix} \Pi. \tag{25}$$

The matrix on the left-hand side in the equation above is, except for the n diagonal elements within $\sqrt{\lambda} I$, already upper triangular. A series of Givens rotations can be used to eliminate these n diagonal elements in order to rotate the left matrix to an upper triangular matrix. It can be proceeded, as sketched in [NW06, p. 260], in the following way:

- eliminate the (n, n) element of $\sqrt{\lambda} I$ by rotating row n of R with row n of $\sqrt{\lambda} I$.

- eliminate the $(n-1, n-1)$ element of $\sqrt{\lambda}I$ by rotating row $n-1$ of R with row $n-1$ of $\sqrt{\lambda}I$. By this procedure position $(n-1, n)$ of $\sqrt{\lambda}I$ is filled. This position can be eliminated by rotating row n of R with row $n-1$ of $\sqrt{\lambda}I$.
- eliminate the $(n-2, n-2)$ element of $\sqrt{\lambda}I$ by rotating row $n-2$ of R with row $n-2$ of $\sqrt{\lambda}I$. By this procedure position $(n-2, n-1)$ and $(n-2, n)$ of $\sqrt{\lambda}I$ are filled. This position can be eliminated by rotating ...

In summation, $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ Givens rotations are executed. These can all be gathered in the matrix $\bar{Q}_\lambda$ which results in the following statement:

$$\bar{Q}_\lambda^T \begin{bmatrix} R \\ 0 \\ \sqrt{\lambda}I \end{bmatrix} = \begin{bmatrix} R_\lambda \\ 0 \\ 0 \end{bmatrix}. \tag{26}$$

Finally Q_λ can be defined as

$$Q_\lambda = \begin{bmatrix} Q & 0 \\ 0 & \Pi \end{bmatrix} \bar{Q}_\lambda. \tag{27}$$

Thus the required QR decomposition is achieved:

$$\begin{bmatrix} J \\ \sqrt{\lambda}I \end{bmatrix} \Pi = Q_\lambda \begin{bmatrix} R_\lambda \\ 0 \end{bmatrix} = [Q_{\lambda 1} \quad Q_{\lambda 2}] \begin{bmatrix} R_\lambda \\ 0 \end{bmatrix} = Q_{\lambda 1} R_\lambda, \tag{28}$$

where Π is still an $n \times n$ permutation matrix, Q_λ is an $(n+m) \times (n+m)$ orthogonal matrix, $Q_{\lambda 1}$ is defined as the first n colums of Q_λ, and $Q_{\lambda 2}$ as its last $(m-n)$ colums, R_λ is an $n \times n$ upper triangular matrix with positive diagonal elements.
By applying Lemma 2.1, we get for a given $\lambda > 0$ the solution of (23) by:

$$p = \Pi R_\lambda^{-1} \left(-Q_{\lambda 1}^T \begin{bmatrix} r \\ 0 \end{bmatrix} \right). \tag{29}$$

Note that in practice, the direct calculation of the inverse of R_λ should be avoided. Therefore, first a triangular subsitution can be processed to get z in $R_\lambda z = -Q_{\lambda 1}^T \begin{bmatrix} r \\ 0 \end{bmatrix}$. Then the components of z must be permuted to finally get $p = \Pi z$.
Besides, with the QR decomposition (28) it follows also:

$$\Pi^T (J^T J + \lambda I) \Pi = R_\lambda^T R_\lambda, \tag{30}$$

which will be useful in the following section.

In sum, the advantage of this approach is that Part 1 and especially the QR decomposition of J has to be calculated only once for changing λ. According to Nocedal and Wright [NW06, p. 260] this is particularly useful when $m \gg n$.

3.2.2 The Levenberg-Marquardt Parameter λ

Whenever the Gauss-Newton step p^{GN} lies outside the trust-region, a $\lambda > 0$ has to be found, such that the equations (20a, 20b) are fulfilled. The solution of (20a) can be written as a function of λ, such that it is:

$$p(\lambda) = -(J^T J + \lambda I)^{-1} g. \tag{31}$$

Note that for $\lambda > 0$, the matrix $(J^T J + \lambda)$ is positive definite and thus its inverse exists such that the function is well-defined for every $\lambda > 0$. The solution $p(0)$ can be defined in this way only if $J^T J$ is positive definit.
The second condition (20b) can then be rewritten as:

$$\|p(\lambda)\| = \Delta,$$

or equivalently to find the root of

$$\phi(\lambda) = \|p(\lambda)\| - \Delta. \tag{32}$$

Thus, the problem which has to be solved to obtain λ can be characterized as a root-finding problem in one dimension [NW06, p. 84].

Before an iterative algorithm is presentend on how to find the root of (32), it will be first shown that a unique λ exists, whenever the solution p^{GN} of (22) for $\lambda = 0$ does not fulfill the trust-region constraint.

Lemma 3.3. *Suppose that for the solution p^{GN} of (22) for $\lambda = 0$ it holds $\|p^{GN}\| > \Delta$. Then there is a unique $\lambda > 0$ such that for $p(\lambda)$ defined in (31), it holds*

$$\|p(\lambda)\| = \Delta.$$

Proof. In the following proof, the ideas of Nocedal and Wright [NW06, pp. 84-85] are followed and adapted to our subproblem (19).
In a first step $J^T J$ can be diagonalized because it is a symmetric matrix. Thus, there is an orthonormal matrix Q and a diagonal matrix $\Lambda = \operatorname{diag}(\lambda_1, \lambda_2, \ldots, \lambda_n)$ which contains the eigenvalues $0 \leq \lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$ of $J^T J$. The resulting eigendecomposition $J^T J = Q \Lambda Q^T$ can also be used to simplify the matrix $J^T J + \lambda I$. It is:

$$J^T J + \lambda I = Q \Lambda Q^T + \lambda Q Q^T = Q(\Lambda + \lambda I) Q^T. \tag{33}$$

The new expression in (33) for $J^T J + \lambda I$ can then be substituted in $p(\lambda)$. Furthermore, it can be reformulated to obtain

$$p(\lambda) = -Q(\Lambda + \lambda I)^{-1} Q^T g = -\sum_{j=1}^{n} \frac{q_j^T g}{\lambda_j + \lambda} q_j,$$

where q_j represents the j-th column of Q.
As a consequence, that Q was chosen to be an orthonormal matrix, it follows further:

$$\|p(\lambda)\|^2 = \sum_{j=1}^{n} \frac{(q_j^T g)^2}{(\lambda_j + \lambda)^2}. \tag{34}$$

If $J^T J$ is positive definite, then the case $\lambda = -\lambda_j$ cannot occur for all $j \in \{1, \ldots, n\}$ when $\lambda \geq 0$, because all eigenvalues are positive.
However, if $J^T J$ is only positive semidefinite and not postive definite, then the case $\lambda = -\lambda_j$ cannot occur for all $j \in \{1, \ldots, n\}$ when $\lambda > 0$. But there is at least one $\lambda_j = 0$, which implies that for $\lambda = 0$ the function $\|p(\cdot)\|$ has a pole and it is $\lim_{\lambda \to 0} \|p(\lambda)\| = \infty$.
With the help of the following three considerations, it can be argued that there is exactly one appropriate λ which fullfils $\|p(\lambda)\| = \Delta$:

- If $p(0)$ is uniquely defined, then $p(0) = p^{GN}$ and it is $\|p(0)\| > \Delta$.
  If $p(0)$ is not defined, then J^TJ is not positive definite. Thus, a positive λ close to zero can be chosen, such that $\|p(\lambda)\| > \Delta$.
- If $\lambda > 0$ then, because J^TJ is positive semidefinite, it is $\lambda > -\lambda_j$ and so the denominator $\lambda_j + \lambda > 0$ is increasing for every $j \in \{1, \ldots, n\}$ with regards to λ.
  Thus $\|p(\lambda)\|$ is a strictly decreasing function for $\lambda \in (0, \infty)$.
- Furthermore, it is $\lim_{\lambda \to \infty} \|p(\lambda)\| = 0$.

Since $\lambda \to \|p(\lambda)\|$ is a continuous function, it follows that there is a unique $\lambda > 0$, for which $\|p(\lambda)\| = \Delta$ holds. □

The remaining task is to find this unique root $\lambda \in (0, \infty)$ of ϕ.
For that purpose, an iterative algorithm is introduced. A first idea would be to use Newton's method to find the root of ϕ. As Nocedal and Wright [NW06, p. 86] point out, this approach has a disadvantage, because ϕ is highly nonlinear close to $-\lambda_1$:
For a λ close but greater than $-\lambda_1$, ϕ can be approximated due to (34) by:

$$\phi(\lambda) \approx \frac{C_1}{\lambda + \lambda_1} + C_2,$$

where $C_1 > 0$ and C_2 are constants. As a result, the approximation and thus ϕ are highly nonlinear. This implies that Newton's method will not perform well in our situation [NW06, p. 86].

Alternatively, Hebden [Heb73] presents an iterative scheme $\{\lambda_k\}_{k \in \mathbb{N}}$, which takes the nonlinearity of ϕ into account.
Here, the main idea is to aproximate ϕ by a rational function $\tilde{\phi}(\lambda)$ with constant numerator and linear denominator around the current iterate λ_k [Heb73, pp. 8-9].
Because for $\|p(\lambda)\|$ the expression in (34) is true, it can be argued that it is reasonable to define an approximation of ϕ by $\tilde{\phi}(\lambda)$ in the following way:

$$\tilde{\phi}(\lambda) = \frac{a}{b + \lambda} - \Delta \tag{35}$$

where $a, b \in \mathbb{R}$ are chosen as constants such that $\tilde{\phi}(\lambda_k) = \phi(\lambda_k)$ and $\tilde{\phi}'(\lambda_k) = \phi'(\lambda_k)$. As shown in the next lemma, it is easy to calculate the root of $\tilde{\phi}(\lambda)$. In the iterative scheme, this root is then chosen as the next iterate λ_{k+1} [Mor78, p. 110].

Lemma 3.4. *It is* $\tilde{\phi}(\lambda_{k+1}) = 0$ *if*

$$\lambda_{k+1} = \lambda_k - \left(\frac{\phi(\lambda_k) + \Delta}{\Delta}\right) \cdot \left(\frac{\phi(\lambda_k)}{\phi'(\lambda_k)}\right) \tag{36}$$

Proof. The proof is based on Hebden [Heb73, p. 9].
Define $\tilde{\phi}_1$ as the first part of $\tilde{\phi}$:

$$\tilde{\phi}_1(\lambda) = \frac{a}{b + \lambda}. \tag{37}$$

Consequently, the derivative is:

$$\tilde{\phi}_1'(\lambda) = \frac{-a}{(b + \lambda)^2}.$$

Thus, $\frac{1}{b+\lambda}$ is given by:

$$\frac{1}{b+\lambda} = -\frac{\tilde{\phi}_1'(\lambda)}{\tilde{\phi}_1(\lambda)}. \tag{38}$$

Finally, the approximation can be used to calculate the step s, such that $\tilde{\phi}_1(\lambda_{k+1}) = \Delta$. Therefore it has to be solved for $\lambda_{k+1} = \lambda_k + s$ that

$$\frac{a}{b + (\lambda_k + s)} = \Delta,$$

which is, after some rearrangements, equivalent to

$$s = \frac{a - \Delta(b + \lambda_k)}{\Delta}.$$

Now, (37) can be used to substitute a:

$$s = \frac{\tilde{\phi}_1(\lambda_k) - \Delta}{\Delta} \cdot (b + \lambda_k).$$

Again, substitution of $(b + \lambda_k)$ through (38) results in:

$$s = \frac{\tilde{\phi}_1(\lambda_k) - \Delta}{\Delta} \cdot \left(\frac{-\tilde{\phi}_1(\lambda_k)}{\tilde{\phi}_1'(\lambda_k)} \right). \tag{39}$$

We have that $\phi(\lambda_k) = \tilde{\phi}_1(\lambda_k) - \Delta$ and $\tilde{\phi}_1'(\lambda_k) = \phi'(\lambda_k)$. Inserted in (39), this results in

$$s = \frac{\phi(\lambda_k)}{\Delta} \cdot \left(\frac{-(\phi(\lambda_k) + \Delta)}{\phi'(\lambda_k)} \right),$$

which gives us already the statement of this lemma:

$$\lambda_{k+1} = \lambda_k - \left(\frac{\phi(\lambda_k) + \Delta}{\Delta} \right) \left(\frac{\phi(\lambda_k)}{\phi'(\lambda_k)} \right).$$

□

It is important to note that the iteration must only contain $\lambda_k > 0$, because the desired λ is also positive and $\phi(\lambda_k)$ might not be defined for $\lambda_k < 0$. To ensure this, safeguards are added to the iteration. Before doing so, it is first shown how $\phi'(\lambda_k)$ can be evaluated for a given iterate $\lambda_k > 0$.
Building on Hebden [Heb73, p. 8] and adapted to our situation, it is:

$$\begin{aligned} \|p(\lambda)\| &= ((p(\lambda))^T p(\lambda))^{\frac{1}{2}} \\ &= ((-(J^T J + \lambda I)^{-1} g)^T (-(J^T J + \lambda I)^{-1} g))^{\frac{1}{2}} \\ &= (g^T (J^T J + \lambda I)^{-2} g)^{\frac{1}{2}}. \end{aligned}$$

and thus for $\lambda > 0$ it follows:

$$\begin{aligned} \phi'(\lambda) = \frac{\partial \|p(\lambda)\|}{\partial \lambda} &= \frac{1}{2\|p(\lambda)\|} \cdot (-2 g^T (J^T J + \lambda I)^{-3} g) \\ &= \frac{1}{\|p(\lambda)\|} \cdot (-g^T (J^T J + \lambda I)^{-3} g). \end{aligned} \tag{40}$$

Note that in order that $\phi'(\lambda)$ exisits, it has to be true that $\|p(\lambda)\| \neq 0$. Because of the statement in (34) this holds for all $\lambda > 0$, as long as $g \neq 0$. The situation that the gradient of f is the zero vector at the current iterate x_k will not occur, because in this case we already would have found a stationary point and no further step p is needed to be calculated. With (40) it is also clear that $\phi'(0)$ only exists if J^TJ is positive definite.

To get an inexpensive calculation of $\phi'(\lambda_k)$, it is necessary to avoid the calculation of $(J^TJ + \lambda_k I)^{-1}$. Fortunately, Moré provides a handy way to deal with this difficulty in our situation [Mor78, p. 111]. In the following, we drop the index k and simply use λ as our current iterate.
As a first step we can calculate $p(\lambda)$ like described in (29). During the computation, we obtain with (30) a new expression:

$$(J^TJ + \lambda I) = \Pi R_\lambda^T R_\lambda \Pi^T. \tag{41}$$

Note, if the special case $\Pi = I$ occurs, (41) would be the Cholesky decomposition of $J^TJ + \lambda I$.
The next step in order to calculate $\phi'(\lambda)$ is to introduce a vector q. For brevity p is used as an abbreviation of $p(\lambda)$ in the following.
Define q such that it is the solution of $\Pi R_\lambda^T q = p$. Thus, q is obtained inexpensively by a permutation and a triangular substitution. In particular, it holds after the substitution of p:

$$\begin{aligned} q &= (R_\lambda^{-T})\Pi^T p \\ &= (R_\lambda^{-T})\Pi^T(\Pi(R_\lambda^T R_\lambda)^{-1}\Pi^T(-g)) \\ &= (R_\lambda^{-T})(R_\lambda^T R_\lambda)^{-1}\Pi^T(-g). \end{aligned} \tag{42}$$

Finally, it can be stated, that

$$\begin{aligned} \phi'(\lambda) &= \frac{1}{\|p\|} \cdot (-g^T(J^TJ + \lambda I)^{-3} g) \\ &= \frac{1}{\|p\|} \cdot (-g^T(\Pi R_\lambda^T R_\lambda \Pi^T)^{-3} g) \\ &= \frac{1}{\|p\|} \cdot (-g^T \Pi (R_\lambda^T R_\lambda)^{-3} \Pi^T g) \\ &= \frac{1}{\|p\|} \cdot (-g^T \Pi (R_\lambda^T R_\lambda)^{-1}(R_\lambda^T R_\lambda)^{-1}(R_\lambda^T R_\lambda)^{-1} \Pi^T g) \\ &= \frac{1}{\|p\|} \cdot (-g^T \Pi (R_\lambda^{-1}(R_\lambda^{-T})R_\lambda^{-1})((R_\lambda^{-T})R_\lambda^{-1}(R_\lambda^{-T})) \Pi^T g) \\ &= \frac{1}{\|p\|} \cdot (-(R_\lambda^{-T})(R_\lambda^T R_\lambda)^{-1}\Pi^T g)^T((R_\lambda^{-T})(R_\lambda^T R_\lambda)^{-1}\Pi^T) g). \end{aligned}$$

Now (42) can be used for another substitution, it follows:

$$\begin{aligned} \phi'(\lambda) &= \frac{1}{\|p\|} \cdot (-q^T q) \\ &= -\frac{\|q\|^2}{\|p\|}. \end{aligned} \tag{43}$$

To summarise, it is now possible to calculate λ_{k+1} as defined in (36) for the iteration process. However, it still has to be ensured that for all iterates k it holds, that $\lambda_k > 0$ is

true. In the iteration process, it might be the case that this condition is not met and the next iterate cannot be calculated, which would imply that the iteration process breaks down.

In order to avoid this problem and exclude $\lambda_k \leq 0$ for every iterate k, safeguards are introduced, as described in Moré [Mor78, pp. 110-111]. This is done with an interval of uncertainty $I \subset \mathbb{R}_{>0}$, in which the true λ is known to lie within. Whenever $\lambda_k \notin I$ occurs, it is altered to lie in the interval.
The following Lemma 3.5 gives the boundaries to create such an interval of uncertainty in which every iterate λ_k is forced to lie. The introduction of the upper boundary is due to Hebden [Heb73, p. 13], in addition the lower boundary is mentioned in Moré [Mor78, p. 110].

Lemma 3.5. *Suppose that for the Gauss-Newton step p^{GN} it holds $\|p^{GN}\| > \Delta$.*
Then the unique root $\lambda > 0$ of ϕ, defined in (32) lies in the interval $(l_0, u_0]$, where the upper and lower boundaries are defined as follows:

$$u_0 = \frac{\|J^T r\|}{\Delta},$$

and if $J^T J$ has full rank:

$$l_0 = -\frac{\phi(0)}{\phi'(0)},$$

otherwise set $l_0 = 0$.

Proof. The proof of the upper boundary was given in [Heb73, p. 13] and is in the following adapted to our subproblem.
Upper boundary u_0:

$$\begin{aligned} \|p(\lambda)\| &= \|(J^T J + \lambda I)^{-1} J^T r\| \\ &\leq \|(J^T J + \lambda I)^{-1}\| \|J^T r\| \\ &\leq \lambda^{-1} \|J^T r\| \\ &< \Delta \qquad \text{for } \lambda > \frac{\|J^T r\|}{\Delta} \end{aligned}$$

Note the inequality in the third line holds, because $J^T J$ is positive semidefinite, so the eigenvalues of $(J^T J + \lambda I)$ are bounded by λ from below. Consequently, the eigenvalues $(J^T J + \lambda I)^{-1}$ are bounded by λ^{-1} from above.
Because $\lambda \to \|p(\lambda)\|$ is strictly decreasing, we know that for $\|p(\lambda)\| = \Delta$ it must hold $\lambda \leq \|J^T r\|/\Delta$.
Lower boundary l_0:
Case 1: J has full rank. As a consequence $J^T J$ is invertible and thus positive definite. In this case $\phi'(0)$ exists and can be calculated by (40).
Furthermore, ϕ is a strictly convex function for $\lambda \in [0, \infty)$ and $g \neq 0$, because

$$\begin{aligned} \phi''(\lambda) &= \frac{\mathrm{d}}{\mathrm{d}\lambda} \left(\frac{1}{\|p(\lambda)\|} \cdot (-g^T (J^T J + \lambda I)^{-3} g) \right) \\ &= \left(\frac{(\|p(\lambda)\|)'}{\|p(\lambda)\|^2} \right) \cdot (-g^T (J^T J + \lambda I)^{-3} g) + \frac{1}{\|p(\lambda)\|} \cdot (3 g^T (J^T J + \lambda I)^{-4} g) \\ &= \left(\frac{1}{\|p(\lambda)\|^3} \right) \cdot (-g^T (J^T J + \lambda I)^{-3} g)^2 + \frac{1}{\|p(\lambda)\|} \cdot (3 g^T (J^T J + \lambda I)^{-4} g) \\ &> 0 \end{aligned}$$

Note, that in the second equation $(\|p(\lambda)\|)'$ was replaced by (40). Thus, $\phi''(\lambda) > 0$ implies the strict convexity for $\lambda \in [0, \infty)$.
Now it can be argued as in the proof of Satz 11.38 in [DR11, p. 159], that the strict convexity implies that for the root λ it holds:

$$\phi'(0)\lambda < \phi(\lambda) - \phi(0).$$

Obviously, it is $\phi(\lambda) = 0$ as it is the root. Furthermore $\phi'(0)$ is negative due to (40), so the following statement is true:

$$\lambda > -\frac{\phi(0)}{\phi'(0)}$$

Case 2: If J has no full rank, then the inverse of $J^T J$ does not exist and neither $\phi'(0)$. Because we are looking for $\lambda > 0$, $l_0 = 0$ is a natural lower boundary. □

Now that all the building blocks are together, the following Algorithm 2 can be presented to find an appropriate λ. This algorithm follows the structure of Moré [Mor78, p. 111].

Algorithm 2 Finding $\lambda > 0$

Given λ_0, $\Delta > 0$
Calculate the initial interval of uncertainty $(l_0, u_0]$ given in Lemma 3.5
for k = 0,1,2 ... **do**
  if $\lambda_k \notin (l_k, u_k]$ **then**
    Set $\lambda_k = \max\{0.001u_k, \sqrt{l_k u_k}\}$
  end if
  Calculate $\phi(\lambda_k)$ and $\phi'(\lambda_k)$
  if $\phi(\lambda_k) < 0$ **then**
    Set $u_{k+1} = \lambda_k$
  else
    Set $u_{k+1} = u_k$
  end if
  Set $l_{k+1} = \max\left\{l_k, \lambda_k - \frac{\phi(\lambda_k)}{\phi'(\lambda_k)}\right\}$
  Calculate

$$\lambda_{k+1} = \lambda_k - \left(\frac{\phi(\lambda_k) + \Delta}{\Delta}\right) \cdot \left(\frac{\phi(\lambda_k)}{\phi'(\lambda_k)}\right)$$

end for

In Algorithm 2 there are a few new parts, which will be briefly discussed in the following:

- If the current iterate λ_k does not lie in the interval of uncertainty, then it has to be altered to do so. An easy way to achieve this would be to choose the middle point of the interval of uncertainty, which is the arithmetic mean: $\frac{1}{2}(u_k + l_k)$. However, the geometric mean $\sqrt{l_k u_k}$ is used instead. It has the advantage that it is smaller than the arithmetic mean and lies therefore closer to the lower boundary l_k. This bias is more favorable for the algorithm, see for more information the book by Conn, Gould and Toint [CGT00, pp. 189–190] .
  However, the choice becomes problematic whenever $l_k = 0$, because then the geometric mean is also zero. To avoid this, λ is chosen as the maximum of $0.001u_k$ and the geometric mean.

- The update of u_k can be determined by the value of $\phi(\lambda_k)$. Because ϕ is monotonically decreasing, $\phi(\lambda_k) < 0$ implies that λ_k is a smaller upper bound for the intervall of uncertainty than u_k and it can be shrunk.
- The update of l_k requires more effort. Here the Newton iterate is used, which is possible because of the convexity of ϕ shown in the proof of Lemma 3.5. If it is smaller than the current lower bound, l_k is not changed.

Algorithm 2 converges quadratically to the λ we are looking for [Mor78, p. 111]. A proof and a more detailed introduction of the algorithm for a general trust-region method can be found in [CGT00, pp. 181–193].
For the Levenberg-Marquardt method an approximation for λ will be sufficient. This can usually be achieved in only two or three iterations, according to Nocedal and Wright [NW06, p. 87].
Finally, it has to be decided which starting value λ_0 is chosen. It is natural to argue similarly as in the case where λ_k lies not in the interval of uncertainty. Thus, in the following implementation we will use:

$$\lambda_0 = \max\{0.001u_0, \sqrt{l_0 u_0}\},$$

as the first iterate.

3.3 Algorithm

3.3.1 Overview

By combining the general trust-region algorithm given in Algorithm 1 and the calculation of the subproblem (19) it is already possible to present a first version of the Levenberg-Marquardt method. It is illustrated in the flowchart in Figure 2.

3.3.2 Stopping Criteria

To make the Levenberg-Marquardt method practical, a suitable stopping criteria has to be added. An easy approach, like recommended in [UU12, p. 22], would be to choose ε, for example $\varepsilon = 10^{-8}$, and use it as a threshold to terminate the algorithm whenever the norm of the gradient at a current iterate drops below it. In summation, the iteration process would then be terminated once an iterate x_k holds that:

$$\|\nabla f(x_k)\| \leq \varepsilon. \tag{44}$$

To make the stopping criteria more adaptive to the gradient evaluated at the initial starting point, the considerations in [Kel99, p. 16] are followed. Therefore the stopping criteria ε is chosen as a combination of an absolute error bound τ_{abs} and a relative error bound τ_{rel} with $0 < \tau_{abs} \leq \tau_{rel}$. The stopping criteria is then given by:

$$\|\nabla f(x_k)\| \leq \tau_{rel}\|\nabla f(x_0)\| + \tau_{abs}. \tag{45}$$

However, if $\|\nabla f(x_0)\|$ is very large, the righthand side in (45) might become too large. If this case occurs, it might be more reasonable to follow (44). Based on these considerations, in the implementation we will use the minimum of the thresholds in (45) and (44). In the implementation, the corresponding parameters $\varepsilon, \tau_{rel}, \tau_{abs}$ will be made adjustable, such that the user can choose its preferred stopping criteria.

Overview of the Levenberg-Marquardt method

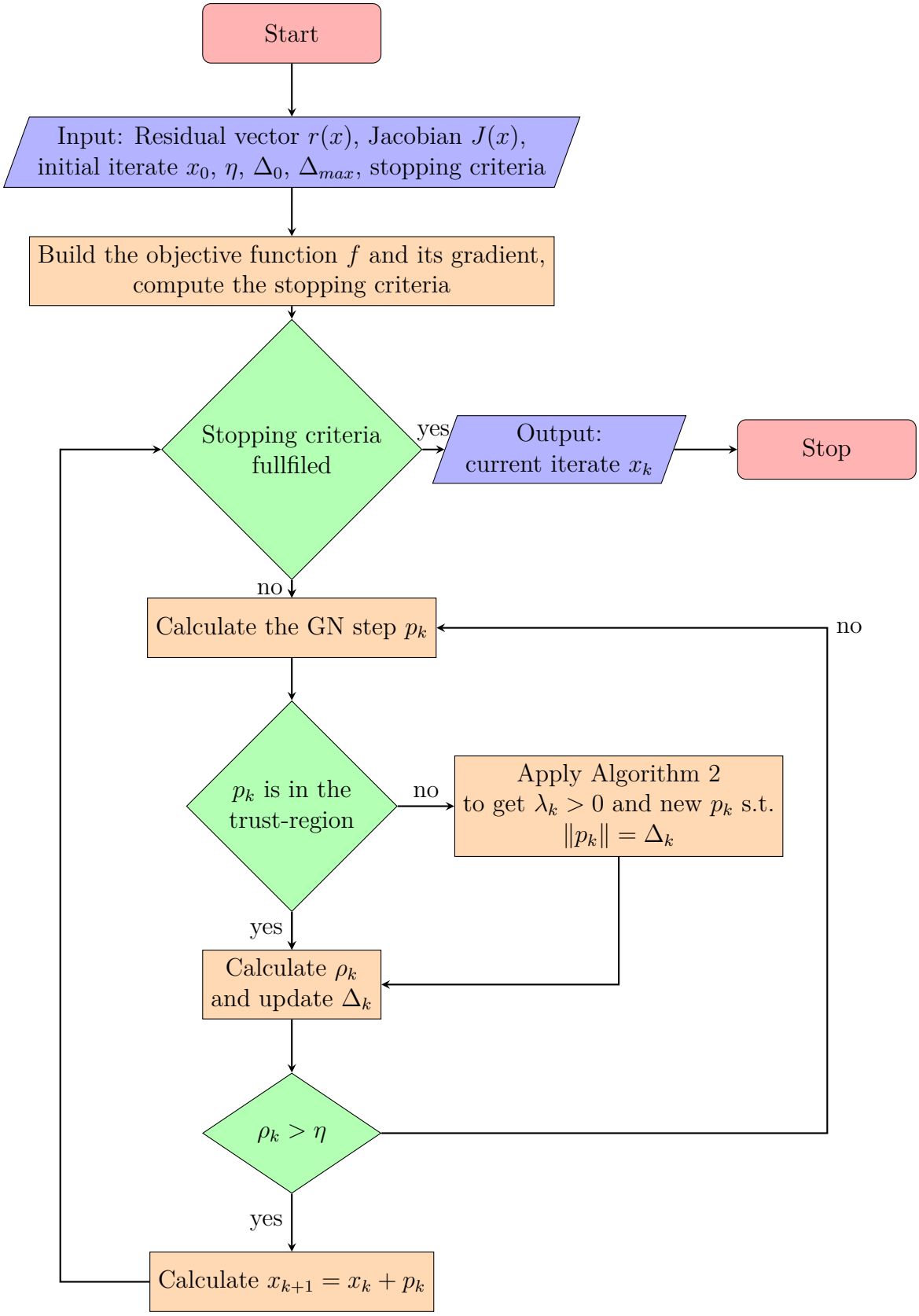


Figure 2: Overview of the Levenberg-Marquardt method, the input parameters Δ_{max}, Δ_0 and η are described in Algorithm 1.

3.4 Convergence of the Levenberg-Marquardt Method

3.4.1 Global Convergence

Under some conditions, the presented Levenberg-Marquardt method converges globally to a stationary point of the objective function f. Since the Levenberg-Marquardt method is a special trust-region method, results concerning the convergence of the general trust-region algorithm in Algorithm 1 can be applied to it.
The following section is therefore divided into two parts. First, a theorem on the convergence of the general trust-region algorithm is cited. In the second step, this theorem is applied to the presented Levenberg-Marquardt method. This section is based on [NW06, pp. 82-83 and pp. 261-262].

For the following theorem, some conditions have to be fulfilled in order that the general trust-region algorihm presented in Algorithm 1 converges. Those conditions are summarized in the following box:

Conditions for global convergence properties of the general trust-region algorithm:

- The norms of the approximate Hessians are uniformly bounded:

$$\|B_k\| \leq \beta \qquad \text{for some constant } \beta. \tag{46a}$$

- The objective function f is bounded from below on the level set:

$$S := \{x \,|\, f(x) \leq f(x_0)\}, \tag{46b}$$

  where x_0 represents the initial iterate of the algorithm.

- f is continuously Lipschitz differentiable on the set $S(R_0)$, which is defined as

$$S(R_0) := \{x \,|\, \|x - y\| < R_0 \;\text{ for some } y \in S\}, \tag{46c}$$

  for some constant $R_0 > 0$. Note that $S(R_0)$ is an open neighborhood around S.

- In practical applications the trust-region radius Δ_k in Algorithm 1 is allowed to be more versatile. Therefore a constant $\sigma \geq 0$ is introduced, such that the trust-region constraint of the subproblem is modified to:

$$\|p_k\| \leq (1 + \sigma)\Delta_k. \tag{46d}$$

- The approximate solution p_k of the subproblem (17) has to fullfil the condition:

$$m_k(0) - m_k(p_k) \geq c_1 \|g_k\| \min\left(\Delta_k, \frac{\|g_k\|}{\|B_k\|}\right) \tag{46e}$$

  for some constant $c_1 \in (0, 1]$.

One important note concerning condition (46d) is that it is weaker than in the original Algorithm 1. The usage of the more versatile trust-region has an advantage for the Levenberg-Marquardt method, because it also implies that it is sufficient to get an approximative λ_k through Algorithm 2. This will be used in the implementation and will

be discussed later in Section 3.5.3.

Theorem 3.6. *Let $\eta \in (0, \frac{1}{4})$ in the general trust-region algorithm. Suppose that the conditions* (46) *are fulfilled.*
Then it holds:

$$\lim_{k\to\infty} \nabla f_k = 0.$$

Proof. A proof can be found in the book by Nocedal and Wright [NW06, pp. 82-83] □

In the next step, the conditions (46) can be modified to the Levenberg-Marquardt method and Theorem 3.6 can be directly applied to our method. The following corollary is cited from [NW06, pp. 261-262].

Corollary 3.7. *Let $\eta \in (0, \frac{1}{4})$ in the Levenberg-Marquard method, and suppose that the objective function f is bounded below on the level set S. Furthermore the residual functions $r_j(\cdot)$ are continuously Lipschitz differentiable in $S(R_0)$ for some $R_0 > 0$ for every $j \in \{1, 2, \ldots, m\}$. Assume that for each k, the approximate solution p_k of the subproblem* (19) *satisfies the inequality:*

$$m_k(0) - m_k(p_k) \geq c_1 \|J_k^T r_k\| \min\left(\Delta_k, \frac{\|J_k^T r_k\|}{\|J_k^T J_k\|}\right)$$

for some constant $c_1 > 0$ and in addition $\|p_k\| \leq (1+\sigma)\Delta_k$ for some constant $\sigma \geq 0$.
Then it holds:

$$\lim_{k\to\infty} \nabla f_k = \lim_{k\to\infty} J_k^T r_k = 0.$$

Proof. This is a direct application of Theorem 3.6. See for more details [NW06, p. 262]. □

3.4.2 Local Convergence

Locally, around a minimizer x^*, the trust-region constraint in the subproblem becomes inactive. As a consequence the Levenberg-Marquardt method takes the Gauss-Newton step p^{GN} to calculate the next iterate. Thus, it is possible to use results from the Gauss-Newton method and apply those to our method. The following presented result is also from the book by Nocedal and Wright [NW06, p. 257].

Before we start a short reminder: A key idea of the Levenberg-Marquardt method was to approximate $\nabla^2 f(x)$ by its first term in (4). The following result, stating rapid local convergence if the first term dominates the second-order term in (3) is therefore not suprising.

For the ease of reading, it is chosen $H(x) = \sum_{j=1}^m r_j(x)\nabla^2 r_j(x)$. Furthermore $J^T J(x)$ is used as an abbreviation for $J(x)^T J(x)$.
For the convergene result we have to assume that x_k is close to x^* and that there is a constant $\gamma > 0$ such that

$$\|J(x)z\| \geq \gamma \|z\| \tag{47}$$

for all $x \in S(R_0)$ as defined in (46c) [NW06, p. 257]. This means that the singular values of the Jacobians $J(x)$ are uniformly bounded away from zero.

Finally, if $H(x)$ is Lipschitz continuous near x^* we get that for our next step $x_{k+1} = x_k + p^{GN}$ in our iteration it holds [NW06, p.257]:

$$\begin{aligned}\|x_k + p_k^{GN} - x^*\| \\ &\leq \int_0^1 \|[J^T J(x_k)]^{-1} H(x^* + t(x_k - x^*))\| \|x_k - x^*\| \, dt + \mathcal{O}(\|x_k - x^*\|^2) \\ &\approx \|[J^T J(x^*)]^{-1} H(x^*)\| \|x_k - x^*\| + \mathcal{O}(\|x_k - x^*\|^2).\end{aligned} \tag{48}$$

The consequence is as expected, when the first term in (3) dominates the second term, in other words if $\|[J^T J(x^*)]^{-1} H(x^*)\| \ll 1$, then we have a rapid local convergence of the Levenberg-Marquardt method. The best scenario for the algorihtm would be the case when $H(x^*) = 0$, as the convergence rate is then quadatic.

3.5 Improvements

In the following section, two improvements to the Levenberg-Marquardt method are presented. The first improvement is concerned with the trust-region. Until now, only spherical trust-regions were considered. In order to take poor scaling of the problem into account, an ellipsoidal trust-region can be applied. The second improvement is rather minor and introduces a more robust way to calculate ρ_k, the measuremnt of agreement between the model function m_k and the objective function f. Finally, a last remark is given before the Levenberg-Marquardt method is implemented in Python in the next chapter.

3.5.1 Scaling

Some problems that occur are *poorly scaled*. Poorly scaled means that a change of x in a given direction leads to a large variation in the function value of f, whereas a change in another direction leads to only a small variation of its function value [NW06, p. 26]. This problem occurs often for least-squares problems, as on one hand, variables could have values around 10^4, while on the other hand, they could be around 10^{-6} [NW06, p. 260]. This has an important implication on the key concept of the trust-region approach, where the model function m_k around the current iterate x_k is used as an approximation of the objective function f within the trust-region. A poor scaling affects this approach, because m_k might be an inappropriate approximation along directions with large changes in f while being still an appropriate approximation of f along directions with minor changes.
The idea is therefore to include scaling properties of the objective function within the algorithm and alter the trust-region accordingly. As a consequence, the constraint in the subproblem (17) is changed from a spherical trust-region $\|p\| \leq \Delta$ to an ellipsoidal trust-region:

$$\|Dp\| \leq \Delta,$$

where D is defined as a diagonal matrix which will be chosen accordingly to the scaling properties of the problem. Consequently, the scaled trust-region subproblem is given by:

$$\min_{p \in \mathbb{R}^n} m(p) = f + g^T p + \frac{1}{2} p^T J^T J p \qquad \text{such that } \|Dp\| \leq \Delta. \tag{49}$$

The trust-region is now a hyperellipsoid and the main difference to a spherical constraint is that the length of the i-th semi-axis is depending on d_i, where d_i represents the i-th diagonal element of D. More precisely, the length of the i-th semi-axis is now Δ/d_i [Mor78,

p. 111]. Whenever a change in the direction x_i, where x_i is defined as the i-th component of x, results to a large variation in $f(x)$, d_i can be chosen larger.

There are several ways to construct the diagonal matrix D, where often they will use information from the first- or second-order derivative of f in order to characterize (approximately) the scaling properties [NW06, p. 96]. Moré introduces three approaches to face this challenge [Mor78, pp. 111-112]. An intuitive way would be to choose the diagonal elements of D in the beginning of our algorithm as:

$$d_i = \|\partial_i r(x_0)\|.$$

However, when $\|\partial_i r(x_k)\|$ increases with k, then this implies that r and thus f changes faster in this direction. As a consequence, this should be reflected in a decrease of the length of the i-th semi-axis of the trust-region. Therefore a second approach can be used, which calculates a new matrix D_k for every iteration step k in the following manner:

$$d_i^{(k)} = \|\partial_i r(x_k)\|, \qquad k \geq 0. \tag{50}$$

Nocedal and Wright [NW06, p. 261] follow the suggestions of Seber and Wild [SW03] and recommend to use the approximation of the Hessian to determine D. Therefore, they define the diagonal elements of D_k^2 as those of the diagonal elements of $J_k^T J_k$. This is equivalent to choose the diagonal elements like in (50).
Moré [Mor78, p. 112] recommends a third, slightly modified approach:

$$d_i^{(0)} = \|\partial_i r(x_0)\|$$

$$d_i^{(k)} = \max\left\{d_i^{(k-1)}, \|\partial_i r(x_k)\|\right\}, \qquad k \geq 1.$$

This approach ensures that a decrease in $\|\partial_i r(x_k)\|$ does not result in a decrease in $d_i^{(k)}$. The numerical results of Moré favor the third approach [Mor78, p. 115]. As a consequence, the third scaling variant is used in the following implementation. However, it is worth noting, that with all three options, the algorithm becomes scale invariant [Mor78, p. 112]. It is also important to note, that the calculation of the diagonal elements of D_k do not cost further computational effort, because the gradients of the residuals are already calculated in $J(x_k)$.

Due to the changed subproblem in (49), some new considerations must be made to find its minimizer. Fortunately, it is possible to build on previous considerations. The following Lemma 3.8 gives the counterpart of Theorem 3.1 for the new subproblem.

Lemma 3.8. *The vector p^{LM} is a global solution of the trust-region subproblem* (49) *if and only if p^{LM} is feasible and there is a scalar $\lambda \geq 0$ such that*

$$(J^T J + \lambda D^2)p^{LM} = -J^T r \tag{51a}$$

$$\lambda(\Delta - \|Dp^{LM}\|) = 0 \tag{51b}$$

Proof. A proof can be found in Theorem 7.4.1 in [CGT00, p. 201] □

Again, (51a) can be identified as the normal equations of the following linear least-squares problem:

$$\min_p \frac{1}{2}\left\| \begin{bmatrix} J_k \\ \sqrt{\lambda} D_k \end{bmatrix} p + \begin{bmatrix} r_k \\ 0 \end{bmatrix} \right\|^2. \tag{52}$$

In order to solve (52), it can be proceeded analogously like in Section 3.2.1. In fact, the calculation of the Gauss-Newton step p^{GN}, where $\lambda = 0$, does not change at all.
However, if p^{GN} does not lie within the trust-region, a new $\lambda > 0$ must be found such that for the solution p_k of (49), it is $\|D_k p_k\| = \Delta$.
Before doing so, the solution of the linear least-squares problem in (52) for a given $\lambda > 0$ is discussed.

In general, the ideas in Section 3.2.1 can be applied, the only difference is that $\sqrt{\lambda}D_k$ is used instead of $\sqrt{\lambda}I$. Thus, equation (25) changes slightly to

$$\begin{bmatrix} R \\ 0 \\ \sqrt{\lambda}D_\lambda \end{bmatrix} = \begin{bmatrix} Q^T & 0 \\ 0 & \Pi^T \end{bmatrix} \begin{bmatrix} J \\ \sqrt{\lambda}D \end{bmatrix} \Pi, \tag{53}$$

where $D_\lambda = \Pi^T D \Pi$ holds.
This slight modification is not problematic, because $D_\lambda = \Pi^T \sqrt{\lambda} D \Pi$ is a diagonal matrix either, so the QR decomposition of the left-hand side can still be found with the same Givens rotation approach.

In the case that the Gauss-Newton step p^{GN} does not lie in the trust-region, Algorithm 2 can still be used to find a suitable $\lambda > 0$. But this time $\tilde{J} = JD^{-1}$ is used instead of J, as it is shown in the next lemma [Mor78, p. 110].

Lemma 3.9. *The function $\phi(\lambda) = \|Dp(\lambda)\| - \Delta$ can be expressed for $\tilde{J} = JD^{-1}$ as*

$$\phi(\lambda) = \|(\tilde{J}^T\tilde{J} + \lambda I)^{-1}\tilde{J}^T r\| - \Delta. \tag{54}$$

Proof. We have:

$$\begin{aligned}
\phi(\lambda) &= \|D(J^TJ + \lambda D^2)^{-1}J^T r\| - \Delta \\
&= \|(D^{-1}(J^TJ + \lambda D^2)D^{-1})^{-1}D^{-1}J^T r\| - \Delta \\
&= \|(D^{-1}J^TJD^{-1} + \lambda D^{-1}D^2D^{-1})^{-1}D^{-1}J^T r\| - \Delta \\
&= \|((JD^{-1})^TJD^{-1} + \lambda I)^{-1}(JD^{-1})^T r\| - \Delta \\
&= \|(\tilde{J}^T\tilde{J} + \lambda I)^{-1}\tilde{J}^T r\| - \Delta,
\end{aligned}$$

which gives the claim. □

Finally, to use Algorithm 2, we have to be able to evaluate $\phi'(\lambda_k)$. This time we have instead of (40):

$$\phi'(\lambda) = \frac{\partial\|Dp(\lambda)\|}{\partial\lambda} = -\frac{1}{\|Dp(\lambda)\|} \cdot (D^2p(\lambda))^T(J^TJ + \lambda D^2)^{-1}(D^2p(\lambda)) \tag{55}$$

and thus, for q which solves the following equation $\Pi R_\lambda^T q = D^2p(\lambda)$ it is:

$$\phi'(\lambda) = -\frac{\|q\|^2}{\|Dp(\lambda)\|}. \tag{56}$$

With this, all the modifications are together, so we are able to solve the new subproblem (49) and include the scaling properties of the objective function.

Note that when we choose $D = I$, the subproblem with scaling (49) is identical to the subproblem without scaling (19). Thus, in the following implementation we can always use the modifications discussed in this chapter and choose D accordingly, whether scaling is included or not.

3.5.2 Calculation of ρ_k

For the calculation of ρ_k, there is a more robust approach available. The following lemma is due to Moré [Mor78, p. 108-109].

Lemma 3.10. *For the model function defined in* (49)*, ρ_k can be simplified:*

$$\rho_k = \frac{1 - \frac{f(x_k+p_k)}{f(x_k)}}{\frac{\frac{1}{2}\|J(x_k)p_k\|^2}{f(x_k)} + \frac{\lambda\|Dp_k\|^2}{f(x_k)}} \tag{57}$$

Proof. The following proof is based on the ideas presented in Moré [Mor78, p. 108]. After inserting the model function and the objective function in (16) we have:

$$\rho_k = \frac{\frac{1}{2}\|r(x_k)\|^2 - \frac{1}{2}\|r(x_k+p_k)\|^2}{\frac{1}{2}\|r(x_k)\|^2 - \frac{1}{2}\|J(x_k)p_k + r(x_k)\|^2}$$

Next the predicted reduction is simplified for a given step p_k:

$$\begin{aligned}
\text{predicted reduction} &= \frac{1}{2}\|r(x_k)\|^2 - \frac{1}{2}\|J(x_k)p_k + r(x_k)\|^2 \\
&= \frac{1}{2}\|r(x_k)\|^2 - \left(\frac{1}{2}\|r(x_k)\|^2 + r(x_k)^T J(x_k)p_k + \frac{1}{2}\|J(x_k)p_k\|^2\right) \\
&= -r(x_k)^T J(x_k)p_k - \frac{1}{2}\|J(x_k)p_k\|^2 \\
&= ((J(x_k)^T J(x_k) + \lambda D_k^2)p_k)^T p_k - \frac{1}{2}\|J(x_k)p_k\|^2 \\
&= p_k^T J(x_k)^T J(x_k)p_k + p_k^T \lambda D_k^2 p_k - \frac{1}{2}\|J(x_k)p_k\|^2 \\
&= \frac{1}{2}\|J(x_k)p_k\|^2 + \lambda\|D_k p_k\|^2.
\end{aligned} \tag{58}$$

Note that in the fourth equation it was used that p_k was chosen such that (51a) holds. Inserting (58) in ρ_k plus dividing the numerator and denominator by $f(x_k) \neq 0$ gives the result. □

As a consequence, the denominator of (57) is always non-negative. Furthermore, it will not lead to a potential overflow because with (58) it is clear that $\frac{1}{2}\|J(x_k)p_k\| \leq f(x_k)$ and $\lambda\|D_k p_k\|^2 \leq f(x_k)$.
By setting $\rho_k = 0$, if $f(x_k + p_k) > f(x_k)$ the numerator cannot produce an overflow. As a consequence we get a robust calculation of ρ_k.

3.5.3 Final Remark

For finding $\lambda > 0$ in Algorithm 2, a suitable stopping criteria has yet to be presented. As pointed out by Nocedal and Wright [NW06, p. 87], it is usually sufficient to get an approximate solution for λ. For the practical implication it is useful to stop the algorithm as soon as a step p_k and $\lambda > 0$ is found such that:

$$(1 - \sigma)\Delta \leq \|D_k p_k\| \leq (1 + \sigma)\Delta, \tag{59}$$

where $\sigma > 0$ is a constant [Mor78, p. 109]. As a consequence σ makes the trust-region bound more versatile, instead of the exact goal to find $\|D_k p_k\| = \Delta$. This fits the convergency properties, where this was already included in (46d). Accordingly, the Gauss-Newton step p_k is taken, whenever it holds:

$$\|D_k p_k\| \leq (1 + \sigma)\Delta. \tag{60}$$

4 Implementation in Python

"Seven at one blow!"

The valiant little tailor
(from a fairy tale by the
Brothers Grimm)

In this section the final implementation of the Levenberg-Marquardt in Python is introduced. First, a brief overview of the algorithm is presented and final questions concerning the calculation of Givens rotations and optional input parameters are addressed. Finally, the algorithm is put to the test by solving seven problems. Here it will be shown that the algorithm has difficulties especially in problems with large residuals.

4.1 Preparations

For the implementation Python 3 is used [Fou]. To deal with matrices in general the NumPy package is imported. In the implementation we will often face the challenge to solve linear equation systems. The SciPy package provides a function to solve such a task and is therefore used.
As well, the computation of several QR decompositions is required. As described in Section 3.2.1, the QR decomposition will depend on whether the solution is for the linear least-squares problem in Part 1 for $\lambda = 0$ or in the second part for $\lambda > 0$. For the first part a standard QR decomposition with column pivoting is required. Therefore the SciPy package can be again used.
However, for the second part a QR decomposition must be calculated through Givens rotations as described in Section 3.2.1. At this point, no suitable function could be found, so two auxiliary functions were implemented to account for this purpose. These are also introduced in the following, and the code is placed in the appendix.

Note: It is tempting to use the SciPy function to solve the linear least-square problem directly for each given λ. The algorithm would work in this way too, nevertheless we would not recommend to do so, because it neglects the specialties of our problem described in Section 3.2.1 and is therefore computationaly less efficient.

4.1.1 Optional Input Parameters

It is obvious that the residual vector $r(x)$ and its Jacobian $J(x)$ are required so that the algorithm can be started, as well as the first iterate x_0. Besides that, a few optional parameters can be chosen, otherwise their default value is used. All adjustable parameters with their short description and default values are given in Table 1.

It is important to note, that scaling is not applied in the default settings. If a problem is poorly scaled the option "Scaling" should be set "True". The scaling matrix D might lead to the situation that $\|Dp\|$ gets very large, such that "Delta_max" needs to be increased as well in the optional parameters. In this way it is ensured that the algorithm can still make a large enough step p which fullfils the constraint $\|Dp\| \leq (1 + \sigma)\Delta$.

Adjustable Parameters	Description
nmax	Maximum number of iterations. The default is 500.
Delta	Inital trust-region radius Δ. The default is 100.
Delta_max	Upper bound for the trust-region radius Δ_k. The default is 10^4.
eta	Threshold eta $\in (0,1)$. It determines whether a step is taken or not. The default is 10^{-4}.
sigma	It determines the possible deviation from the trust region, sigma must lie in $(0,1)$. The default is 0.1.
tol_abs	Absolute error bound. The default is 10^{-7}.
tol_rel	Relative error bound. The default is 10^{-7}.
eps	Upper bound for the tolerance. The default is 10^{-3}.
Scaling	If True, then a scaling procedure is applied, otherwise not. The default is False.

Table 1: Optional input parameters in the implementation of the Levenberg-Marquardt method.

4.1.2 QR Decomposition With Givens Rotations

In order to apply the QR decomposition in Part 2 as described in Section 3.2.1, Givens rotations are required. A concise introduction on Givens rotations can be found in [Lyc20, p. 117-119]. The concerning implementation builds on these considerations.
Two functions were implemented in order to obtain the desired QR decomposition of a matrix as in (26). The implementation can be found in the appendix.
To illustrate the process, it is useful to look at an example for a matrix such as in (26). Consider the matrix A with 2 columns and $3+2$ rows

$$A = \begin{pmatrix} 1 & 2 \\ 0 & 3 \\ 0 & 0 \\ 2 & 0 \\ 0 & 2 \end{pmatrix}.$$

Through the command:

R,Q = givens_qr(A,2,3)

we get the following matrices (all values are rounded until the third digit)

$$R = \begin{pmatrix} 4.123 & 0.485 \\ 0 & 5.363 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}, \qquad Q = \begin{pmatrix} 0.243 & 0.351 & 0 & -0.904 & 0 \\ 0 & 0.560 & 0 & 0.217 & -0.8 \\ 0 & 0 & 1 & 0 & 0 \\ 0.970 & -0.088 & 0 & 0.226 & 0 \\ 0 & 0.746 & 0 & 0.289 & 0.6 \end{pmatrix},$$

such that the the desired QR decomposition holds:

$$Q^T A = R.$$

4.2 The Final Algorithm

Finally, all the building blocks can be brought together in order to present the implementation of the Levenberg-Marquardt method. In the following, the commented Python code is given. In the code the name "lambda" could not be used, because it is a keyword in Python. This is the reason, why in the following "alpha" is used instead of "lambda".

```
import numpy as np
from numpy import inf
import scipy
from scipy import linalg


def levenberg_marquardt(r, J, x, Delta = 100, Delta_max = 10000,
                        eta = 0.0001, sigma = 0.1,
                        nmax = 500, tol_abs = 10**(-7),
                        tol_rel= 10**(-7), eps = 10**(-3),
                        Scaling = False):
    """
    Description
    -----------
    This function performs the Levenberg-Marquardt method,
    an iterative algorithm to find a minimizer of a
    given nonlinear least-squares problem.
    See for more details Chapter 3.


    Parameters
    ----------
    r : function
        Residual vector r(x).
    J : function
        Jacobian matrix of the residual vector r(x).
    x : float
        Starting point of the iteration process.
    The description of the optional parameters
    can be found in Table 1.

    Returns
    -------
    x : ndarray
        The last iterate of the iteration process.
    Information : list
        List which contains information about the
        iteration process. It contains for every
        taken step p its norm, the new iterate x
        and the norm of the gradient at x.

    """
```

```
## Preliminaries

# Construction of the objective function f(x)
# and its gradient gradf(x)
def f(x):
    return 0.5*np.linalg.norm(r(x))**2
def gradf(x):
    return np.dot(np.transpose(J(x)),r(x))

# counter for the number of iterations
counter = 0
# first function evaluation at x
fx = f(x)
# counter for the number of function evaluations
func_eval = 1
# dimensions (m,n) of the problem
m,n = J(x).shape

# list for storing information
Information = [['counter', 'norm of step p',
                'x','norm of the gradient at x']]
# append starting situation
Information.append([counter, 'not available',x,
                    np.linalg.norm(gradf((x)))])

# define tolerance
tolerance = min((tol_rel * Information[-1][-1]
                 + tol_abs), eps)

# for Scaling: two identity matrices are created
D = np.eye(n)
D_inv = np.eye(n)

## Calculation of step p
while Information[-1][-1] > tolerance and counter < nmax:

    # often used, thus stored
    Jx = J(x)

    # Construct the Scaling matrix D (like in chapter 3.5.1)
    if Scaling == True:
        for i in list(range(0,n,1)):
            D[i,i] = max(D[i,i], np.linalg.norm(Jx[:,i]))
        # Calculate also the inverse of D
        for i in list(range(0,n,1)):
            D_inv[i,i] = 1/D[i,i]
    # else: D stays simply the identity matrix
```

```
# often used, thus stored
D_2 = np.dot(D,D)


## Calculation of the Gauss-Newton step p

# "Part 1" QR Decomposition of Jx
# It can be calculated with a SciPy function
Q, R, Pi = scipy.linalg.qr(Jx,  pivoting=True)

# Calculate the permutation matrix P
P = np.eye(n)[:,Pi]

# Calculate Gauss-Newton step p
# It depends if Jx has full rank
rank = np.linalg.matrix_rank(Jx)
if rank==n: #unique solution
    p = np.dot(P, scipy.linalg.solve_triangular(
        R[0:n,:], np.dot(Q[:,0:n].T,-r(x))))
else:
    y = np.zeros((n))
    y[0:rank]=scipy.linalg.solve_triangular(
        R[0:rank,0:rank],np.dot(Q[:,0:rank].T,-r(x)))
    p = np.dot(P,y)

# often used, thus stored
Dp = np.linalg.norm(np.dot(D,p))


# Check if p is in the trust-region
if Dp <= ((1+sigma)*Delta):
    # p is already sufficient!
    alpha = 0

else:
    ## Algorithm 2 is started to find a suitable alpha

    # Calculate safeguards for the iteration
    # substitute J by J_scaled
    J_scaled = np.dot(Jx,D_inv)

    # upper boundary u
    u = np.linalg.norm(np.dot(J_scaled.T,r(x)))/Delta

    # lower boundary l
    # It depends if Jx has full rank
    if rank==n:
        q = scipy.linalg.solve_triangular(
            R[0:n,:].T, np.dot(P.T,np.dot(D_2,p)),
```

```
            lower = True)

        # Compute l = -phi(0)/phi'(0)
        l = (Dp- Delta)/(np.linalg.norm(q)**2
                                        /Dp)
    else:
        l = 0

    # Calculate the first alpha
    if u == inf:#safeguard, against possible overflow
        alpha = 1
    else:
        alpha = max(0.001*u, (l*u)**(0.5))

    # Start the iteration process
    while Dp > (1+sigma)*Delta or Dp < (1-sigma)*Delta:

        if alpha == inf: #alpha gets too big
            print('Error: '
                  + 'The LM method fails to converge.'
                  + '(Lambda gets too large)'
                  + 'Please try a different starting point.')
            return x, Information

        # safeguarding alpha
        if alpha <= l or alpha > u:
            alpha = max(0.001*u, (l*u)**(0.5))

        # "Part 2:" Define R_I
        D_lambda = np.dot(P.T,np.dot(D,P))
        R_I = np.concatenate((R,alpha**(0.5)*
                              D_lambda), axis = 0)

        # QR decomposition of R_I
        # Givens rotations are used
        R_lambda, Q_lambda2 = givens_qr(R_I, n,m)

        # Build Q_lambda like described in (27)
        Q_lambda = np.dot(np.concatenate(
            (np.concatenate((Q,np.zeros((m,n))),axis = 1),
             np.concatenate((np.zeros((n,m)),P), axis = 1)),
            axis=0),Q_lambda2)

        # n additional zeros for r
        r_0 = np.append(r(x), np.zeros(n))

        # p can now be calculated, like in (29)
        p = np.dot(P, scipy.linalg.solve_triangular(
            R_lambda[0:n,:],np.dot(Q_lambda[:,0:n].T,-r_0)))
```

```
            # often used, thus stored:
            Dp = np.linalg.norm(np.dot(D,p))

            # Calculate phi(alpha) and phi'(alpha)
            q = scipy.linalg.solve_triangular(
                R_lambda[0:n,:].T,
                np.dot(P.T,np.dot(D_2,p)), lower=True)
            phi = Dp-Delta
            phi_derivative = -np.linalg.norm(q)**2/Dp

            # update u
            if phi < 0:
                u = alpha
            # update l
            l  = max(l, alpha - phi/phi_derivative)

            # Compute the new alpha
            alpha = alpha - ((phi+Delta)/
                             Delta) * (phi/phi_derivative)



        ## Calculation of rho
        fxp = f(x+p) # new function value at x+p
        func_eval += 1

        # Note: fxp might be nan or inf, so reject p
        if fxp > fx or fxp == inf or np.isnan(fxp)==True:
            rho = 0

        else:
            ared = 1 - (fxp/fx)
            pred = (0.5*np.linalg.norm(np.dot(Jx, p))**2)/fx+(
                alpha*Dp**2)/fx
            rho = ared/pred

        # Update Delta accordingly
        if rho < 0.25:
            Delta = 0.25 * Delta
        else:
            if (rho > 0.75 and (Dp>= (1-sigma)*Delta)):
                Delta = min(2*Delta,Delta_max)
            else:
                Delta = Delta
        if rho > eta:
            x = x + p  # take the step

            # Update the Information
```

```
            fx = fxp
            counter += 1
            Information.append([counter, np.linalg.norm(p),x,
                                np.linalg.norm(gradf((x)))])

    if Information[-1][-1] <= tolerance:
        print('The LM method terminated successfully.')
        print('\n    Current function value: ' + str(fx))
        print('    Iterations: ' + str(counter))
        print('    Function evaluations: ' + str(func_eval))
    else:
        print('The LM method fails to converge within '
              + str(nmax) + ' steps.')
    return x, Information
```

4.3 Numerical Results

After the implementation, the Levenberg-Marquardt method can now be tested. To do so, seven example functions are selected. After a brief introduction of these examples, the numerical results are presented.
In the following, all values are rounded to three decimal places.

4.3.1 Seven Test Problems

Small-Residual Problems:

1. Rosenbrock function [Ros60] $m = 2, n = 2$
   Objective function:

$$f(x) = (1 - x_1)^2 + 100(x_2 - x_1^2)^2. \tag{61}$$

   To reformulate the objective function and get the form (1), residuals can be chosen as follows:

$$r_1(x) = \sqrt{2}(1 - x_1),$$
$$r_2(x) = 10\sqrt{2}(x_2 - x_1^2).$$

   The function has one minimum at $x^* = (1, 1)^T$ with $f(x^*) = 0$.
   As a starting point for the iteration the vector $x_0 = (0.1, -0.1)^T$ is used.

2. Himmelblau's function [Him72] $m = 2, n = 2$
   Objective function:

$$f(x) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2. \tag{62}$$

   Again, the objective function can be reformulated to obtain the form in (1), such that the residuals are chosen as follows:

$$r_1(x) = \sqrt{2}(x_1^2 + x_2 - 11),$$
$$r_2(x) = \sqrt{2}(x_1 + x_2^2 - 7).$$

   The function has four minima at $x_1^* = (3, 2)^T$, $x_2^* = (-2.805, 3.131)^T$, $x_3^* = (-3.779, -3.283)^T$, $x_4^* = (3.584, -1.848)^T$.
   At every minimizer the function value is 0.

Functions 1 and 2 are both standard test functions. Both have in common that around a minimizer x^* it is $f(x^*) = 0$. Thus the residuals are zero for x^* and the Levenberg-Marquardt method converges locally quadratically, as stated in (48). It is therefore no surprise that the following tests will turn out to be favorable.

Application Problems:

3. Pasture Regrowth [Hue+04; Rat83] $m = 9, n = 4$
The residuals are given by

$$r_j(x) = x_1 - x_2 \exp[-\exp(x_3 + x_4 \log(t_j))] - y_j,$$

where the data (t_j, y_j) is specified in the following table:

Time t after grazing	9	14	21	28	42	57	63	70	79
Yield y	8.93	10.8	18.59	22.33	39.35	56.11	61.73	64.92	67.08

As a starting value it is $x_0 = (80, 70, -10, 2.5)^T$ chosen.
Huet et al. [Hue+04, p. 13] find a minimizer at $x^* = (69.95, 61.68, -9.209, 2.378)^T$.

4. Population Growth [CPR12] $m = 8$, $n = 2$
The residuals are given by

$$r_j(x) = x_1 \exp(x_2 t_j) - y_j,$$

where the data (t_j, y_j) is specified in the following table:

t (Year)	1 (1815)	2 (1825)	3 (1835)	4 (1845)	5 (1855)	6 (1865)	7 (1875)	8 (1885)
y Population	8.3	11.0	14.7	19.7	26.7	35.2	44.4	55.9

As a starting value it is $x_0 = (0.6, 0.3)^T$ chosen.
A minimum can be found at $x^* = (7.000, 0.262)^T$.

5. Feulgen Hydrolysis Kinetics [DH19] $m = 30, n = 3$
The residuals are given by

$$r_j(x) = x_1 \exp(-(x_2^2 + x_3^2)t_j) \frac{\sinh((x_3^2)t_j)}{x_3^2} - y_j,$$

where the data (t_j, y_j) is specified in the following table:

t	6	12	18	24	30	36	42	48	54	60
y	24.19	35.34	43.43	42.63	49.92	51.53	57.39	59.56	55.60	51.91
t	66	72	78	84	90	96	102	108	114	120
y	58.27	62.99	52.99	53.83	59.37	62.35	61.84	61.62	49.64	57.81
t	126	132	138	144	150	156	162	168	174	180
y	54.79	50.38	43.85	45.16	46.72	40.68	35.14	45.47	42.40	55.21

As a starting value it is used $x_0 = (8, 0.055, 0.21)^T$.
A minimum can be found at $x^* = (3.536, 0.055, 0.154)^T$.

At this point, we will not discuss the underlying ideas, which results in the formulation of our application problems. This information can be found in the corresponding books or papers stated above.

Note that in practical applications it is sometimes possible to transform the data in order to obtain a linear least-squares problem. Thus, the problem can be solved easier and the usage of the Levenberg-Marquardt method is not required. A concise overview about such transformations can be found in the book by Hedderich and Sachs [HS16, pp. 150–153]. For example, a logarithm transformation in the population data in Problem 4 would be useful, such that it can be modelled with a linear function. Because we want to test the Levenberg-Marquardt we will not apply such a transformation.

Large-Residual Problems:

6. Function of Brown and Dennis [BD71] $m = 20, n = 4$
   The residuals are given by

$$r_j(x) = [x_1 + x_2 t_j - \exp(t_j)]^2 + [x_3 + x_4 \sin(t_j) - \cos(t_j)]^2, \tag{63}$$

   where $t_j = 0.2j$ and $j \in \{1, \ldots, 20\}$.

   According to Brown and Dennis [BD71] the first iterate can be chosen at $x_0 = (25, 5, -5, 1)^T$ and the function has a minimum at $x^* = (-11.594, 13.204, -0.403, 0.237)^T$.

7. Modified Function of Brown and Dennis $m = 20, n = 4$
   In order to make the function more difficult, the parameters x_1 and x_3 are rescaled, x_1 by the factor 10^3 and x_3 by the factor 10^{-3}. We then have a slightly modified version of the residuals above:

$$r_j(x) = [10^3 x_1 + x_2 t_j - \exp(t_j)]^2 + [10^{-3} x_3 + x_4 \sin(t_j) - \cos(t_j)]^2,$$

   where $t_j = 0.2j$ and $j \in \{1, \ldots, 20\}$.

   The starting vector is altered accordingly. It is now chosen $x_0 = (10^{-3} \cdot 25, 5, 10^3 \cdot (-5), 1)^T$.
   A minimum can be found at $x^* = (-11.594 \cdot 10^{-3}, 13.204, -0.403 \cdot 10^3, 0.237)^T$.

The function of Brown and Dennis has a special feature, at the minimizer x^* it has a large residual with $\|r(x^*)\| \approx 292.954$. Thus, we can assume that the approximation of the Hessian (4) will be inadequate around x^*. As a consequence the Levenberg-Marquardt method has troubles to converge during solving the large-residual problem. Moré [Mor78, p. 115] uses Problem 6 to decide which of the three scaling options, introduced in 3.5.1, is working best. As it turns out in our results, Problem 6 is not that poorly scaled and the Levenberg-Marquardt method is still able to solve it without the scaling option set to "True".
To combine the large-residual problem with the other weakness, poor scaling, the function was modified. Thus, the seventh problem is particularly difficult for the Levenberg-Marquardt method.

4.3.2 Test Results

The results of the small-residual problems are stated in Table 2. As already assumed, these problems are easy for the Levenberg-Marquardt method to solve. This can be illustrated by the first problem with the starting value $10x_0$, where it only needs one step to find the minimizer. In the table the number of function evaluations is stated as well for every test. These results are also good for the Levenberg-Marquardt method. More detailed, for the second problem, 6 or 9 function evaluations are required, and in the first problem, between 2 and 15 evaluations are needed.
Finally, it is noteworthy that our method converges quickly even if we choose a value far away from the actual minimizer, like $100x_0$. These results support the theoretical consideration, such that the Levenberg-Marquardt method works especially fine when the residuals of the underlying problem are small.

Problem	Starting Value	Number of Iterations	Function Evaluations	Result x^*	$\|\nabla f(x^*)\|$	$f(x^*)$
1	x_0	9	15	$(1,1)^T$	0	0
	$10x_0$	1	2	$(1,1)^T$	$1.027 \cdot 10^{-13}$	$8.493 \cdot 10^{-30}$
	$100x_0$	2	3	$(1,1)^T$	$8.506 \cdot 10^{-12}$	$3.682 \cdot 10^{-26}$
2	x_0	5	9	$(3,2)^T$	$9.18 \cdot 10^{-7}$	$1.212 \cdot 10^{-14}$
	$10x_0$	5	9	$\begin{pmatrix} -2.805 \\ 3.131 \end{pmatrix}$	$2.192 \cdot 10^{-8}$	$3.668 \cdot 10^{-18}$
	$100x_0$	5	6	$\begin{pmatrix} -2.805 \\ 3.131 \end{pmatrix}$	$2.473 \cdot 10^{-6}$	$3.796 \cdot 10^{-14}$

Table 2: Test results for small-residual problems, all optional values are set to default.

The test results of the application problems can be found in Table 3. Starting from the recommended initial value x_0, the Levenberg-Marquardt method can solve Problem 3 in 5, Problem 4 in 7 iterations and Problem 5 in 10 iterations. A closer look at Problem 3 and Problem 4 reveal that they are rather small-residual problems. In fact, at the respective minimum x^*, it is in Problem 3 $\|r(x^*)\| \approx 2.908$ and in Problem 4 $\|r(x^*)\| \approx 2.452$. However, problem 5 has a larger residual at x^* with $\|r(x^*)\| \approx 27.870$.

When we start the iteration further away of the corresponding minimizer x^*, like starting at $10x_0$ the iteration still converges in Problem 3 and 4. However, concerning Problem 3 the iteration needs now 30 steps in the case of $10x_0$ and if $100x_0$ is chosen, the algorithm converges in only 7 steps to a different stationary point.
The function values of Problem 4 tend to get very large, as a consequence the residuals are also large and the approximation of the Hessian might be poor. For example if we start the iteration at $100x_0$, it is $f(100x_0) \approx 5.207 \cdot 10^{211}$. To avoid potential overflows we used as starting values $10x_0$ and $15x_0$. For $15x_0$ the algorithm needed 63 steps.

Problem 5 causes more problems for the Levenberg-Marquardt method. Besides the rather big residuals around the minimizer, it seems to be poorly scaled: a change in x_2 can have a bigger effect on the respective objective function f than a change in x_1. The starting values $10x_0$ and $100x_0$ cannot be used this time, because $f(10x_0)$ and $f(100x_0)$ are both too large and result in an overflow. Thus, it is used $5x_0$ as the first iterate. The algorithm

then fails to converge within 500 steps. However, if we include the scaling properties by setting the scaling option to "True", the Levenberg-Marquardt method works fine and x^* is obtained within 30 iterations.

To summarize, the Levenberg-Marquardt method is able to solve the three application problems in a reasonable amout of steps. If the problem is poorly scaled, it might be the case that, as seen in Problem 5, the scaling option has to be used, otherwise the algorithm might fail to converge.

Problem	Starting Value	Number of Iterations	Function Evaluations	Result x^*	$\|\nabla f(x^*)\|$	$f(x^*)$	$\|r(x^*)\|$
3	x_0	5	6	$\begin{pmatrix} 70.068 \\ 61.773 \\ -9.227 \\ 2.382 \end{pmatrix}$	$6.45 \cdot 10^{-5}$	4.227	2.908
	$10x_0$	30	40	$\begin{pmatrix} 70.068 \\ 61.773 \\ -9.227 \\ 2.382 \end{pmatrix}$	$8.527 \cdot 10^{-4}$	4.227	2.908
	$100x_0$	7	8	$\begin{pmatrix} 62.46 \\ 42.46 \\ -1000 \\ 250 \end{pmatrix}$	$5.382 \cdot 10^{-12}$	328.638	25.637
4	x_0	7	11	$\begin{pmatrix} 7.000 \\ 0.262 \end{pmatrix}$	$2.068 \cdot 10^{-4}$	3.007	2.452
	$10x_0$	25	31	$\begin{pmatrix} 7.000 \\ 0.262 \end{pmatrix}$	$6.742 \cdot 10^{-5}$	3.007	2.452
	$15x_0$	63	72	$\begin{pmatrix} 7.000 \\ 0.262 \end{pmatrix}$	$3.584 \cdot 10^{-5}$	3.007	2.452
5	x_0	10	11	$\begin{pmatrix} 3.536 \\ 0.055 \\ 0.154 \end{pmatrix}$	$2.592 \cdot 10^{-4}$	388.377	27.870
	$5x_0$	FC	FC	-	-	-	-
Scaling	$5x_0$	30	38	$\begin{pmatrix} 3.536 \\ 0.055 \\ 0.154 \end{pmatrix}$	$8.548 \cdot 10^{-4}$	388.377	27.870

Table 3: Test results for the application problems, all optional values are set to default (except once scaling is added). FC is used as an abbreviation for failed to converge within 500 steps.

Until now the Levenberg-Marquardt method performed well and could solve all problems so far. The large-residual Problem 6 and especially its modification in Problem 7 were introduced to change this.
The results can be found in Table 4. As it can be seen, the Levenberg-Marquardt method is still able to solve Problem 6 in 24 steps and if $100x_0$ is used, then 34 iterations are required. Compared to the other problems, it required the highest amount of steps at the recommended starting value x_0. Nevertheless, it still converges. In Problem 7 this changes,

as the Levenberg-Marquardt method fails to converge with the adjustable parameters set to default. In particular the result highlights, that the combination of poor scaling and large residuals is problematic for our method.

Problem	Starting Value	Number of Iterations	Function Evaluations	Result x^*	$\|\nabla f(x^*)\|$	$f(x^*)$
6	x_0	24	37	$\begin{pmatrix} -11.594 \\ 13.204 \\ -0.403 \\ 0.237 \end{pmatrix}$	$8.726 \cdot 10^{-4}$	42911.101
	$10x_0$	33	46	$\begin{pmatrix} -11.594 \\ 13.204 \\ -0.403 \\ 0.237 \end{pmatrix}$	$4.101 \cdot 10^{-4}$	42911.101
	$100x_0$	34	49	$\begin{pmatrix} -11.594 \\ 13.204 \\ -0.403 \\ 0.237 \end{pmatrix}$	$6.286 \cdot 10^{-4}$	42911.101
7	x_0	FC	FC	-	-	-
	$10x_0$	FC	FC	-	-	-
	$100x_0$	FC	FC	-	-	-

Table 4: Test results for large-residual problems, all optional values are set to default. FC is used as an abbreviation for failed to converge within 500 steps.

Obviously the algorithm should perform better in Problem 7 if we include the scaling properties by adjusting the input parameters accordingly. As shown in Table 5 the Levenberg-Marquardt method can work fine. Note that when "Scaling" is set to "True" it is often useful to increase Δ and Δ_{max} as described in Section 4.1.1. For the starting value x_0 it converges, but it still needs 390 iterations. Compared to previous results this is the highest amount of iterations so far.
After adjusting the parameters and especially increasing "eps" slightly, the Levenberg-Marquart method even converges with the initial value $3x_0$ in 64 steps. However, if we use a step further away, the algorithm fails to converge or even breaks down. This is because the model function cannot be trusted and Δ is shrinked so much that it is interpreted as 0. Thus, the algorithm can only make very small steps, or it cannot find a step anymore.

Starting Value	Options	Number of Iterations	Function Evaluations
x_0	Delta = 1000, Delta_max= 10^6, Scaling = True	390	392
$3x_0$	Delta = 10^5, Delta_max= 10^6, eps= 0.1, Scaling = True	64	79
$5x_0$	Delta = 10^5, Delta_max= 10^6, eps= 0.1, Scaling = True	FC	FC

Table 5: Test results for Problem 7, when the scaling option is chosen.

All in all, the Levenberg-Marquardt method works particularly well for small-residual problems. This fits with the theoretical reasoning in Chapter 3. Nevertheless, it has two weaknesses: Poor scaling and large residuals. In the respective problems, this lead to slow convergence or in the worst case, the Levenberg-Marquardt method fails completely, as seen in Problem 7. However, by applying the scaling option in some cases, all problems could still be solved.

It is not a surprise that the Levenberg-Marquardt performs poorly if we solve a large-residual problem. As theoretically derived in Section 3.4.2, we cannot expect a fast convergence rate.
If we know in advance that we are dealing with a large-residual problem, it might be a better choice to use different methods, as recommended by Nocedal and Wright [NW06, p. 262]. An obvious choice would be to use Newton or quasi-Newton methods, depending on how expensive the second-order part in the Hessian (3) is to be obtained. This could be an improvement, because we would obtain at least a superlinear convergence rate.
If it is unknown whether the problem has large residuals around its minimizer x^*, Nocedal and Wright [NW06, p. 263] recommend to choose a *hybrid algorithm* as for example introduced in [Fle87]. Briefly summarized, a hybrid algorithm would then be implemented in a way that it can switch between a Levenberg-Marquardt method and a Newton or quasi-Newton method, depending on how large the residuals are [NW06, p. 263]. For more information about hybrid algorithms see for example [NW06, p. 262-265].
To conclude the discussion about large-residual problems it is important to note, that large residuals around the minimizer imply that the discrepancy between the chosen model and the obtained data is large. On the one hand, this might indicate that the model is inappropriate and a different choice of a model function would be better. On the other hand, it could imply that the process of measurement is erroneous.

Another common approach to solve least-squares problems is the Gauss-Newton method. It builds upon the same Hessian approximation (4) as the Levenberg-Marquardt method, but within a line search approach instead of a trust-region approach [NW06, p. 254]. After a short introduction, it is compared to our Levenberg-Marquardt method in the following chapter.

5 Comparison to Line Search Methods

> The ball is round, the game
> lasts ninety minutes, and
> everything else is just theory.
>
> Sepp Herberger

In this section the Levenberg-Marquardt method is tested against other algorithms which are used to minimize nonlinear least-squares problems (1). Albicker [Alb22] implemented the Gauss-Newton method and another Levenberg-Marquardt method. They have both in common that they use a line search approach. To avoid confusion, the implementation of Albicker is refered to as Levenberg-Marquardt variant, and the name Levenberg-Marquardt method is solely used for the presented algorithm in Chapter 3. After a short description of those algorithms, we focus on the numerical comparison. More detailed, the three algorithms are tested on the seven problems from Chapter 4 concerning number of iterations, number of function evaluations and overall runtime.

5.1 Line Search Methods

A line search method, like the trust-region approach, is also an iterative method. Given a starting value x_0, the next step for an iterate x_k is calculated by finding a suitable search direction p_k and a step length α_k which determines how far it is moved along p_k (see for a more detailed introduction Chapter 3 in [NW06]). We then have:

$$x_{k+1} = x_k + \alpha_k p_k, \tag{64}$$

as our next iterate. Usually, the search direction p_k is obtained by

$$p_k = -B_k^{-1} \nabla f_k, \tag{65}$$

where B_k is chosen, depending on the method, as an approximation to the Hessian $\nabla^2 f(x_k)$. In the Gauss-Newton method it is chosen $B_k = J(x_k)^T J(x_k)$ as we are using the same approximation as in the Levenberg-Marquardt method. Obviously, whenever $J(x_k)$ does not have full rank, the inverse does not exist and the method fails to calculate a search direction.
The Levenberg-Marquardt variant uses a slightly modified approximation

$$B_k = J(x_k)^T J(x_k) + \lambda_k I \qquad \text{for a } \lambda_k \geq 0.$$

Depending on λ_k the search direction interpolates between a Gauss-Newton and a steepest descent direction [CGT00, p. 178]. In the implementation by Albicker [Alb22] λ_{k+1} is updated in a way that it is increased by factor 2, whenever the new step fails to provide a smaller function value and decreased by 0.5 otherwise. By default we will choose $\lambda_0 = 1$.

Besides the search direction p_k, a suitable step length α_k has to be found. Here a Wolfe-Powell strategy can be used, like described for example in [NW06, p. 60]. However, in the Levenberg-Marquardt variant this process is skipped and it is always chosen $\alpha = 1$, and the step is only taken if it results in a decrease of the function value at the new iterate.
For more details it is recommended to see the master thesis of Albicker [Alb22]. An implementation of the two line search algorithms can be found in the optimization package *oppy* [Vol].

5.2 Numerical Comparison

In Table 6 the numerical results are given. All three algorithms perform well for the small-residual problems in Problem 1 and 2. The Levenberg-Marquardt method tend to be slightly less fast than the Gauss-Newton method and the Levenberg-Marquardt variant, which both need less than 1 miliseconds (ms). The number of iterations and function evaluations are quite similar in these two examples between the three methods.

The application problems also do not cause any difficulties for our three algorithms. Only in Problem 5 does the Gauss-Newton method need approximately twice the time as the other methods. A possible explanation can be given with regard to the function evaluations, where it needs more evaluations due to the search for the step length. All in all, the three algorithms are performing quite well so far.

This changes for the large-residual problem. Here the Gauss-Newton method fails to converge within 500 steps. This can be theoretically explained, because within the large-residual situation the approximation of the Hessian is inappropriate and thus the search direction in the Gauss-Newton method is inappropriate as well.
The Levenberg-Marquardt variant is able to increase λ_k in this situation and use as a consequence rather a steepest descent search direction. This makes the method more flexible and it requires only 30 steps to converge to the minimizer.
As already discussed in Chapter 4, the Levenberg-Marquardt method has no problem solving this problem, however it is slightly slower than the Levenberg-Marquardt variant.

Finally Problem 7, the combination of poor scaling and large residuals, is used. Obviously, the Gauss-Newton method fails to converge, because it has already failed to solve the Problem 6, which is similar to Problem 7 except for the poor scaling properties.
In this situation the Levenberg-Marquardt variant fails too. Here it can be again argued that the algorithm is able to use instead of a Gauss-Newton step rather a steepest descent step. Combined with poor scaling properties, this choice is also problematic because the steepest descent search direction is sensitive towards poor scaling [NW06, p. 27].
As already discussed, the Levenberg-Marquardt method still converges, but relatively slow when the scaling option is set to true. If the scaling option is not chosen than it fails to converge within 500 steps.

Consequently, in our examples, the Levenberg-Marquardt method, as compared to the other two algorithms, is the most robust one, especially because it can take poor scaling into account. The Gauss-Newton method fails to converge in Problem 6, because it depends on the appropriate approximation of the Hessian, which is not reasonable in the case of a large-residual problem. Another problem which did not occur in our examples in Table 6 is that as soon as J_k has no full rank, the Gauss-Newton method cannot find a new iterate, as described in Section 5.1. While the Levenberg-Marquardt variant is more flexible in this concern, when poor scaling properties are added to Problem 6, it fails as well.

Problem		Levenberg-Marquardt method	Gauss-Newton method	Levenberg-Marquardt variant
1	Iterations	9	7	10
	f evaluations	15	15	13
	Runtime (ms)	2.723	0.836	0.746
2	Iterations	5	5	6
	f evaluations	9	11	9
	Runtime (ms)	1.647	0.569	0.602
3	Iterations	5	5	12
	f evaluations	6	6	13
	Runtime (ms)	3.512	4.334	5.031
4	Iterations	7	10	18
	f evaluations	11	21	34
	Runtime (ms)	3.527	4.226	5.478
5	Iterations	10	12	10
	f evaluations	11	16	11
	Runtime (ms)	13.423	27.589	13.322
6	Iterations	24	FC	30
	f evaluations	37	FC	70
	Runtime (ms)	41.218	FC	38.386
7	Iterations	390*	FC	FC
	f evaluations	392*	FC	FC
	Runtime (ms)	482.654*	FC	FC

Table 6: Numerical results of the tests. The default values for the Levenberg-Marquardt are used, except in Problem 7, in which "Scaling" was set to "True" (the result is marked with *). In all three algorithms the same stopping criteria was used, as described in Section 3.3.2. FC is used for failed to converge in 500 steps.

6 Conclusion

The Levenberg-Marquardt method as a trust-region algorithm based on the works of Moré and Nocedal and Wright [Mor78; NW06] was implemented and successfully tested. Applied to seven test problems, the method performed robustly, as it could solve them all. This was not the case for the line search algorithms like the Gauss-Newton method and a Levenberg-Marquardt variant. Based on the test results, the Levenberg-Marquardt method performed as the most robust algorithm in the comparison.

In practical applications, two weaknesses may be relevant and could lead to a poor performance of the Levenberg-Marquardt method:

- Poor scaling
  This problem can be handled by altering the trust-region accordingly. The implementation of the Levenberg-Marquardt method has an option which should be used in this case such that the algorithm includes (approximately) the scaling properties. In the test problems, this option worked fine and proved to be useful for poorly scaled problems.

- Large residuals
  Large residuals around the minimizer x^* affect one of the key ingredients of the algorithm, the approximation of the Hessian as stated in (4). As a consequence the approximation turns out to be inappropriate and the algorithm converges slower, as in Problem 6, or even fails to converge completely.

A new modification of the test function introduced by Brown and Dennis [BD71] contains both, large residuals and poor scaling. The Levenberg-Marquardt method was able to solve the problem, however it needed a relatively high number of steps. With the example, it could be illustrated that the Levenberg-Marquardt variant based on a line search approach is less robust than the Levenberg-Marquardt method based on a trust-region approach, as the Levenberg-Marquardt variant fails to converge.

As a consequence of the comparison, we would recommend the usage of the Levenberg-Marquardt method, as implemented in Chapter 4, in order to solve least-squares problems. For large-residual problems however, the algorithm can be slow. Here other methods which do not rely on the approximation of the Hessian in (4) are expected to be more efficient. When it is known in advance that the problem has large residuals, than a better choice would be to use a Newton or quasi-Newton method, for example, as they guarantee at least a superlinear convergence rate.
If it is unknown, than hybrid algorithms which behave like a Levenberg-Marquardt method or a (quasi-) Newton method, depending on the size of the residuals, could provide a better option, as described in [NW06, p. 262-265].
As this thesis has built on the master thesis by Albicker about the Gauss-Newton method [Alb22], a next valuable step would be to implement such a hybrid algorithm, which can then be used to solve problems with large residuals more efficiently.

References

[Alb22] Julia Albicker. "The Gauß-Newton Method and its Implementation in the Optimization Library Oppy". MA thesis. 2022. URL: https://kops.uni-konstanz.de/handle/123456789/57727.

[BD71] Kenneth M. Brown and John E. Dennis. *New computational algorithms for minimizing a sum of squares of nonlinear functions*. report 71-6. Yale University: Department of Computer Science, 1971.

[BF10] Nicholas H. Bingham and John M. Fry. *Regression: Linear models in statistics*. Springer Science & Business Media, 2010. DOI: 10.1007/978-1-84882-969-5.

[CGT00] Andrew R. Conn, Nicholas IM Gould, and Philippe L. Toint. *Trust region methods*. SIAM, 2000. DOI: 10.1137/1.9780898719857.

[CPR12] Alfonso Croeze, Lindsey Pittman, and Winnie Reynolds. *Solving nonlinear least-squares problems with the Gauss-Newton and Levenberg-Marquardt methods*. 2012. URL: https://www.math.lsu.edu/system/files/MunozGroup1%20-%20Paper.pdf.

[DH19] Peter Deuflhard and Andreas Hohmann. *Numerische Mathematik 1: eine algorithmisch orientierte Einführung*. 5th ed. de Gruyter, 2019. DOI: 10.1515/9783110614329.

[DR11] Robert Denk and Reinhard Racke. *Kompendium der Analysis. Band 1: Differential- und Integralrechnung. Gewöhnliche Differentialgleichungen*. Vieweg+Teubner Verlag, 2011. DOI: 10.1007/978-3-8348-8184-7.

[Fle87] Roger Fletcher. *Practical methods of optimization*. 2nd ed. New York: John Wiley & Sons, 1987. DOI: 10.1002/9781118723203.

[Fou] Python Software Foundation. *Python*. Version 3.10.5. URL: https://www.python.org/.

[GMW21] Philip E. Gill, Walter Murray, and Margaret H. Wright. *Numerical linear algebra and optimization*. Classics in Applied Mathematics. Philadelphia: SIAM, 2021. DOI: 10.1137/1.9781611976571.

[HCM19] Robert V. Hogg, Allen T. Craig, and Joseph W. McKean. *Introduction to mathematical statistics*. 8th ed. Pearson, 2019.

[Heb73] M. D. Hebden. *An algorithm for minimization using exact second derivatives*. Tech. rep. TP515. Harwell: Atomic Energy Research Establishment, 1973.

[Him72] David M. Himmelblau. *Applied nonlinear programming*. McGraw-Hill, 1972.

[HL19] Frank Haußer and Yuri Luchko. *Mathematische Modellierung mit MATLAB® und Octave: Eine praxisorientierte Einführung*. Springer-Verlag, 2019. DOI: 10.1007/978-3-662-59744-6.

[HS16] Jürgen Hedderich and Lothar Sachs. *Angewandte Statistik: Methodensammlung mit R*. 17th ed. Springer, 2016. DOI: 10.1007/978-3-662-45691-0.

[Hue+04] S Huet et al. *Statistical Tools for Nonlinear Regression*. Springer Series in Statistics. New York: Springer-Verlag, 2004. DOI: 10.1007/b97288.

[Kel99] Carl T. Kelley. *Iterative methods for optimization*. Frontiers in applied mathematics. SIAM, 1999. DOI: 10.1137/1.9781611970920.

[Lev44] Kenneth Levenberg. "A method for the solution of certain non-linear problems in least squares". In: *Quarterly of applied mathematics* 2.2 (1944), pp. 164–168.

[LL15] Colin Lewis-Beck and Michael Lewis-Beck. *Applied Regression: An Introduction*. 2nd ed. Vol. 22. Quantitative Applications in the Social Sciences. SAGE Publications, 2015. DOI: 10.4135/9781483396774.

[Lyc20] Tom Lyche. *Numerical linear algebra and matrix factorizations*. Vol. 22. Texts in Computational Science and Engineering. Springer Nature, 2020. DOI: 10.1007/978-3-030-36468-7.

[Mar63] Donald W. Marquardt. "An algorithm for least-squares estimation of nonlinear parameters". In: *SIAM Journal on Applied Mathematics* 11.2 (1963), pp. 431–444. DOI: 10.1137/0111030.

[Mor78] Jorge J. Moré. "The Levenberg-Marquardt algorithm: Implementation and theory". In: *Numerical Analysis*. Ed. by G. A. Watson. Numerical Analysis 630. Springer Berlin Heidelberg, 1978, pp. 105–116. DOI: 10.1007/BFb0067700.

[NW06] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. 2nd ed. Springer Series in Operations Research and Financial Engineering. Springer New York, 2006. DOI: 10.1007/978-0-387-40065-5.

[Rat83] David Ratkowsky. *Nonlinear regression modelling*. New York: M. Dekker, 1983.

[Ros60] H. H. Rosenbrock. "An Automatic Method for Finding the Greatest or Least Value of a Function". In: *The Computer Journal* 3.3 (1960), pp. 175–184. DOI: 10.1093/comjnl/3.3.175.

[Sch22] Google Scholar. *The Levenberg-Marquardt algorithm: implementation and theory*. 2022. URL: https://scholar.google.de/citations?view_op=view_citation&hl=de&user=0uz45LcAAAAJ&citation_for_view=0uz45LcAAAAJ:nrtMV_XWKgEC.

[SW03] George AF Seber and Christopher John Wild. *Nonlinear Regression*. New Jersey: John Wiley & Sons, 2003. DOI: 10.1002/0471725315.

[UU12] Michael Ulbrich and Stefan Ulbrich. *Nichtlineare Optimierung*. Birkhäuser, 2012. DOI: 10.1007/978-3-0346-0654-7.

[Vol] WG Volkwein. *Oppy Documentation*. URL: https://www.mathematik.uni-konstanz.de/en/volkwein/python/oppy/index.html.

Appendix

QR Decomposition With Givens Rotations

In the following, the Python code in order to obtain a QR decomposition with Givens rotations is given as described in Section 4.1.2 and (25).

```
import numpy as np


def givens_qr(R_I, n,m):
        """
        Description
        ___________
        This function calculates the QR decomposition of a matrix
        of the form given in (25). It performs (n(n+1))/2 Givens
        rotations to get its QR decomposition. See for more
        information pp. 14-15 in this thesis.

        Parameters
        __________
        R_I : ndarray
             (m+n) x n matrix, which is already upper triangular,
             except for the elements (m+1,1), (m+2,2), ..., (m+n,n).
        n : int
             Number of columns of the matrix.
        m : int
             Number such that the rows of R_I are equal to m+n.

        Returns
        _______
        R_I : ndarray
             Rotated matrix R_I, which is now an
             upper triangular matrix.
        Q.T : ndarray
             (m+n) x (m+n) matrix. It is the transpose of the
             product of all rotations, which were used to rotate
             the input matrix R_I.

        """

        # list for the following loop
        rows = list(range(m+n,m,-1))

        # l states the number of rotations required in each row,
        l = 1

        # Q stores all the rotations, it will be altered accordingly
```

```
    Q = np.eye(n+m)

    # proceed like described on pp. 14-15
    for k in rows:
        for i in list(range(min(l,n),0,-1)):
            R_I, Q_1 = givens_rotation(R_I, k-1, n-i)
            Q = np.dot(Q_1,Q)
        # in the next row one additional rotation is required
        l +=1

    return R_I,Q.T



def givens_rotation(A, i ,j):
    """
    Description
    -----------
    This function calculates one Givens rotation,
    such that the given matrix A is rotated accordingly.
    As a consequence, the entry A[i,j] is then zero.
    It is based on the considerations about Givens
    rotations in [Lyc20, pp. 117-119].

    Parameters
    ----------
    A : ndarray
        Matrix for which one Givens rotation
        is calculated.
    i : int
        Row number of the entry which is after the
        rotation zero.
    j : int
        Column number of the entry which is after the
        rotation zero.

    Returns
    -------
    B : ndarray
        Givens rotated matrix B
    Q : ndarray
        Givens rotation, it is np.dot(Q,A)= B
    """
    n,m = A.shape

    #calculate the length of the vector (A[j,j],A[i,j])
    r = (A[j,j]**2+A[i,j]**2)**(0.5)
```

```
    c = A[j,j]/r
    s = A[i,j]/r

    # create the Givens rotation matrix Q
    Q = np.eye(n)
    Q[j,j]=c
    Q[i,i]=c
    Q[i,j]=-s
    Q[j,i]=s

    # B = QA, B[i,j]=0
    # (calculation without complete matrix multiplication)
    B = A.astype(float)
    S = np.array([[c, s],[-s, c]])
    C = A[(j,i),:]
    D = np.dot(S, C)
    B[j]= D[0]
    B[i]= D[1]

    return B,Q
```

List of Figures

List of Tables

Acknowledgements

I would like to start by thanking my supervisor Prof. Dr. Stefan Volkwein. His great support and feedback helped me a lot.
An adapted implementation of the Levenberg-Marquardt method can be found in the optimization package *oppy*. Here I am very grateful for the support of Christian Jäkle, who made this step possible.
Finally, I want to thank my family and all my friends who supported me during this thesis. Thank you!