





Article

RcdMathLib: An Open Source Software Library for Computing on Resource-Limited Devices

Zakaria Kasmi ¹, Abdelmoumen Norrdine ^{2,*}, Jochen Schiller ¹, Mesut Güneş ³ and Christoph Motzko ²

¹ Freie Universität Berlin, Department of Mathematics and Computer Science, Takustraße 9, 14195 Berlin, Germany; zakaria.kasmi@fu-berlin.de (Z.K.); jochen.schiller@fu-berlin.de (J.S.)

² Technische Universität Darmstadt, Institut für Baubetrieb, El-Lissitzky-Straße 1, 64287 Darmstadt, Germany; c.motzko@baubetrieb.tu-darmstadt.de

³ Otto-von-Guericke University, Faculty of Computer Science, Universitätsplatz 2, 39106 Magdeburg, Germany; mesut.guenes@ovgu.de

* Correspondence: a.norrdine@baubetrieb.tu-darmstadt.de

Abstract: We developed an open source library called RcdMathLib for solving multivariate linear and nonlinear systems. RcdMathLib supports on-the-fly computing on low-cost and resource-constrained devices, e.g., microcontrollers. The decentralized processing is a step towards ubiquitous computing enabling the implementation of Internet of Things (IoT) applications. RcdMathLib is modular- and layer-based, whereby different modules allow for algebraic operations such as vector and matrix operations or decompositions. RcdMathLib also comprises a utilities-module providing sorting and filtering algorithms as well as methods generating random variables. It enables solving linear and nonlinear equations based on efficient decomposition approaches such as the Singular Value Decomposition (SVD) algorithm. The open source library also provides optimization methods such as Gauss–Newton and Levenberg–Marquardt algorithms for solving problems of regression smoothing and curve fitting. Furthermore, a positioning module permits computing positions of IoT devices using algorithms for instance trilateration. This module also enables the optimization of the position by performing a method to reduce multipath errors on the mobile device. The library is implemented and tested on resource-limited IoT as well as on full-fledged operating systems. The open source software library is hosted on a GitLab repository.

Keywords: singular value decomposition; trilateration; Gauss–Newton; Levenberg–Marquardt; multipath recognition and mitigation; positioning; RIOT-OS; microcontrollers; embedded systems; Internet of Things



Citation: Kasmi, Z.; Norrdine, A.; Schiller, J.; Güneş, M.; Motzko, C. RcdMathLib: An Open Source Software Library for Computing on Resource-Limited Devices. *Sensors* **2021**, *21*, 1689. <https://doi.org/10.3390/s21051689>

Academic Editor: Raffaele Bruno

Received: 20 January 2021

Accepted: 22 February 2021

Published: 1 March 2021

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Copyright: © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Algorithms and scientific computing are workhorses of many numerical libraries that support users to solve technical and scientific problems. These libraries use mathematics and numerical algebraic computations, which contribute to a growing body of research in engineering and computational science. This leads to new disciplines and academic interests. The use of computers has accelerated the trend as well as enhanced the deployment of numerical libraries and approaches in scientific and engineering communities. Originally, computers were built for numerical and scientific applications. Konrad Zuse built a mechanical computer in 1938 to perform repetitive and cumbersome calculations. A specific problem, from the area of static engineering, requires performing tedious calculations to design load-bearing structures by solving systems of linear equations [1]. Howard Aiken independently developed an electro-mechanical computing device that can execute predetermined commands typed on a keyboard in the notation of mathematics and translated into numerical codes. These are stored on punched cards and perforated magnetic tapes or drums [2,3].

The first software libraries of numerical algorithms were developed in the programming language ALGOL 60 including procedures for solving linear systems of equations

or eigenvalue problems [4]. This software was rewritten in FORTRAN and ported to the LINPACK and EISPACK software packages [5,6]. Cleve Moler implemented a user-friendly interface to enable his students an easy access to LINPACK and EISPACK without writing Fortran programs [7]. He called the interface MATLAB (Matrix Laboratory), which was so successful that he founded a company called MathWorks. MATLAB is now a full-featured computing platform.

Ubiquitous computing on resource-limited devices has become an important issue in the Internet of Things (IoT) and the Machine to Machine (M2M) communication technologies, enabling the implementation of various applications such as health monitoring or vehicle tracking and detection. IoT is an emerging and challenging technology that facilitates the realization of computing services in various areas by using advanced communication protocols, technologies, and intelligent data analytical software [8]. The M2M communication in combination with the Radio-Frequency Identification (RFID), localization, observation by sensors, and controlling of actuators provide context-aware intelligent decisions as well as high-quality services. Computing plays a key role in implementing such applications, particularly by applying local and decentral processing on mobile and ubiquitous devices. This affords in-network and local context-aware decisions without the use of external computing services (e.g., cloud services). Therefore, we provide an open source software library for numerical linear algebra called [RcdMathLib](#) (Mathematical Library for Resource-constrained devices) [9]. This software library is suitable for devices with limited resources such as microcontrollers or portable computing devices. These devices are mostly low-cost, are equipped with low-end processors, and have limited memory and energy resources. [RcdMathLib](#) supports a decentralized and on-the-fly numerical computing locally on a mobile device. The decentralized numerical calculations allow for pushing the application-level knowledge into the mobile device and avoiding the communication as well as the calculation on a central unit such as a processing server. The decentralization allows for the reduction of latency and processing time as well as enhancing the real-time capability because the length of the path to be traveled from data are shortened. Furthermore, [RcdMathLib](#) provides useful algorithms such as the Singular Value Decomposition (SVD) which has become an indispensable tool in science and engineering [10]. SVD is applied for image compression and restoration or biomedical applications, for example, noise reduction of biomedical signals [11].

[RcdMathLib](#) allows for computing on mobile devices as well as embedded systems providing algorithms for the solution of linear and nonlinear multivariate system of equations. The solution of these equation systems is achieved by using robust matrix decomposition algorithms. The software library offers an optimization module for curve fitting or solving of problems of regression smoothing. Applications can be built and organized as modules using the [RcdMathLib](#), therefore, we offer a localization module. This is an application module for distance- and Direct-Current (DC)-pulsed, magnetic-based localization systems. The localization module allows for a position estimation of a mobile device. Localization enables the realization of context-aware computing applications in combination with mobile devices. In this sense, the software library enables the computation of the localization on mobile systems. [RcdMathLib](#) also enables the optimization of the estimated location by using an adaptive approach based on the SVD, Levenberg–Marquardt (LVM) algorithms, and the Position Dilution of Precision (PDOP) value [12–14]. In addition, the localization module provides an algorithm for the multipath detection and mitigation enables an accurate localization of the mobile device in Non-Line-of-Sight (NLoS) scenarios. [RcdMathLib](#) can be also used on a full-fledged device such as a Personal Computer (PC) or a computing server.

In this article, we will present the [RcdMathLib](#) as well as briefly address the difficulties by using linear algebra methods and the techniques to overcome the limitation of resource-constrained devices. Our main contributions are:

- An open source library for numerical computations on resource-limited devices and embedded systems. The software permits a user or a mobile device to solve multivari-

ant linear equation systems based on efficient algorithms such as the Householder or the Moore–Penrose inverse. The Moore–Penrose inverse is implemented by using the SVD method.

- A module for solving multivariant nonlinear equation systems as well as optimization and curve fitting problems on a resource-contained device on the basis of the SVD algorithm.
- A utilities-module provides various algorithms such as the Shell sort algorithm or the Box–Muller method to generate normally distributed random variables.
- A localization module for positioning systems that use distance measurements or DC-pulsed, magnetic signals. This module enables an adaptive, optimized localization of mobile devices.
- A software routine to locally reduce multipath errors on mobile devices.

Guckenheimer perceived that we are conducting increasingly complex computations built upon the assumption that the underlying numerical approaches are complete and reliable. Furthermore, we ignore numerical analysis by using mathematical software packages [15]. Thus, the user must be aware of the limitations of the algorithms and be able to choose appropriate numerical methods. The use of inappropriate methods can lead to incorrect results. Therefore, we briefly address certain difficulties by using linear algebra algorithms.

The remainder of this article is structured as follows: Firstly, we review related works in Section 2. We present the architecture as well as describe the modules of the software library in Section 3. We introduce the implementation issues in Section 4 and the usage of the [RcdMathLib](#) in Section 5. In Section 6, we evaluate the algorithms on a resource-limited device and on a low-cost, single-board computer. Finally, we conclude our article and give an outlook on future works in Section 7.

2. Related Work

Computing software especially for linear algebra is indispensable in science and engineering for the implementation of applications like text classification or speech recognition. To the best of our knowledge, there are few mathematical libraries for resource-limited devices, and most of them are limited to simple algebraic operations.

Libraries for numerical computation such as the GNU Scientific Library (GSL) are suitable for Digital Signal Processors (DSPs) or Linux-based embedded systems. For example, the commercially available Arm Performance Libraries (ARMPL) offer basic linear algebra subprograms, fast Fourier transform routines for real and complex data as well as some mathematical routines such as exponential, power, and logarithmic routines. Nonetheless, these routines do not support resource-constrained devices such as microcontrollers [16].

The C standard mathematical library includes mathematical functions defined in `<math.h>`. This mathematical library is widely used for microcontrollers, since it is a part of the C compiler. It provides only basic mathematical functions for instance trigonometric functions (e.g., \sin , \cos) or exponentiation and logarithmic functions (e.g., \exp , \log) [17].

Our research shows that there are very few attempts to build numerical computations libraries, which can run on microcontrollers: Texas Instruments® (Dallas, Texas, USA) provides for its MSP430 and MSP432 devices the IQmath and Qmath Libraries, which contain a collection of mathematical routines for C programmers. However, this collection is restricted only to basic mathematical functions such as trigonometric and algorithmic functions [18].

Libfixmatrix is a matrix computation library for microcontrollers. This library includes basic matrix operations such as multiplication, addition, and transposition. Equation solving and matrix inversion are implemented by the QR decomposition. Libfixmatrix is only suitable for tasks involving small matrices [19].

MicroBLAS is a simple, tiny, and efficient library designed for PC and microcontrollers. It provides basic linear algebra subprograms on vectors and matrices [20].

Other libraries for microcontrollers are the MatrixMath and BasicLinearAlgebra libraries. However, these libraries offer limited functionalities and are restricted to the Arduino platform [21,22].

The Python programming language is becoming widely used in scientific and numeric computing. The Python package NumPy (Numeric Python) is used for manipulating large arrays and matrices of numeric data [23]. The Scientific Python (SciPy) extends the functionality of NumPy with numerous mathematical algorithms [24]. Python is widely used for PC and single-board computers such as the Raspberry Pi. A new programming language largely compatible with Python called MicroPython is optimized to run on microcontrollers [25]. MicroPython includes the math and cmath libraries, which are restricted to basic mathematical functions. The mainline kernel of MicroPython supports only the ARM Cortex-M processor family. CircuitPython is a derivative of the MicroPython created to support boards like the Gemma M0 [26]. Various mathematical libraries are outlined in Table 1, which reveals their capabilities, limitations, and supported platforms.

Table 1. Comparison of mathematical libraries. FFT, Fast Fourier Transform.

Library	Capabilities	Platform
GNU Scientific Library (GSL)	Basic linear algebra, FFT, basic mathematical routines	No support for resource-limited devices
C standard mathematical library <math.h>	Basic mathematical functions	Support for resource-limited devices
IQmath and Qmath	Basic mathematical functions	MSP430 and MSP432 microcontrollers
Libfixmatrix	Basic mathematical functions and matrix inversion	ARM Cortex-M3 processors
MicroBLAS	Basic linear algebra	PC and microcontrollers
MatrixMath and BasicLinearAlgebra	Basic matrix operations	Restricted to the Arduino platform
NumPy, SciPy	Linear algebra computational mathematics	No support for resource-limited devices
math and cmath (MicroPython)	Basic mathematical functions	ARM Cortex-M processors or CircuitPython-powered boards
RcdMathLab	<i>mathematical functions for nonlinear, linear algebra optimization, and localization. Utilities</i>	<i>PCs and microcontrollers: Platforms using C compiler</i>

3. Library Architecture and Description

RcdMathLib has a pyramidal and a modular architecture as illustrated in Figure 1, whereby each layer rests upon the underlying layers. For example, the linear algebra module layer rests on the basic algebraic module layer. Each module layer is composed of several submodules such as the matrix or vector submodules. The submodules can be built up from the underlying submodules, for example, the pseudo-inverse submodule is based

on the SVD, the Householder, and the Gevins submodules. For the sake of brevity, Figure 1 presents only the sublayers. The software layers will be briefly addressed in Section 3.1 through Section 3.3.

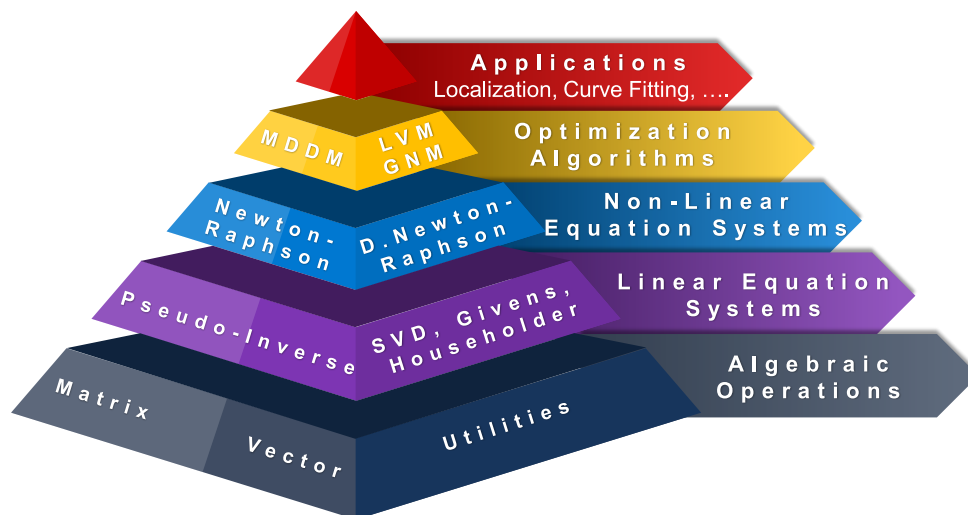


Figure 1. Architecture of the RcdMathLib. SVD, Singular Value Decomposition. D. Newton-Raphson, Damped Newton-Raphson. LVM, Levenberg–Marquardt Method. GNM, Gauss–Newton Method. MDDM, Multipath Distance Detection and Mitigation.

3.1. Linear Algebra Module Layer

The module layer of linear algebra is composed of the following submodules:

- Basic operations submodule: provides algebraic operations such as addition or multiplication of vectors or matrices. This submodule distinguishes between vector and matrix operations.
- Matrix decomposition submodule: allows for the decomposition of matrices by using algorithms such as Givens, Householder, or the SVD. The SVD method is implemented using the Golub–Kahan–Reinsch algorithm [27,28].
- Pseudo-inverse submodule: enables the computation of the inverse of quadratic as well as of rectangular matrices. The matrix inverse can be calculated by using the Moore–Penrose, Givens, or Householder algorithms [29].
- Linear solve submodule: permits the solution of under-determined and over-determined linear equation systems. We solve the linear equation systems using two matrix decompositions: the SVD and QR factorizations. The first method uses the Moore–Penrose inverse, while the second approach applies the Householder or the Givens algorithms with the combination of the back substitution method. We also provide the Gaussian Elimination (GE) with a pivoting algorithm, which is based on the LU decomposition. We use the GE-based method only for testing purposes or for devices with very limited stack memory. We suggest using the SVD- or the QR-based methods due to the numerical stability and the support of non-quadratic matrices [30].
- Utilities submodule: offers filtering algorithms such as median, average, or moving average. Furthermore, it provides the Shell algorithm to put elements of a vector in a certain order as well as the Box–Muller method to generate normally distributed random variables [31,32].

3.2. Non-Linear Algebra Module Layer

The nonlinear algebra module includes the following submodules:

- Optimization submodule: enables the optimization of an approximate solution by using Nonlinear Least Squares (NLS) methods such as modified Gauss–Newton (GN) or the LVM algorithms. These methods are iterative and need a start value as an

approximate solution. Moreover, the user should give a pointer to an error function and to a Jacobian matrix. The modified GN and the LVM algorithms will be briefly described in Sections 3.2.1 and 3.2.2.

- Nonlinear equations submodule: allows for the solution of multivariate nonlinear equation systems by using Newton–Raphson and damped Newton–Raphson methods [33]. The user must deliver a start value as well as a pointer to nonlinear equation systems to solve, and a pointer to the appropriate Jacobian matrix.

3.2.1. Gauss–Newton Algorithm

The Gauss–Newton algorithm works iteratively to find the solution \vec{x} that minimizes the sum of the square errors. During the iteration process, we cache the value of \vec{x} with the minimal sum of squares to prevent the divergence of the GN algorithm [34]. The computed solution by the GN can be used as a start value for the subsequent LVM algorithm if the start value is unknown or the GN algorithm diverges.

3.2.2. Levenberg–Marquardt Algorithm

The LVM algorithm is also a numerical optimization approach enabling solving NLS problems [35–37]. The LVM algorithm can be used for optimization or fitting problems. The LVM method proceeds iteratively as follows:

$$\vec{x}^{(k+1)} = \vec{x}^{(k)} + \vec{s}^{(k)}, \quad (1)$$

where $\vec{x}^{(k)}$ is the k -th approximation of the searched solution and $\vec{s}^{(k)}$ is the k -th error correction vector. The LVM improves the approximate solution \vec{x}_0 in each iteration step by calculating the correction vector $\vec{s}^{(i)}$ as follows [38,39]:

$$(J_f^T(\vec{x}^{(i)})J_f(\vec{x}^{(i)}) + (\mu^{(i)})^2I)\vec{s}^{(i)} = -J_f^T(\vec{x}^{(i)})\vec{f}(\vec{x}^{(i)}), \quad (2)$$

where μ is the damping parameter, f is the error function, and J_f is the Jacobian matrix. The LVM algorithm has the advantage over the GN method because the matrix on the left side of Equation (2) is no longer singular. This is accomplished by the factor μ^2I regulating the matrix $J_f^T J_f$. The LVM method is described in Algorithm 1.

Algorithm 1 LVM algorithm

```

1: function LVM_ALG( $\epsilon_x, \beta_0, \beta_1, \tau, i_{max}, \vec{x}^{(0)}, \vec{f}, J_f$ )
2:    $i = 0; \vec{x} = \vec{x}^{(0)}; B = J_f^T(\vec{x})J_f(\vec{x}); \vec{H} = J_f^T(\vec{x})\vec{f}(\vec{x});$ 
3:    $\mu^{(0)} = \tau \cdot \max_i \{b_{ii}(\vec{x})\}; \mu = \mu^{(0)};$ 
4:   Solve  $(B + \mu^2I)\vec{s} = -\vec{H}$  for  $\vec{s}$ ;
5:   while  $(\|\vec{s}\|_2 > \epsilon_x(1 + \|\vec{x}\|_2)$  and  $(i < i_{max})$  do
6:      $[\vec{s}, \mu] = \text{CORRECTION\_FUNC}(\vec{x}, \mu, \beta_0, \beta_1);$ 
7:     while (true) do
8:       if  $(\rho_\mu \leq \beta_0)$  then
9:          $\mu = 2\mu$ 
10:         $[\vec{s}, \rho_\mu] = \text{CORRECTION\_FUNC}(\mu, \beta_0, \beta_1, \vec{x}, \vec{f}, J_f);$ 
11:       else if  $(\rho_\mu \geq \beta_1)$  then
12:          $\mu = \frac{\mu}{2}$ 
13:         break;
14:       else
15:         break;
16:       end if
17:     end while
18:      $\vec{x} = \vec{x} + \vec{s};$ 
19:      $i = i + 1;$ 
20:   end while
21: end function
22: function CORRECTION_FUNC( $\mu, \beta_0, \beta_1, \vec{x}, \vec{f}, J_f$ )
23:    $B = J_f^T(\vec{x})J_f(\vec{x}); \vec{H} = J_f^T(\vec{x})\vec{f}(\vec{x});$ 
24:   Solve  $(B + \mu^2I)\vec{s} = -\vec{H}$  for  $\vec{s}$ ;
25:    $\rho_\mu = \frac{\|\vec{f}(\vec{x})\|_2^2 - \|\vec{f}(\vec{x} + \vec{s})\|_2^2}{\|\vec{f}(\vec{x})\|_2^2 - \|\vec{f}(\vec{x}) + J_f(\vec{x})\vec{s}\|_2^2};$ 
26: end function

```

3.3. Localization Module Layer

Localization of users or IoT devices is indispensable for Localization-Based Services (LBSs) such as tracking in smart buildings, advertising in shopping centers, or routing and navigation in large public buildings [40]. Indoor Localization Systems (ILSs) are used to locate IoT or mobile devices inside environments where the Global Positioning System (GPS) cannot be deployed. Numerous technologies have been evaluated for ILSs, for example, Ultra-Wideband (UWB) [41], Wireless Local Area Network (WLAN) [42], ultrasound [43], magnetic signals [44], or Bluetooth [45].

3.3.1. Introduction and Layer Description

The Localization Module (LM) layer provides algorithms to calculate as well as optimize a position of an ILS. At the current stage of development, we provide two example applications of the LM as submodules: a distance-based as well as a DC-pulsed, magnetic-based submodule of ILSs. We also offer a common positioning submodule comprising shared algorithms like the trilateration algorithm [46].

Figure 2 illustrates the principle of a distance-based ILS composed of four anchors with known positions and one mobile device. The distances between the anchors and the mobile device can be measured by using UWB or ultrasound sensors. The mobile device performs distance measurements to the four anchors. Furthermore, the collected distances are preprocessed using the median filter from the utilities submodule to remove outliers (see Section 3.1). Finally, the mobile device locally calculates a three-dimensional position using the trilateration algorithm provided by the [RcdMathLib](#).

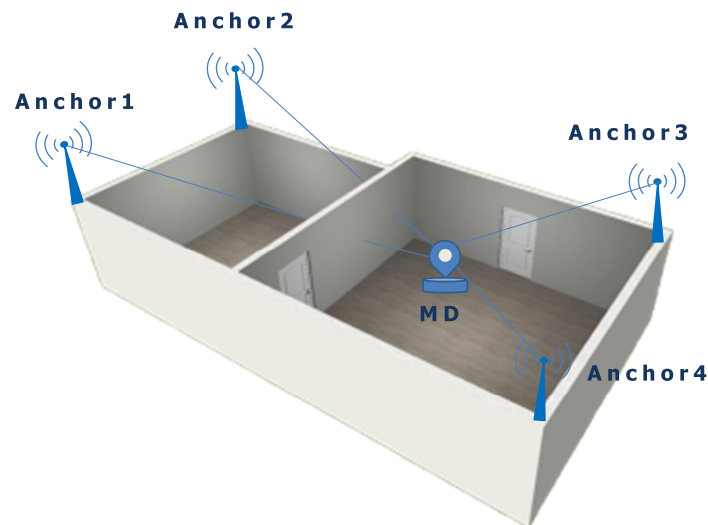


Figure 2. Distance-based Indoor Localization System. MD, Mobile Device.

The trilateration algorithm computes the position of an unknown point with the coordinates (x, y, z) and distances d_i to the reference positions (x_i, y_i, z_i) for $i = 1, 2, \dots, n$. This problem requires the estimation of a vector $\vec{x} = (w, x, y, z)$ such that:

$$A\vec{x} = \vec{b}, \quad (3)$$

where the matrix A and the vector \vec{b} have the following forms [46–48]:

$$A = \begin{bmatrix} 1 & -2x_1 & -2y_1 & -2z_1 \\ 1 & -2x_2 & -2y_2 & -2z_2 \\ 1 & -2x_3 & -2y_3 & -2z_3 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & -2x_n & -2y_n & -2z_n \end{bmatrix} \text{ and} \quad (4)$$

$$\vec{b} = \begin{bmatrix} d_1^2 - x_1^2 - y_1^2 - z_1^2 \\ d_2^2 - x_2^2 - y_2^2 - z_2^2 \\ d_3^2 - x_3^2 - y_3^2 - z_3^2 \\ \vdots \\ d_n^2 - x_n^2 - y_n^2 - z_n^2 \end{bmatrix}. \quad (5)$$

The solution \vec{x} is given by:

$$\vec{x} = A^+ \vec{b}, \quad (6)$$

where A^+ is the pseudo-inverse of the matrix A . The pseudo-inverse matrix A^+ is computed by using the pseudo-inverse submodule of the [RcdMathLib](#) (see Section 3.1). The quality q of the calculated position \vec{x} is given by:

$$q = w - (x^2 + y^2 + z^2). \quad (7)$$

Figure 3 illustrates the principle of a magnetic-based ILS composed of various coils with known positions and a mobile device. This system enables the calculation of the position of the mobile device by measuring magnetic fields to the coils as anchors. The magnetic signals are artificially generated from the coils using a pulsed direct current. The collected measurement data are preprocessed from the utilities-submodule for removing outliers and calibrating magnetic data. Finally, the position is calculated on the mobile device.

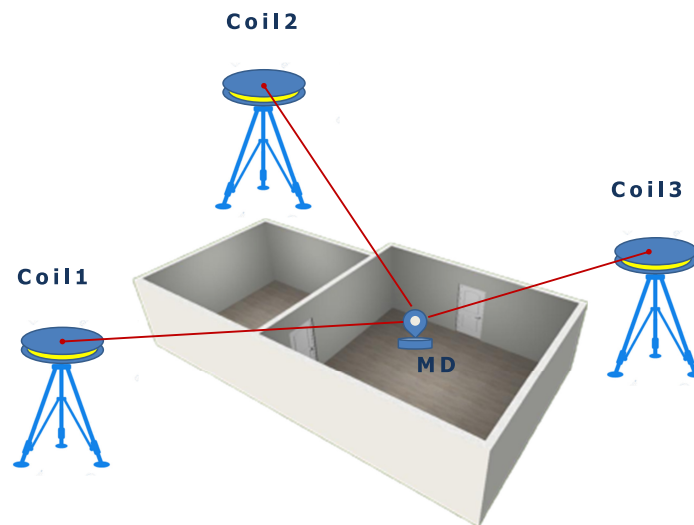


Figure 3. Magnetic-based indoor localization system. MD, Mobile Device.

The magnetic field B_i generated from the coil i is equal to [47,49]:

$$B_i = \frac{K}{r_i^3} \sqrt{1 + 3 \sin^2(\theta_i)} \quad i = 1, 2, \dots, n. \quad (8)$$

In this setting, $K = \frac{\mu_0 N_t I F}{4\pi}$, where N_t describes the number of turns of the wire, I is the current running through the coil, F expresses the base area of the coil, μ_0 is the permeability of free space, r_i is the distance between the mobile device and coil i , and θ_i is the mobile device elevation angle relative to the coil plane. The distance r_i and the elevation angle θ_i are equal to:

$$r_i = \sqrt{(x - x_i)^2 + (y - y_i)^2 + (z - z_i)^2} \quad (9)$$

$$\sin \theta_i = \frac{z - z_i}{r_i}. \quad (10)$$

Equation (8) is a nonlinear equation system with the unknowns coordinates x , y and z , which can be solved by applying the LVM algorithm from the optimization-submodule of the [RcdMathLib](#), whereby, (x_i, y_i, z_i) and (x, y, z) are the coordinates of the i -th coil and the mobile device.

3.3.2. Multipath Distance Detection and Mitigation and Position Optimization Algorithm

The location module is not restricted to simple position calculations but rather performs complex tasks such as the Multipath Distance Detection and Mitigation (MDDM) by the usage of other modules of the [RcdMathLib](#). The MDDM algorithm enables to reduce the effects of multipath fading on digital signals in radio environments of a mobile device to Reference Stations (RSs) with known locations. The MDDM approach is based on the Robust Position Estimation in Ultrasound Systems (RoPEUS) algorithm [50]. The MDDM is adapted for precise distance-based ILSs such as the Ultra-Wideband (UWB)-based localization systems, whereas it is simultaneously optimized for resource-limited devices [12]. The MDDM algorithm is summarized in Algorithm 2.

Algorithm 2 MDDM algorithm

```

1: function RECOG_MITIGATE_MULTIPATH_ALG( $k, n, threshold, d, RS$ )
2:    $j = 0; r_{min} = \infty;$ 
3:   distances  $d_i, i = 0, \dots, n - 1;$  ▷ distance measures to  $n$  RSs
4:   while ( $j < \binom{n}{k}$ ) do
5:      $comb(k) = RS_0 \dots RS_{k-1};$  ▷ choose  $k$  RSs
6:      $\vec{x}_k;$  ▷ Compute a position related to  $k$  RSs
7:      $r_i = R_i - d_i;$  ▷ residuals
8:      $r = \sum_{i=0}^{n-1} r_i;$ 
9:      $r_{min} = \min(r, r_{min});$ 
10:     $j = j + 1;$ 
11:  end while
12:   $\vec{x}_0 = \arg \min_{r_{min}}$  ▷ the solution
13:   $PDOP_x$  ▷ calculate PDOP-value of  $\vec{x}$ 
14:  if ( $PDOP_x > threshold$ ) then
15:    LVM_ALG( $\epsilon_x, \beta_0, \beta_1, \tau, i_{max}, \vec{x}_0, \vec{f}, J_f$ ) ▷ optimize the position
16:  end if
17: end function

```

3.4. Documentation and Examples' Modules

[RcdMathLib](#) includes a module that provides an Application Programming Interface (API) documentation. The API documentation is in Portable Document Format (PDF) and in Hypertext Markup Language (HTML) format. It is generated from the C source code by using the Doxygen tool [51]. The software reference documentation covers the description of the implemented functions as well as their passing parameters. In addition, the example module comprises samples of each module to familiarize the users with the API. The example module has the same structure as the [RcdMathLib](#).

4. Implementation Issues

Given a $(m \times n)$ non-singular matrix A and an n -vector \vec{b} , the fundamental problem of linear algebra is to find an n -vector \vec{x} such that $A\vec{x} = \vec{b}$. This fundamental problem emerges in various areas of science and engineering such as applied mathematics, physics, or electrical engineering [52]. Associated problems are finding the inverse, the rank, or projections of a matrix A . Attempting to solve the linear algebra problem using common theoretical approaches would face computational difficulties. For example, solving a $(20, 20)$ linear system with the Cramer's Rule, which is a significant theoretical algorithm, might take more than a million years even by using fast computers [52].

We use the Givens, the Householder, and the SVD matrix decomposition algorithms. These decomposition methods enable the solution of various problems such as the computation of the inverse matrix, the linear equations, or the rank of a matrix. We do not use the Cholesky decomposition (AA^T), since it can become unstable due to the rounding errors that are equal to $[\kappa(A)]^2$ instead of $\kappa(A)$. Although the Gaussian Elimination (GE)

is an efficient algorithm to implement the LU factorization, we do not use it, since GE generally requires pivoting and is limited to square matrices. Furthermore, GE can be unstable in certain contrived cases; nonetheless, it performs well for the majority of practical problems [53,54].

We do not use the Classical Gram–Schmidt (CGS) and the Modified Gram–Schmidt (MGS) to implement the QR decomposition due to the numerical instability. Instead, we use the Householder and the Givens algorithms. Even though the Householder is more efficient than the Givens algorithm, the Givens method is easy to parallelize. The Givens and the Householder methods have a guaranteed stability but fail if the matrix is nearly rank-deficient or singular. The SVD algorithm can be used to avoid the rank deficiency problem. This algorithm is not explicitly computable by determining the eigenvalues of the symmetric matrix $A^T A$ due to the round-off errors in the calculation of the matrix $A^T A$. Therefore, we implement the SVD by using the Golub–Kahan–Reinsch (GKR) algorithm, which will be described in Section 4.1.

We calculate the pseudo-inverse matrix by using the Moore–Penrose method based on the SVD or the QR decomposition using the Householder or the Givens algorithms. The QR-based pseudo-inverse A^+ is computed as follows:

$$A = QR, \quad (11)$$

$$A^+ = A^{-1} = (QR)^{-1} = R^{-1}Q^{-1}, \quad (12)$$

$$A^+ = R^{-1}Q^T, \quad (13)$$

where R^{-1} is the inverse of an upper triangular matrix, and Q^T is the transpose of an orthogonal matrix. We calculate the R^{-1} matrix by using Algorithm 3 [55].

Algorithm 3 Inverse of an upper triangular matrix

```

1: function INV_UPPER_TRIANG_MATRIX_ALG(n, U)
2:    $U_{inv}$ ; ▷ holds the calculated inverse of the matrix U
3:   for  $i = 0$  to  $n - 1$  do
4:      $U_{inv}[i, i] = 1/U[i, i]$ ;
5:     for  $j = 0$  to  $i - 2$  do
6:        $U_{inv}[j, i] = -U[i, i](U[j, j : i - 2]U[j : i - 2, i])$ ;
7:     end for
8:   end for
9: end function

```

In general, we avoid the explicit calculation of matrix multiplications such as the construction of Householder matrices ($H_i A$) or of Givens matrices ($J_i A$). The calculated triangular matrix R is stored over the matrix A . We also provide functions to avoid the explicit computation and storage of the transpose matrix such as the function that implicitly calculates the matrix–transpose–vector multiplication ($A^T \vec{x}$). We use the SVD algorithm to overcome the rank deficiency problem, for example, by the modified GN, the Newton–Raphson, and damped Newton–Raphson methods. We used the Householder instead of the SVD method by the LVM algorithm to save computing time and memory stack. This optimization is possible because of the robustness of the LVM algorithm (see Section 3.2.2). We provide the iterative Shell sort algorithm that is suitable for resource-limited devices with a limited stack size. We use the Shell sort algorithm to implement the median filter.

4.1. Singular Value Decomposition

The SVD method has become a powerful tool for solving a wide range of problems in different application domains such as biomedical engineering, control systems, or signal processing [52]. We implemented the SVD approach based on the Golub–Kahan–Reinsch algorithm that works in two phases: a bidiagonalization of the matrix A and the reduction of the calculated bidiagonal matrix to a diagonal form.

First phase (bidiagonalization)

A ($m \times n$) matrix A is transformed to an upper bidiagonal matrix $B \in \mathbb{R}^{m \times n}$ by using

the Householder bidiagonalization, where $m \geq n$. The matrix A is transformed as follows:

$$U_0^T A V_0^T = \begin{bmatrix} B \\ 0 \end{bmatrix}, \quad (14)$$

where B is an $n \times n$ bidiagonal matrix equal to

$$\begin{bmatrix} b_{11} & b_{12} & \dots & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \dots & \ddots & b_{n-1,n} \\ 0 & 0 & 0 & b_{n,n} \end{bmatrix}. \quad (15)$$

Second phase (reduction to the diagonal form)

The bidiagonal matrix B is further reduced to a diagonal matrix Σ by using orthogonal equivalence transformations as follows:

$$U_1^T B V_1^T = \Sigma = \text{diag}(\sigma_1, \sigma_2, \dots, \sigma_n), \quad (16)$$

where Σ is the matrix of the singular values σ_i and the matrices U_1 and V_1 are orthogonal. The singular vector matrices can be computed as follows:

$$U = U_0 U_1, \quad (17)$$

$$V = V_0 V_1. \quad (18)$$

We implemented the first and second phases by the Golub–Kahan bidiagonal procedure and the Golub–Reinsch algorithm. Both algorithms will be described in detail in Sections 4.1.1 and 4.1.2. In this description, we will mention some implementation issues.

4.1.1. Golub–Kahan Bi-Diagonal Procedure

The reduction of matrix A to the upper bi-diagonal matrix B is accomplished by using a sequence of Householder reflections, where the matrix B has the same set of singular values as the matrix A [56]. First, a Householder transformation U_{01} is applied to zero out the sub-diagonal elements of the first column of the matrix A . Next, a Householder transformation V_{01} is used to zero out the last $(n - 2)$ elements of the first row by post-multiplying the matrix $U_{01}A$: $U_{01}AV_{01}$. Repeating these steps a total of n times, the matrix A will be transformed to:

$$B = (U_n U_{n-1} \dots U_1 U_0) A (V_n V_{n-1} \dots V_1 V_0). \quad (19)$$

4.1.2. Golub–Reinsch Algorithm

The Golub–Reinsch algorithm is a variant of the QR iteration [28]. At each iteration i , the implicit symmetric QR algorithm is applied with the Wilkinson shift without forming the product $B_i^T B_i$. The algorithm has guaranteed convergence with a quite fast rate [52]. Starting from the bi-diagonalization of the matrix A obtained from the previous Golub–Kahan bi-diagonal procedure, the algorithm creates a sequence of bi-diagonal matrices $\{B_i\}$ with possibly smaller off-diagonals than the previous one. For simplicity, we write:

$$B = \begin{bmatrix} \alpha_1 & \beta_2 & & & \\ & \ddots & \ddots & & \\ & & \ddots & \ddots & \\ & & & \ddots & \beta_n \\ & & & & \alpha_n \end{bmatrix} \quad (20)$$

We calculate the Wilkinson shift σ that is equal to the eigenvalue λ of the right-hand corner sub-matrix of the matrix $C = B_i^T B_i$:

$$\begin{bmatrix} c_{n-1,n-1} & c_{n-1,n} \\ c_{n-1,n} & c_{n,n} \end{bmatrix} = \begin{bmatrix} \alpha_{n-1}^2 + \beta_{n-1}^2 & \alpha_{n-1}\beta_n \\ \alpha_{n-1}\beta_n & \alpha_n^2 + \beta_n^2 \end{bmatrix}, \quad (21)$$

which is closer to $\alpha_n^2 + \beta_n^2$. G. H. Golub and C. F. Van Loan suggest to calculate the Wilkinson shift as follows [28]:

$$\delta = \frac{c_{n-1,n-1} - c_{n,n}}{2}, \quad (22)$$

$$\sigma = c_{n,n} - \frac{\text{sign}(\delta)c_{n-1,n}^2}{|\delta| + \sqrt{\delta^2 + c_{n-1,n}^2}}. \quad (23)$$

We calculate c_1 and s_1 such that:

$$\begin{bmatrix} c_1 & s_1 \\ -s_1 & c_1 \end{bmatrix}^T \begin{bmatrix} \alpha_1^2 - \sigma \\ \alpha_1\beta_2 \end{bmatrix} = \begin{bmatrix} * \\ 0 \end{bmatrix}, \quad (24)$$

and form the Givens rotation V_1 .

We apply the Givens rotation V_1 to the right of the matrix B :

$$BV_1 = \begin{bmatrix} * & * & & & \\ & * & * & & \\ & & * & * & \\ & & & * & * \\ & & & & * \end{bmatrix} V_1 = \begin{bmatrix} * & * & & & \\ + & * & * & & \\ & & * & * & \\ & & & * & * \\ & & & & * \end{bmatrix}. \quad (25)$$

The bidiagonal form is destroyed by the unwanted non-zero element (bulge) indicated by the “+” sign. Therefore, we apply the Givens rotations $U_1, V_2, U_2, \dots, V_{n-1}$, and U_{n-1} to chase the badges.

We apply a Givens transformation U_1 to the left of the matrix BV_1 to eliminate the unwanted sub-diagonal element. This reintroduces a badge in the first row to the right of the super-diagonal element:

$$U_1BV_1 = \begin{bmatrix} * & * & + \\ 0 & * & * \\ & & * & * \\ & & & * \end{bmatrix}. \quad (26)$$

We apply the Givens rotations V_2 to remove the badge in the matrix U_1BV_1 . This introduces a new badge into the sub-diagonal of the third row, which is eliminated by the Givens rotation U_2 :

$$U_2U_1BV_1V_2 = \begin{bmatrix} * & * & 0 \\ & * & * \\ + & * & * \\ & & * \end{bmatrix} = U_2 \begin{bmatrix} * & * & & \\ & * & * & + \\ 0 & * & * & \\ & & & * \end{bmatrix}. \quad (27)$$

The matrix pair (V_3, U_3) terminates the chasing process and delivers a new bi-diagonal matrix \tilde{B} :

$$\tilde{B} = U_3U_2U_1BV_1V_2V_3. \quad (28)$$

In general, the chasing process creates a new bi-diagonal matrix \tilde{B} that is related to the matrix B as follows [28]:

$$\tilde{B} = (U_{n-1} \dots U_1)B(V_1 V_2 \dots V_{n-1}) = \tilde{U}B\tilde{V}, \tag{29}$$

where \tilde{U} and \tilde{V} are orthogonal. During the chasing process, we distinguish between the splitting, the cancellation, and the negligibility steps [57]:

At the i -th iteration, we assume that the matrix \tilde{B} is equal to:

$$\tilde{B}_i = \begin{bmatrix} q_1 & e_2 & & & \\ & \ddots & \ddots & & \\ & & q_{n-1} & e_n & \\ & & & q_n & \end{bmatrix}. \tag{30}$$

Splitting: If the matrix entry e_i is equal to zero, we split the matrix \tilde{B}_i into two block diagonal-matrices whose singular values can be computed independently:

$$\tilde{B}_i = \begin{bmatrix} \tilde{B}_1 & 0 \\ 0 & \tilde{B}_2 \end{bmatrix}, \tag{31}$$

$$svd(\tilde{B}_i) = svd(\tilde{B}_1) + svd(\tilde{B}_2), \tag{32}$$

where $svd(\tilde{B}_i)$ is the singular value decomposition of the matrix \tilde{B}_i ; in this case, we compute the matrix \tilde{B}_2 first. If the split occurs at i equal to n , then the matrix \tilde{B}_2 is equal to q_n and q_n is a singular value.

Cancellation: If the matrix entry q_i is equal to zero, we split the matrix \tilde{B}_i by using Givens rotations from the left to zero out row i as follows:

$$G_{i,i+1}^T \begin{bmatrix} q_1 & e_2 & & & & & & & \\ & \ddots & \ddots & & & & & & \\ & & q_{i-1} & e_i & & & & & \\ & & & 0 & e_{i+1} & & & & \\ & & & & q_{i+1} & \ddots & & & \\ & & & & & \ddots & e_n & & \\ & & & & & & q_n & & \end{bmatrix} = \begin{bmatrix} q_1 & e_2 & & & & & & & \\ & \ddots & \ddots & & & & & & \\ & & q_{i-1} & e_i & & & & & \\ & & & 0 & 0 & b & & & \\ & & & & q_{i+1} & \ddots & & & \\ & & & & & \ddots & e_n & & \\ & & & & & & q_n & & \end{bmatrix}, \tag{33}$$

whereas the budge b is removed by using the Givens rotations for $k = i + 2, \dots, n$. Since the matrix element e_{i+1} is equal to zero, the matrix splits again into two block diagonal sub-matrices (see the splitting step).

Negligibility: The values of the matrix elements e_i or q_i will be small but not exactly zero due to the finite precision arithmetic used by digital processors. Therefore, we require a threshold to decide when the elements e_i or q_i can be considered zero. Golub and Reinsch [58] recommend the following threshold rule:

$$|e_{i+1}|, |q_i| \leq \varepsilon \max_i (|q_i| + |e_i|) = \varepsilon \|B\|_1, \tag{34}$$

where ε is the machine precision. Björck [27] suggests the following approach:

$$|e_{i+1}| \leq 0.5\varepsilon(|q_i| + |q_{i+1}|), \quad (35)$$

$$|q_{i+1}| \leq 0.5\varepsilon(|e_i| + |e_{i+1}|). \quad (36)$$

Linpack [5] uses a variant Björck's approach that omits the factor 0.5 in Equations (35) and (36).

5. Usage of the RcdMathLib

RcdMathLib can be used on PCs, resource-constrained devices such as microcontrollers, or on small single-board computers like Raspberry Pi devices. It is a free software and available under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, version 2.1 (LGPLv2.1) [59]. The **RcdMathLib** software is written in the C programming language by using the GNU Compiler Collection (GCC) for full-fledged devices and embedded tool chains for resource-limited devices; for instance, the GNU ARM Embedded Toolchain. **RcdMathLib** can also be used on top of an Operating System (OS) for resource-constrained devices with a minimal effort due to the modular architecture of the library. We support the RIOT-OS, which is an open source IoT OS [60]. **RcdMathLib** is interfaced with the RIOT-OS using the GNU Make utility, whereby the user only needs to choose the modules needed by setting the USE_MODULE-macro. We automatically calculate the dependencies of the modules needed and the user can choose between a floating-point single-precision or double-precision depending on the available stack memory. An OS for resource-limited devices is recommended for the use of the **RcdMathLib**, but it is not required. A minimum stack size of 2560 bytes is needed to compute with floating-point numbers. The printf() function needs extra memory stack, therefore a minimum stack size of 4608 bytes is required to work with double-precision for floating-point arithmetic. We recommend a stack size of 8192 bytes.

The Linaro toolchain can be used on Linux or Windows to build applications for a target platform [61]. The OpenOCD can be used for flashing the code to the target (chip) as well as for low level or source level debugging [62]. The source code as well as the documentation (APIs) of the **RcdMathLib** can be downloaded from the GitLab repository [63]. The wikis are available on the homepage of the library to get started with the **RcdMathLib** [9].

Simple Example

We present a simple example to demonstrate how to use the **RcdMathLib** by defining two (3,4) matrices in Listing 1. We create a matrix with specific values equal to π in the main diagonal by calling the "matrix_get_diag_mat()" function. In the next step, we calculate the transpose of the second matrix by invoking the "matrix_get_transpose()" function. Finally, we calculate the multiplication of the first matrix with the transpose of the second matrix by executing the "matrix_mul()" function. In these three cases, the user should deliver the dimension of the matrices as well as a reference to the matrix resulted. The outputs of the calculated results are presented in Listing 2.

Listing 1. Example of basic matrix algebra.

```

1000 #include "utils.h"
1001 #include "matrix.h"
1002
1003 void matrix_test(void)
1004 {
1005     puts("##### Basic Matrix Algebra #####");
1006     uint8_t m = 3;
1007     uint8_t n = 4;
1008     matrix_t diag_elem = M_PI;
1009
1010     matrix_t matrix1 [3][4] = {
1011         { 0.2785, 0.9649, 0.9572, 0.1419 },
1012         { 0.5469, 0.1576, 0.4854, 0.4218 },
1013         { 0.9575, 0.9706, 0.8003, 0.9157 },
1014     };
1015
1016     matrix_t matrix2 [3][4] = {
1017         { 0.7922, 0.0357, 0.6787, 0.3922 },
1018         { 0.9595, 0.8491, 0.7577, 0.6555 },
1019         { 0.6557, 0.9340, 0.7431, 0.1712 },
1020     };
1021
1022     matrix_t res_matrix [m][n];
1023     matrix_t trans_matrix [n][m];
1024     matrix_t res_mul_matrix [m][m];
1025
1026     % Create a matrix with specified values in the main diagonal
1027     matrix_get_diag_mat(m, n, diag_elem, res_matrix);
1028     printf("diag_matrix = ");
1029     matrix_flex_print(m, n, res_matrix, 7, 4);
1030
1031     % Matrix transpose
1032     matrix_transpose(m, n, matrix2, trans_matrix);
1033     printf("trans(matrix2) = ");
1034     matrix_flex_print(n, m, trans_matrix, 7, 4);
1035
1036     % Matrix multiplication
1037     matrix_mul(m, n, matrix1, n, m, trans_matrix, res_mul_matrix);
1038     printf("matrix1 x matrix2 = ");
1039     matrix_flex_print(m, m, res_mul_matrix, 7, 4);
1040 }

```

Listing 2. Outputs of the example of basic matrix algebra.

```

1000 ##### Basic Matrix Algebra #####
1001 diag_matrix = {
1002     { 3.1416, 0.0000, 0.0000, 0.0000},
1003     { 0.0000, 3.1416, 0.0000, 0.0000},
1004     { 0.0000, 0.0000, 3.1416, 0.0000}
1005 };
1006
1007 trans(matrix2) = {
1008     { 0.7922, 0.9595, 0.6557},
1009     { 0.0357, 0.8491, 0.9340},
1010     { 0.6787, 0.7577, 0.7431},
1011     { 0.3922, 0.6555, 0.1712}
1012 };
1013
1014 matrix1 x matrix2 = {
1015     { 0.9604, 1.9048, 1.8194},
1016     { 0.9338, 1.3028, 0.9387},
1017     { 1.6955, 2.9495, 2.2858}
1018 };

```

6. Evaluation of the Algorithms

We evaluated the linear as well as the nonlinear algebra modules on an STM32F407 Microcontroller Unit (MCU) based on the ARM Cortex-M4 core operating at 168 MHz and having a memory capacity of 192 KB RAM. In order to demonstrate the scalability of the algorithms implemented, we also evaluated the same algorithms on Raspberry Pi 3, which has more capacity (Quad Core 1.2 GHz and 1 GB RAM) than the STM32F4-MCU.

6.1. Evaluation of the Linear Algebra Module

We evaluated the linear algebra module by using a $(m \times n)$ matrix A and a vector \vec{b} with uniformly distributed random numbers. The aim is to calculate and measure the mean execution time of the methods for solving linear equation systems. We evaluated the SVD-, QR-, and the LU-based algorithms for solving linear equation systems described in Section 3.1. The determined and the over-determined linear equation systems can be represented by the colon notation as follows:

$$A(1:i, 1:i)\vec{x} = \vec{b}(1:i), \quad (37)$$

where $2 \leq i \leq n$, and

$$A(1:i, 1:n)\vec{x} = \vec{b}(1:i), \quad (38)$$

where $n + 1 \leq i \leq m$ and $A(1:i, 1:n)$ is the sub-matrix of A with rows 1 up to i and columns 1 up to n . We use the same format as the corresponding column notation in MATLAB. The determined and the over-determined linear equation systems are illustrated in the matrix form in Equations (39) and (40), respectively.

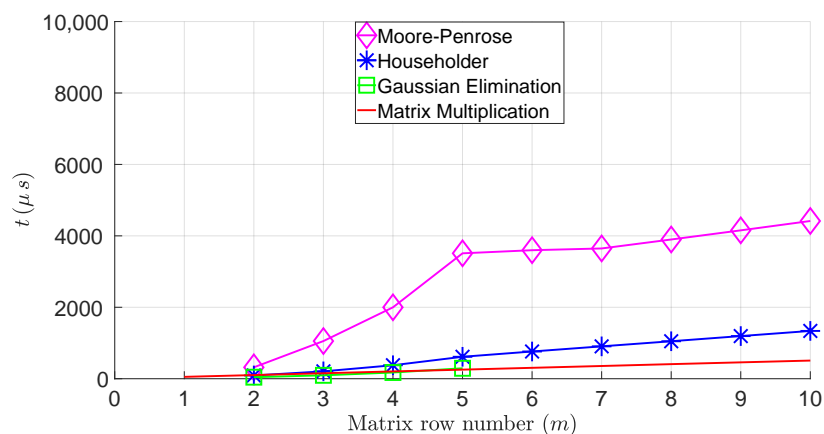
$$\begin{bmatrix} a_{11} & a_{12} & \vdots & a_{13} & \vdots & a_{14} & \vdots & \dots & a_{1n} \\ a_{21} & a_{22} & \vdots & a_{23} & \vdots & a_{24} & \vdots & \dots & a_{2n} \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ a_{31} & a_{32} & \vdots & a_{33} & \vdots & a_{34} & \vdots & \dots & a_{3n} \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ a_{41} & a_{42} & \vdots & a_{43} & \vdots & a_{44} & \vdots & \dots & a_{4n} \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \vdots & a_{m3} & \vdots & a_{m4} & \vdots & \dots & a_{mn} \end{bmatrix} \quad (39)$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{61} & a_{62} & a_{63} & \dots & a_{6n} & \vdots \\ a_{71} & a_{72} & a_{73} & \dots & a_{7n} & \vdots \\ a_{81} & a_{82} & a_{83} & \dots & a_{8n} & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \dots & a_{mn} & \vdots \end{bmatrix} \quad (40)$$

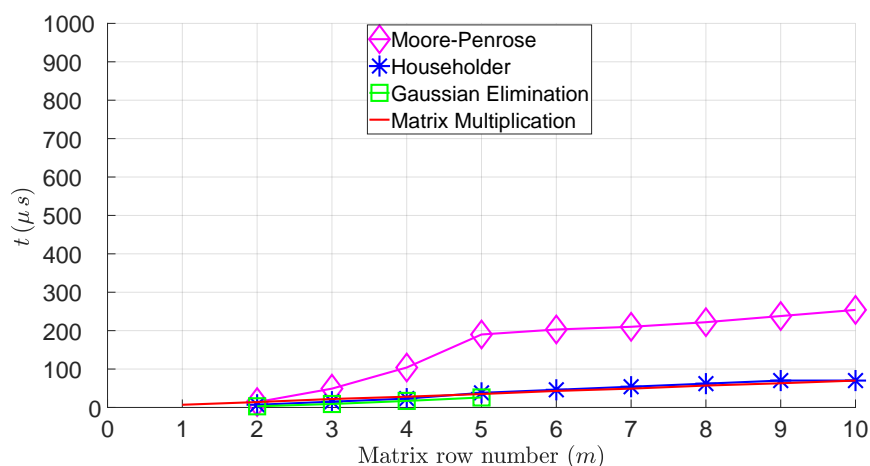
The horizontal and vertical dotted lines enclose the square and rectangular sub-matrices in Equations (39) and (40). We set the maximal row (m) and column (n) numbers to 10 and 5. We also measured the mean execution time of the matrix multiplication by using the same matrix $A_{m,n}$ and a matrix $B_{(n,p)}$ initialized with uniformly distributed random numbers. The row and column number of the matrix B are equal to 5 and 10. The row number (m) of the matrix A varies from 1 to 10. The linear equation systems are solved by using three different decomposition algorithms: the Golub–Kahan–Reinsch, Householder,

and GE with pivoting. We measured the execution time of matrix multiplications as well as of the methods for solving linear equation systems on the STM32F4 MCU and the Raspberry Pi 3.

For solving linear equation systems, Figure 4 compares the execution time of the following algorithms: GE with pivoting, the Householder, and the Golub–Kahan–Reinsch. Furthermore, Figure 4 represents the execution time of the matrix multiplications. Figure 4a,b illustrate the execution time of these algorithms in function of the row number of the matrix on the STM32F4 MCU and the Raspberry Pi 3. The Golub–Kahan–Reinsch-based algorithm has the largest execution time, since it is more expensive than other algorithms (see Table 2). Table 3 summarizes the mean execution time of the matrix evaluated by calculating an $A_{10,5} \times B_{5,7}$ matrix or solving an $(7, 5)$ linear equation system. These execution times are measured on the STM32F4 MCU and the Raspberry Pi 3. The Raspberry Pi 3 outperforms the STM32F4 MCU, as expected, due to the limited computing capacity of the STM32F4 MCU. However, the execution time for finding a solution applying the Golub–Kahan–Reinsch algorithm remains in a micro-second range on the STM32F4 MCU, which would be sufficient for many IoT applications.



(a) Time Measurement on STM32F4 MCU



(b) Time Measurement on Raspberry Pi 3

Figure 4. Computing time evaluation of matrix multiplications and linear equation systems.

Table 2. Complexity of the algorithms evaluated.

Algorithm	Complexity [Flops]
Matrix multiplication: $A_{m,n} \times B_{n,p}$	$mp(2n - 1)$
QR-Householder	$2mn^2 - \frac{2}{3}n^3$
Golub–Kahan–Reinsch	$4m^2n + 8mn^2 + 9n^3$

Table 3. Mean execution time of computing $A_{7,5} \times B_{5,10}$ or solving an (7,5) linear equation system.

Algorithm	Mean Execution Time (μ s)	
	STM32F4	Raspberry Pi 3
Matrix multiplication: $A_{7,5} \times B_{5,10}$	356	49
Householder-based solution	906	54
Golub–Kahan–Reinsch-based solution	3647	204

6.2. Evaluation of the Non-Linear Algebra Module

The nonlinear algebra module is evaluated by using exponential and sinusoidal data [64]. Optimizing of least-squares problems are solved by using the modified GN and LVM methods as described in Sections 3.2.1 and 3.2.2.

6.2.1. Evaluation with Exponential Data

Given the model function $g(\vec{x}, t)$ that is equal to:

$$g(\vec{x}, t) = x_1 e^{x_2 t}, \quad (41)$$

where $\vec{x} = [x_1, x_2]^T$ and $\vec{x}_0 = [6, 0.3]$ is the initial guess. The data set is $d(t_i, y_i)$, whereby t_i is equal to $\{1, \dots, 8\}$ and y_i is equal to $\{8.3, 11.0, 14.7, 19.7, 26.7, 35.2, 44.4, 55.9\}$.

The aim is to find the parameters (x_1, x_2) that most accurately match the model function $g(\vec{x}, t)$ by minimizing the sum of squares of the error function f_i . The function f_i computes the residual values and is equal to:

$$f_i(x_1, x_2) = x_1 e^{x_2 t_i} - y_i. \quad (42)$$

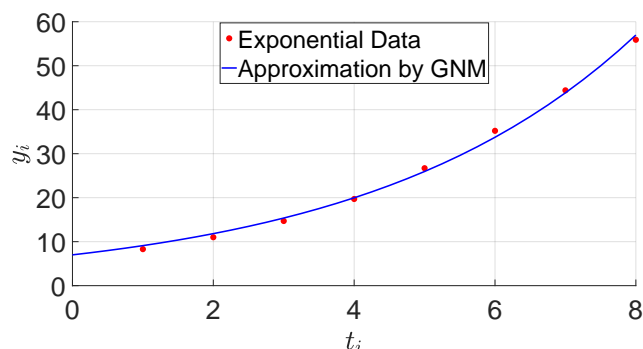
We introduce the error function vector \vec{f} :

$$\vec{f}(x_1, x_2) = [x_1 e^{x_2} - y_1, \dots, x_1 e^{8x_2} - y_8]^T. \quad (43)$$

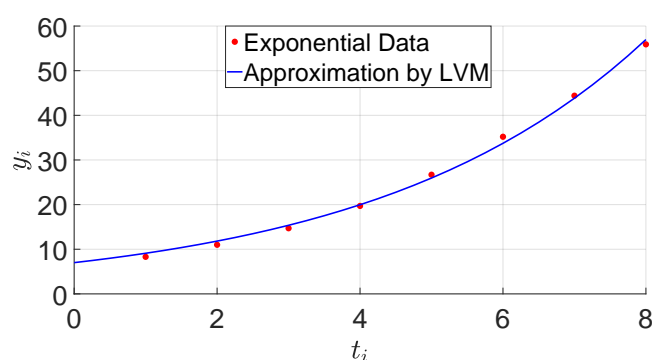
The Jacobian matrix is equal to:

$$J_f = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \\ \vdots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} \end{bmatrix} = \begin{bmatrix} e^{x_2} & e^{x_2} x_1 \\ e^{2x_2} & 2e^{2x_2} x_1 \\ \vdots & \vdots \\ e^{8x_2} & 8e^{8x_2} x_1 \end{bmatrix} \quad (44)$$

The initial square residual $\|\vec{f}(\vec{x}_0)\|_2^2$ is equal to 127.309. We get the solution $\vec{x}_3 = [7.000093, 0.262078]^T$ by using the GN algorithm after three iterations. The appropriate square residual $\|\vec{f}(\vec{x}_3)\|_2^2$ is equal to 6.013, which indicates the improvement of the model. We obtain the solution $\vec{x}_3 = [7.000090, 0.262078]^T$ by using the LVM algorithm after three iterations. The LVM algorithm shows nearly the same behavior as the modified GN method. This is confirmed by Figure 5.



(a) Exponential data and GNM after the third iteration



(b) Exponential data and LVM after the third iteration

Figure 5. Exponential model approximation. GNM, Gauss–Newton Method. LVM, Levenberg–Marquardt Method.

Table 4 summarizes the average time per iteration required from the GNM and LVM algorithms on the STM32F4 MCU and the Raspberry Pi 3.

Table 4. Mean execution time of the Gauss–Newton and Levenberg–Marquardt methods (per iteration) using exponential data.

Algorithm	Mean Execution Time (μ s)	
	STM32F4	Raspberry Pi 3
Gauss–Newton	1065	35
Levenberg–Marquardt	1194	42

6.2.2. Evaluation with Sinusoidal Data

The model function is:

$$g(\vec{x}, t) = x_1 \sin(x_2 t + x_3) + x_4, \quad (45)$$

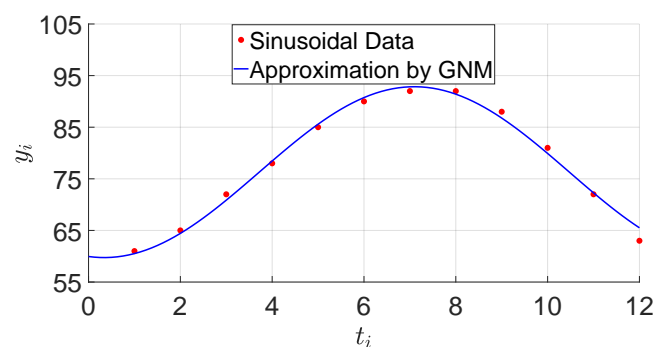
whereby, $\vec{x} = [x_1, x_2, x_3, x_4]^T$ and $\vec{x}_0 = [17, 0.5, 10.5, 77]$ is the initial guess. The set of data points is $d(t_i, y_i)$, where t_i is equal to $\{1, \dots, 12\}$ and y_i is equal to $\{61, 65, 72, 78, 85, 90, 92, 92, 88, 81, 72, 63\}$. The error function is $f_i = x_1 \sin(x_2 t_i + x_3) + x_4 - y_i$; therefore, the error function vector \vec{f} is:

$$\vec{f}(x_1, x_2, x_3, x_4) = \begin{bmatrix} x_1 \sin(x_2 + x_3) + x_4 - y_1 \\ x_1 \sin(2x_2 + x_3) + x_4 - y_2 \\ \vdots \\ x_1 \sin(12x_2 + x_3) + x_4 - y_{12} \end{bmatrix}. \quad (46)$$

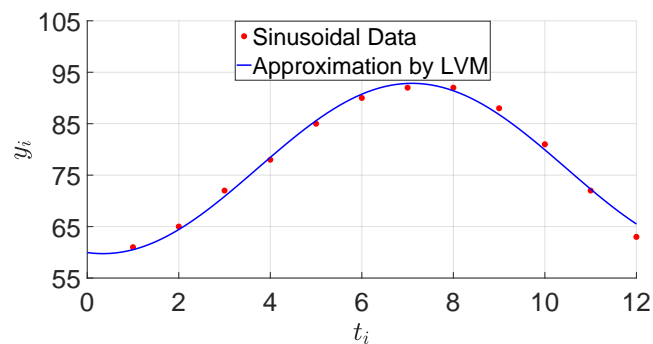
Thus, the Jacobian matrix is calculated using the partial derivatives in Equation (44) and is equal to:

$$J_f = \begin{bmatrix} \sin(x_2 + x_3) & x_1 \cos(x_2 + x_3) & x_1 \cos(x_2 + x_3) \\ \sin(2x_2 + x_3) & 2x_1 \cos(2x_2 + x_3) & x_1 \cos(2x_2 + x_3) \\ \vdots & \vdots & \vdots \\ \sin(12x_2 + x_3) & 12x_1 \cos(12x_2 + x_3) & x_1 \cos(12x_2 + x_3) \end{bmatrix} \quad (47)$$

The initial square residual $\|\vec{f}(\vec{x}_0)\|_2^2$ is equal to 40.048. After one iteration ($\|\vec{f}(\vec{x}_0)\|_2^2 = 13.805$), the LVM algorithm is slightly more efficient than the GN method ($\|\vec{f}(\vec{x}_0)\|_2^2 = 13.810$). Both algorithms show the same behavior after two iterations. Figure 6 shows the sinusoidal model after three iterations by using the GN and LVM algorithms.



(a) Sinusoidal data and GNM after the third iteration



(b) Sinusoidal data and LVM after the third iteration

Figure 6. Sinusoidal model approximation. GNM, Gauss–Newton Method. LVM, Levenberg–Marquardt Method.

Table 5 summarizes the average time per iteration required from the GNM and LVM approaches on the STM32F4 MCU and the Raspberry Pi 3.

Table 5. Mean execution time of the Gauss–Newton and Levenberg–Marquardt methods (per iteration) using sinusoidal data.

Algorithm	Mean Execution Time (μ s)	
	STM32F4	Raspberry Pi 3
Gauss–Newton	3165	157
Levenberg–Marquardt	2825	141

7. Conclusions and Outlook

We presented an open source library for linear and nonlinear algebra as well as an application module for localization that is suitable for resource-limited, mobile, and embedded systems. This library permits the solution of linear equations and matrix operations like the matrix decomposition, the calculation of the inverse, or the rank of a matrix. It provides various algorithms such as sorting or filtering algorithms. This software also enables solving nonlinear problems like curve fitting or nonlinear equations. [RcdMathLib](#) allows for the localization of mobile devices by using localization algorithms like, for instance, the trilateration. The localization can be further refined by using an adaptive optimization algorithm based on the SVD method. The localization software module facilitates the localization of the mobile device in NLoS scenarios by using a multipath distance detection and mitigation algorithm. [RcdMathLib](#) can serve as a basis for artificial intelligence techniques for mobile technologies with IoT, or as a tool in research and industry. Therefore, we intend to extend the [RcdMathLib](#) with machine learning or digital signal processing algorithms. We also aim to extend the package with an additional algorithm for solving nonlinear problems called the Landweber method [65].

Author Contributions: Z.K. conceived the research, designed the software architecture, and implemented the software components of the [RcdMathLib](#). Furthermore, he integrated the software components of the [RcdMathLib](#) in RIOT-OS, performed the experiments and data evaluation, and wrote all parts of the article. A.N. offered valuable tests and evaluation of the algorithms. He helped through the design of the localization algorithms. He wrote the “related works” part and contributed to the writing of the article part “Library Architecture and Description”. M.G. gave valuable suggestions and offered plenty of comments and useful discussions to the paper. J.S. gave valuable suggestions to the paper and reviewed the article. C.M. gave suggestions to the article and reviewed it. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Source code can be download from the GitLab repository on Reference [63].

Acknowledgments: We acknowledge the support of the Deutsche Forschungsgemeinschaft (DFG—German Research Foundation) and the Open Access Publishing Fund of Technical University of Darmstadt. The authors thank Naouar Guerchali for the support in the implementation and evaluation of the Levenberg–Marquardt algorithm. The authors thank Nikolas Voth for the final grammatical corrections of the paper.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Ceruzzi, P.E. The Early Computers of Konrad Zuse, 1935 to 1945. *Ann. Hist. Comput.* **1981**, *3*, 241–262. [[CrossRef](#)]
2. Ceruzzi, P.E. *A History of Modern Computing*, 2nd ed.; MIT Press: Cambridge, MA, USA, 2003.
3. Lee, J.A.N. *Computer Pioneers*; IEEE Computer Society Press: Los Alamitos, CA, USA, 1995.
4. Partlett, B. Handbook for Automatic Computation, Vol. II, Linear Algebra (J. H. Wilkinson and C. Reinsch). *SIAM Rev.* **1972**, *14*, 658–661. [[CrossRef](#)]
5. Dongarra, J.J.; Moler, C.B.; Bunch, J.R.; Stewart, G.W. *LINPACK Users' Guide*; Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 1979; [[CrossRef](#)]
6. Garbow, B.S.; Boyle, J.M.; Dongarra, J.J.; Moler, C.B. *Matrix Eigensystem Routines—EISPACK Guide Extension*; Springer: Berlin/Heidelberg, Germany, 1977. [[CrossRef](#)]
7. Cleve Moler. A Brief History of MATLAB. In *Technical Articles and Newsletters*; MathWorks: Natick, MA, USA, 2019.
8. Naveen, S.; Kounte, M.R. Key Technologies and challenges in IoT Edge Computing. In Proceedings of the 2019 Third International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC), Palladam, India, 12–14 December 2019; pp. 61–65. [[CrossRef](#)]
9. Kasmi, Z. Home of the RcdMathLib (Mathematical Library for Resource-Constrained Devices). 2021. Available online: <https://git.imp.fu-berlin.de/zkasmi/RcdMathLib/-/wikis/Home> (accessed on 25 February 2021).

10. Strang, G. *Linear Algebra and Its Applications*, 4th ed.; Thomson, Brooks/Cole: Belmont, CA, USA, 2006.
11. Schanze, T. Compression and Noise Reduction of Biomedical Signals by Singular Value Decomposition. *IFAC-PapersOnLine* **2018**, *51*, 361–366. [[CrossRef](#)]
12. Kasmi, Z. Open Platform Architecture for Decentralized Localization Systems Based on Resource-Constrained Devices. Ph.D. Thesis, Freie Universität Berlin, Department of Mathematics and Computer Science, Berlin, Germany, 2019.
13. Madsen, K.; Nielsen, H.B.; Tingleff, O. *Methods for Non-Linear Least Squares Problems*, 2nd ed.; Technical University of Denmark, Lyngby, Denmark, 2004.
14. Rose, J.A.; Tong, J.R.; Allain, D.J.; Mitchell, C.N. The Use of Ionospheric Tomography and Elevation Masks to Reduce the Overall Error in Single-Frequency GPS Timing Applications. *Adv. Space Res.* **2011**, *47*, 276–288. [[CrossRef](#)]
15. Guckenheimer, J. Numerical Computation in the Information Age. In *SIAM NEWS*; Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 1988.
16. Arm Limited. Arm Performance Libraries. 2019. Available online: <https://developer.arm.com/tools-and-software/server-and-hpc/arm-architecture-tools/arm-performance-libraries> (accessed on 25 February 2021).
17. Sanchez, J.; Canton, M.P. *Microcontrollers: High-Performance Systems and Programming*; CRC Press: Boca Raton, FL, USA, 2018. [[CrossRef](#)]
18. Texas Instruments Incorporated. *MSP430 IQmathLib User's Guide*; Texas Instruments Incorporated: Dallas, TX, USA, 2015.
19. Aimonen, P. Cross Platform Fixed Point Maths Library 2012–2019. Available online: <https://github.com/PetteriAimonen/libfixmath> (accessed on 25 February 2021).
20. Nicolosi, A. A Simple, Tiny and Efficient BLAS Library, Designed for PC and Microcontrollers. 2018. Available online: <https://github.com/alenic/microBLAS> (accessed on 25 February 2021).
21. Matlack, C. Minimal Linear Algebra Library. 2018. Available online: <https://github.com/eecharlie/MatrixMath> (accessed on 25 February 2021).
22. Stewart, T. A Library for Representing Matrices and Doing Matrix Math on Arduino. 2018. Available online: <https://github.com/tomstewart89/BasicLinearAlgebra> (accessed on 25 February 2021).
23. Oliphant, T.E. *Guide to NumPy*, 2nd ed.; CreateSpace Independent Publishing Platform: North Charleston, SC, USA, 2015.
24. Nunez-Iglesias, J.; van der Walt, S.; Dashnow, H. *Elegant SciPy: The Art of Scientific Python*, 1st ed.; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2017.
25. Bell, C. *MicroPython for the Internet of Things: A Beginner's Guide to Programming with Python on Microcontrollers*; Apress: Berkeley, CA, USA, 2017; [[CrossRef](#)]
26. Kurniawan, A. *CircuitPython Development Workshop*; PE Press: Berlin, Germany, 2018.
27. Björck, A. *Numerical Methods for Least Squares Problems*; Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 1996. [[CrossRef](#)]
28. Golub, G.H.; Van Loan, C.F. *Matrix Computations*, 4th ed.; Johns Hopkins University Press: Baltimore, MD, USA, 2013.
29. Barata, J.; Hussein, M. The Moore-Penrose Pseudoinverse. A Tutorial Review of the Theory. *Braz. J. Phys.* **2011**, *42*, 146–165. [[CrossRef](#)]
30. Kaw, A. *Introduction to Matrix Algebra*; University of South Florida: Tampa, FL, USA, 2008.
31. Sengupta, S.; Korobkin, C.P. *C++: Object-Oriented Data Structures*; Springer: New York, NY, USA, 2012. [[CrossRef](#)]
32. Bailey, R. *The Box-Muller Method and the T-distribution*; Department of Economics Discussion Paper; University of Birmingham, Department of Economics: Birmingham, UK, 1992.
33. Allgower, E.; Georg, K.; Research, U.; Foundation, N. *Computational Solution of Nonlinear Systems of Equations*; Lectures in Applied Mathematics; American Mathematical Society: Providence, RI, USA, 1990.
34. Nocedal, J.; Wright, S. *Numerical Optimization*; Springer: New York, NY, USA, 2006. [[CrossRef](#)]
35. Chen, Y.Y.; Gao, Y. Two New Levenberg-Marquardt Methods for Non-smooth Nonlinear Complementarity Problems. *Sci. Asia* **2014**, *40*, 89. [[CrossRef](#)]
36. Song, L.; Gao, Y. On The Local Convergence of a Levenberg-Marquardt Method for Non-smooth Nonlinear Complementarity Problems. *Sci. Asia* **2017**, *43*, 377. [[CrossRef](#)]
37. Du, S.Q. Some Global Convergence Properties of the Levenberg-Marquardt Methods with Line Search. *J. Appl. Math. Inform.* **2013**, *31*, 373–378. [[CrossRef](#)]
38. Dahmen, W.; Reusken, A. *Numerik für Ingenieure und Naturwissenschaftler*; Springer: Berlin/Heidelberg, Germany, 2008. [[CrossRef](#)]
39. Guerchali, N. Untersuchung des Levenberg-Marquardt-Algorithmus zur Indoor-Lokalisierung für den STM407-Mikrocontroller. Bachelor's Thesis, Fachhochschule Aachen, Aachen, Germany, 2017.
40. Ramnath, S.; Javali, A.; Narang, B.; Mishra, P.; Routray, S.K. IoT-Based Localization and Tracking. In Proceedings of the 2017 International Conference on IoT and Application (ICIOT), Nagapattinam, India, 19–20 May 2017; pp. 1–4. [[CrossRef](#)]
41. Reed, J.H. *An Introduction to Ultra Wideband Communication Systems*, 1st ed.; Prentice Hall Press: Upper Saddle River, NJ, USA, 2005.
42. AeroScout Corporation. Available online: <http://www.aeroscout.com> (accessed on 25 February 2021).
43. Holm, S. Ultrasound Positioning Based on Time-of-Flight and Signal Strength. In Proceedings of the 2012 International Conference on Indoor Positioning and Indoor Navigation (IPIN), Sydney, Australia, 13–15 November 2012; pp. 1–6. [[CrossRef](#)]

44. Pasku, V.; De Angelis, A.; De Angelis, G.; Arumugam, D.D.; Dionigi, M.; Carbone, P.; Moschitta, A.; Ricketts, D.S. Magnetic Field-Based Positioning Systems. *IEEE Commun. Surv. Tutor.* **2017**, *19*, 2003–2017. [[CrossRef](#)]
45. Chawathe, S.S. Low-Latency Indoor Localization Using Bluetooth Beacons. In Proceedings of the 2009 12th International IEEE Conference on Intelligent Transportation Systems, St. Louis, MO, USA, 4–7 October 2009; pp. 1–7. [[CrossRef](#)]
46. Norrdine, A. An Algebraic Solution to the Multilateration Problem. In Proceedings of the Third International Conference on Indoor Positioning and Indoor Navigation (IPIN2012), Sydney, Australia, 13–15 November 2012; pp. 1–5.
47. Kasmi, Z.; Norrdine, A.; Blankenbach, J. Platform Architecture for Decentralized Positioning Systems. *Sensors* **2017**, *17*, 957. [[CrossRef](#)] [[PubMed](#)]
48. Kasmi, Z.; Guerchali, N.; Norrdine, A.; Schiller, J.H. Algorithms and Position Optimization for a Decentralized Localization Platform Based on Resource-Constrained Devices. *IEEE Trans. Mob. Comput.* **2019**, *18*, 1731–1744. [[CrossRef](#)]
49. Prigge, E.A.; How, J.P. Signal architecture for a distributed magnetic local positioning system. *IEEE Sens. J.* **2004**, *4*, 864–873. [[CrossRef](#)]
50. Prieto, J.C.; Croux, C.; Jiménez, A.R. RoPEUS: A New Robust Algorithm for Static Positioning in Ultrasonic Systems. *Sensors* **2009**, *9*, 4211. [[CrossRef](#)] [[PubMed](#)]
51. Van Heesch, D. Doxygen: Generate Documentation from Source Code. Available online: <https://www.doxygen.nl/index.html> (accessed on 25 February 2021).
52. Datta, B.N. *Numerical Linear Algebra and Applications*, 2nd ed.; Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 2010.
53. Trefethen, L.; Schreiber, R. Average-Case Stability of Gaussian Elimination. *SIAM J. Matrix Anal. Appl.* **1990**, *11*. [[CrossRef](#)]
54. Trefethen, L.; Bau, D. *Numerical Linear Algebra*; Other Titles in Applied Mathematics; Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 1997.
55. Stewart, G.W. *Matrix Algorithms: Volume 1, Basic Decompositions*; Matrix Algorithms; Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 1998.
56. Thisted, R.A. *Elements of Statistical Computing: Numerical Computation*; CRC Press/Taylor & Francis Group: Boca Raton, FL, USA, 1988.
57. Gander, W.; Gander, M.J.; Kwok, F. *Scientific Computing—An Introduction Using Maple and MATLAB*; Springer Publishing Company, Incorporated: New York, NY, USA, 2014.
58. Golub, G.H.; Reinsch, C. Singular Value Decomposition and Least Squares Solutions. *Numer. Math.* **1970**, *14*, 403–420. [[CrossRef](#)]
59. Free Software Foundation, Inc. GNU Lesser General Public License, Version 2.1. 2021. Available online: <https://www.gnu.org/licenses/old-licenses/lgpl-2.1.en.html> (accessed on 25 February 2021).
60. Baccelli, E.; Hahm, O.; Günes, M.; Wählich, M.; Schmidt, T.C. RIOT OS: Towards an OS for the Internet of Things. In Proceedings of the 2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), Turin, Italy, 14–19 April 2013; pp. 79–80. [[CrossRef](#)]
61. Hope, M. Linaro Toolchain Binaries. 2020. Available online: <https://launchpad.net/linaro-toolchain-binaries> (accessed on 25 February 2021).
62. Team, O. *OpenOCD—Open On-Chip Debugger Reference Manual*; Samurai Media Limited: London, UK, 2015.
63. Kasmi, Z. GitLab of the RcdMathLib (Mathematical Library for Resource-Constrained Devices). 2021. Available online: <https://git.imp.fu-berlin.de/zkasmi/RcdMathLib> (accessed on 25 February 2021).
64. Croeze, A.; Pittman, L.; Reynolds, W. *Solving Nonlinear Least-Squares Problems with the Gauss–Newton and Levenberg–Marquardt Methods*; Technical Report; Department of Mathematics, Louisiana State University: Baton Rouge, LA, USA, 2012.
65. Zhao, X.L.; Huang, T.Z.; Gu, X.M.; Deng, L.J. Vector Extrapolation based Landweber Method for Discrete ill-posed Problems. *Math. Probl. Eng.* **2017**, *2017*, 1375716. [[CrossRef](#)]