electronics

MDPI

Article

Singular Value Decomposition in Embedded Systems Based on ARM Cortex-M Architecture

Michele Alessandrini †, Giorgio Biagetti †, Paolo Crippa *,†, Laura Falaschetti †, Lorenzo Manoni † and Claudio Turchetti †

Department of Information Engineering, Università Politecnica delle Marche,
Via Brecce Bianche 12, I-60131 Ancona, Italy; m.alessandrini@univpm.it (M.A.); g.biagetti@univpm.it (G.B.); l.falaschetti@univpm.it (L.F.); l.manoni@pm.univpm.it (L.M.); c.turchetti@univpm.it (C.T.)

* Correspondence: p.crippa@univpm.it; Tel.: +39-071-220-4541

† These authors contributed equally to this work.

Abstract: Singular value decomposition (SVD) is a central mathematical tool for several emerging applications in embedded systems, such as multiple-input multiple-output (MIMO) systems, data analytics, sparse representation of signals. Since SVD algorithms reduce to solve an eigenvalue problem, that is computationally expensive, both specific hardware solutions and parallel implementations have been proposed to overcome this bottleneck. However, as those solutions require additional hardware resources that are not in general available in embedded systems, optimized algorithms are demanded in this context. The aim of this paper is to present an efficient implementation of the SVD algorithm on ARM Cortex-M. To this end, we proceed to (i) present a comprehensive treatment of the most common algorithms for SVD, providing a fairly complete and deep overview of these algorithms, with a common notation, (ii) implement them on an ARM Cortex-M4F microcontroller, in order to develop a library suitable for embedded systems without an operating system, (iii) find, through a comparative study of the proposed SVD algorithms, the best implementation suitable for a low-resource bare-metal embedded system, (iv) show a practical application to Kalman filtering of an inertial measurement unit (IMU), as an example of how SVD can improve the accuracy of existing algorithms and of its usefulness on a such low-resources system. All these contributions can be used as guidelines for embedded system designers. Regarding the second point, the chosen algorithms have been implemented on ARM Cortex-M4F microcontrollers with very limited hardware resources with respect to more advanced CPUs. Several experiments have been conducted to select which algorithms guarantee the best performance in terms of speed, accuracy and energy consumption.

Keywords: singular value decomposition (SVD); matrix decomposition; embedded systems; microcontrollers; ARM Cortex-M; Kalman

Citation: Alessandrini, M.; Biagetti, G.; Crippa, P.; Falaschetti, L.; Manoni, L.; Turchetti, C. Singular Value Decomposition in Embedded Systems Based on ARM Cortex-M Architecture. *Electronics* **2021**, *10*, 34. https://doi.org/10.3390/electronics10010034

Received: 11 November 2020
Accepted: 25 December 2020
Published: 28 December 2020

1. Introduction

Singular value decomposition (SVD) is one of the basic and most important mathematical tools in modern signal processing. The method was firstly established by Beltrami and Jordan [1] and successively generalized by Autonne [2], Eckart and Young [3]. Since then SVD has successfully been applied on a huge number of different application fields, such as biomedical signal processing [4–10], image processing [11–19], Kalman filtering [20–22], array signal processing [23], dynamic networks [24], speech processing [25], simultaneous localization and mapping (SLAM) systems [26], and variable digital filter design [27], to cite just a few.

Various algorithms have been developed during the last decades for solving SVD [28], among these Golub–Reisch [29], Demmel–Kahan [30], Jacobi rotation [31], one-sided Jacobi rotation [32], and divide and conquer [33] algorithms are largely used since they guarantee good performance in common applications. An overview of SVD as well as the methods for its computation can be found in [34,35].

In recent years, there has been intense interest for emerging applications in embedded systems, such as multiple-input multiple-output (MIMO) systems [36,37], data analytics [38–41], sparse representation of signals [42–47], that require efficient SVD algorithms. Since SVD algorithms reduce to solve an eigenvalue problem, that is computationally expensive, both specific hardware solutions [48–56] and parallel implementations [57,58] have been proposed to overcome this bottleneck.

For example, optimized libraries such as Linear Algebra PACKage (LAPACK) and Basic Linear Algebra Subprograms (BLAS) (in their several implementations) that represent the state-of-the-art solution on computer-class CPUs, use several optimizations to efficiently take advantage of the hardware resources. An important optimization is reorganizing the operation flow and the data storage to make a better use of the processor's cache memory, in order to speed-up the access to the large amount of data from and to the slower RAM. Another improvement comes from parallelism, that is splitting the problem in several smaller subsets and exploiting the parallel execution of modern multi-core CPUs. A further form of parallelism is given by single instruction multiple data (SIMD) technology, consisting of single instructions that perform the same operation in parallel on multiple data elements of the same type and size. For example, a CPU that normally adds two 32-bit values, performs instead four parallel additions of 8-bit values in the same amount of time. Therefore, SIMD makes better use of the available resources in a microprocessor observing that when performing operations on large amount of data on a microprocessor, parts of the computation units are unused, while continuing to consume power.

It is worth to notice that all the SVD algorithms exhibit a large quantity of data-level parallelism, making them well suited to be implemented with SIMD acceleration and multi-core parallelism.

However, none of the cited hardware resources (cache memory, parallel execution units) exist in microcontrollers, so all the relevant optimization techniques are not useful on this class of embedded systems. About SIMD, actually the Cortex-M4 and newer embedded architectures offer a primitive form of SIMD instructions that are limited to operate on 8 or 16-bit integer values, so they are not suitable for complex computations requiring higher precision floating-point values for their stability.

Regarding more complex architectures, there is a rich literature on how to implement mathematical algorithms on Field Programmable Gate Arrays (FPGAs) and other programmable hardware [53–56]. Those are generally sophisticated and expensive systems, used for high-end applications and exploiting a completely different computation model, based on massive parallelism. Conversely the scope of this paper is on affordable, low-power and low-resource embedded systems, generally based on microcontrollers, as universally used in distributed scenarios like for example the Internet of Things concept.

Many embedded systems have a very limited amount of available memory, especially data memory, and may not employ an underlying operating system to further reduce resource usage. Because of this, the existing optimized libraries intended for computer-class CPUs are not readily usable in embedded systems, nor easily portable. For this reason, the demand of specific accelerating techniques that are able to implement SVD algorithms in embedded processors, has seen a growing interest in the past few years [59].

Regarding the SVD implementation a large variety of algorithms exist, thus from the embedded system designer point of view, comparing the performance achieved with different algorithms implemented in a high-performance embedded processor is of great interest.

The aim of this paper is to present an efficient implementation of the SVD algorithm on ARM Cortex-M (Arm Ltd., Cambridge, UK). More specifically, as the SVD algorithm is a central mathematical tool for several emerging applications in embedded systems and, to our knowledge, no previous implementations of this algorithm exists in literature for bare-metal microcontrollers, i.e., without an operating system, it is of paramount importance to develop a library suitable for these devices, specifically for the ARM Cortex-M4F architecture [60] . To achieve this goal, we proceeded:

(i) to provide a comprehensive treatment of the most common SVD algorithms, writing the existing formulations found in literature in a more compact and uniform manner, suitable to be used as a comprehensive reference;
(ii) to implement them on an ARM Cortex-M4F (Arm Ltd., Cambridge, UK). microcontroller, in order to develop a library that is suitable for embedded systems that work without an operating system. As already mentioned, LAPACK and BLAS represent the state-of-the-art solution on computer-class CPUs, but these optimized libraries are not readily available and easily portable for embedded systems, because of the very limited available memory and the lack of an underlying operating system of these devices. To this end, we decided to implement five common SVD algorithms (Golub–Reinsch, Demmel–Kahan, Jacobi rotation, one-sided Jacobi rotation and divide and conquer) in order to adapt them to operate in limited resource embedded systems and without an operating system.
(iii) to find the best implementation that fits the low-resources of bare-metal embedded systems. For the purpose of finding the algorithms with the best performance for an embedded system, this paper reports a comparative study of their performance in terms of several implementation metrics: speed, accuracy and energy consumption.
(iv) to show a practical real-time application of the SVD in an embedded system, by using the proposed optimized library for ARM Cortex-M4F in the Kalman filtering of an inertial measurement unit (IMU). This example also proofs the advantage of implementing the SVD on this class of systems, showing the improvement of the numerical accuracy of Kalman filtering by applying the SVD algorithm with respect to the conventional Kalman approach.

Thus, the paper is a good starting point for researchers and practitioners interested in developing new algorithms to be implemented in low-resource microcontrollers.

This paper is organized as follows. Section 2 is mainly focused on the five most representative SVD algorithms and gives a comprehensive treatment of them, with Appendix A summarizing the basic concepts of matrix algebra related to SVD transformation. Section 3 reports a comparative study of the five algorithms performance. Section 4 presents an application of SVD to Kalman filtering of data from an IMU, showing important improvements with respect to traditional algorithm implementation. Finally, some conclusions end this work.

2. Algorithms for the Singular Value Decomposition

Let A be a real $(m \times n)$ matrix with $m \geq n$. It is known that the decomposition

$$A = U\Sigma V^T \tag{1}$$

where

$$U^T U = V^T V = VV^T = I,\ \Sigma = \mathrm{diag}(\sigma_1, \ldots, \sigma_n) \tag{2}$$

exists [35]. The matrix U consists of n orthonormal eigenvectors corresponding to the n largest eigenvalues of AA^T, and the matrix V consists of the orthonormal eigenvectors of $A^T A$. The diagonal elements of Σ are the non-negative square roots of the eigenvalues of AA^T, called singular values. Assuming

$$\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_n \geq 0 \tag{3}$$

thus if $rank(A) = M$, it results $\sigma_{M+1} = \sigma_{M+2} = \ldots = \sigma_n = 0$. The decomposition (1) is called the singular value decomposition (SVD) of matrix A.

2.1. Golub–Reinsch Algorithm

This method, developed by G. H. Golub and C. Reinsch [29], acts directly on the matrix A thus avoiding unnecessary numerical inaccuracy due to the computation of $A^T A$. The algorithm can be divided into these consecutive steps:

(i) Householder's bidiagonalization (see Appendix A.2 for details);
(ii) implicit QR method with shift (see Appendix A.6 for details);

The pseudo-code of the algorithm is reported in Algorithm 1.

Algorithm 1 Golub–Reinsch

Require: $A \in \mathbb{R}^{m \times n} (m \geq n)$, ϵ a small multiple of the unit round-off

Use Algorithm A1 to compute bidiagonalization.

$$\begin{bmatrix} B \\ 0 \end{bmatrix} \leftarrow (U_1 \dots U_n)^T A (V_1 \dots V_{n-2})$$

Repeat

for $i = 1, \dots, n-1$ **do**

- Set $b_{i,i+1}$ to zero if $|b_{i,i+1}| \leq \epsilon(|b_{ii}| + |b_{i+1,i+1}|)$
- Find the largest q and the smallest p such that if

$$B = \begin{bmatrix} B_{11} & 0 & 0 \\ 0 & B_{22} & 0 \\ 0 & 0 & B_{33} \end{bmatrix} \begin{matrix} p \\ n-p-q \\ q \end{matrix}$$
$$\begin{matrix} p & n-p-q & q \end{matrix}$$

then B_{33} is diagonal and B_{22} has a nonzero superdiagonal.

if $q = n$ **then**

STOP

end if

if any diagonal entry in B_{22} is zero **then**

zero the superdiagonal entry in same row

else

apply the Algorithm A3

end if

end for

2.2. Demmel–Kahan Algorithm

The structure of the algorithm is based on the Golub–Reinsch algorithm previously described. Nevertheless the Demmel–Kahan algorithm [30] can achieve a better accuracy for small singular values. This algorithm consists of the following main consecutive steps:

(i) Householder's bidiagonalization (see Appendix A.2 for details);
(ii) QR iteration with zero-shift (see Appendix A.7 for details);

The description of this algorithm is shown in Algorithm 2.

Algorithm 2 Demmel–Kahan

Require: $A \in \mathbb{R}^{m\times n}(m \geq n) \in$ a small multiple of the unit round-off

Use Algorithm A1 to compute bidiagonalization.

$$\begin{bmatrix} B \\ 0 \end{bmatrix} \leftarrow (U_1 \dots U_n)^T A (V_1 \dots V_{n-2})$$

Repeat

for $i = 1 : n-1$ **do**

- Set $b_{i,i+1}$ to zero if a relative convergence criterion is met
- Find the largest q and the smallest p such that if

$$B = \begin{bmatrix} B_{11} & 0 & 0 \\ 0 & B_{22} & 0 \\ 0 & 0 & B_{33} \end{bmatrix} \begin{matrix} \text{p} \\ \text{n -p -q} \\ \text{q} \end{matrix}$$

$$\begin{matrix} \text{p} & \text{n-p-q} & \text{q} \end{matrix}$$

than B_{33} is diagonal and B_{22} has a nonzero superdiagonal.

if $q = n$ **then**

STOP

end if

if any diagonal entry in B_{22} is zero **then**

zero the superdiagonal entry in same row

else

apply the implicit zero-shift QR algorithm

end if

end for

2.3. Jacobi Rotation Algorithm

In this case given the real and symmetric matrix $A \in \mathbb{R}^{n\times n}$ the algorithm [31] aims to obtain a diagonal matrix $B \in \mathbb{R}^{n\times n}$ through the transformation

$$B = J^T A J \tag{4}$$

where J represents a sequence of rotation matrices. In particular for the k-th rotation or sweep can be rewritten as

$$A_{k+1} = J_k^T A_k J_k, \quad A_0 = A \tag{5}$$

where $J_k = J(p, q, \theta)$ is the Jacobi rotation matrix that rotates rows and columns p and q of A_k through the angle θ so that the (p, q) and (q, p) entries are zeroes. The p and q values are chosen properly at each iteration step. With reference to the sub matrices corresponding to the p, q columns we have

$$\begin{bmatrix} b_{pp} & b_{pq} \\ b_{qp} & b_{qq} \end{bmatrix} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T \begin{bmatrix} a_{pp} & a_{pq} \\ a_{qp} & a_{qq} \end{bmatrix} \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \tag{6}$$

The key point of the algorithm is to determine the rotation coefficients c and s in such a way the off-diagonal terms b_{pq} and b_{qp} are zeroed.

The algorithm is reported in Algorithm 3.

Algorithm 3 Jacobi rotation

Require: $A \in \mathbb{R}^{n\times n}$ symmetric

$B \leftarrow A \in \mathbb{R}^{n\times n}$

Repeat

for $i = 1 : n-1$ **do**

for $j = i+1 : n$ **do**

Compute the rotations coefficients s, c such that

$$\begin{bmatrix} b_{ii} & 0 \\ 0 & b_{jj} \end{bmatrix} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T \begin{bmatrix} a_{ii} & a_{ij} \\ a_{ji} & a_{jj} \end{bmatrix} \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \tag{7}$$

if $\text{off}(A) < \epsilon$, where $\text{off}(A) = \sqrt{\sum_{i\neq j} a_{ij}^2}$ and ϵ is a small multiple of the unit round-off **then**

STOP

end if

end for

end for

2.4. One-Sided Jacobi Rotation Algorithm

The main idea of this algorithm [32] is to rotate columns i and j of A through the angle θ so that they become orthogonal to each other. In such a way the (i, j) element of A^TA is implicitly zeroed resulting in the scalar product of the i, j columns.

Let $J(i, j, \theta)$ be the Givens matrix that when applied to the matrix A yields

$$B = (b_{:1} \cdots b_{:i} \cdots b_{:n}) = AJ = (a_{:1} \cdots a_{:i} \cdots a_{:n}) \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 \\ \ddots & & c & s & \\ & & -s & c & 0 \\ 0 & 0 & \cdots & 0 & 1 \end{bmatrix} \tag{8}$$

where the i, j columns of B are given by

$$\begin{aligned} b_{:i} &= c\, a_{:i} - s\, a_{:j} \\ b_{:j} &= s\, a_{:i} + c\, a_{:j} \end{aligned} \tag{9}$$

The key point of the algorithm is to determine the rotation coefficients c and s in such a way the elements $(B^TB)_{ij}$ and $(B^TB)_{ji}$ of the product $B = AJ(i, j, \theta)$ are zeroed.

The pseudo-code of the algorithm is reported in Algorithm 4.

Algorithm 4 One-sided Jacobi rotation

Require: $A \in \mathbb{R}^{n\times n}$ symmetric

$B \leftarrow A \in \mathbb{R}^{n\times n}$

Repeat

for $i = 1 : n-1$ **do**

for $j = i+1 : n$ **do**

Compute the rotations coefficients s, c such that

$$\begin{bmatrix} b_{ii} & 0 \\ 0 & b_{jj} \end{bmatrix} = \begin{bmatrix} a_{ii} & a_{ij} \\ a_{ji} & a_{jj} \end{bmatrix} \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \tag{10}$$

if off$(A) < \epsilon$, where off$(A) = \sqrt{\sum_{i\neq j} a_{ij}^2}$ and ϵ is a small multiple of the unit round-off **then**

STOP

end if

end for

end for

2.5. Divide and Conquer Algorithm

For a square symmetric matrix A a relationship between singular values and eingenvalues, as well as between singular vectors and eigenvectors exists. Indeed, as A can be diagonalized we can write

$$A = Q\Lambda Q^T = A\,\mathrm{sign}(\Lambda)|\Lambda|Q^T \tag{11}$$

where Λ is the eigenvalue diagonal matrix and Q is a unitary matrix (orthogonal in real case). Since we can always assume that the elements of $|\Lambda|$ are in decreasing order, then to the diagonalization (11) corresponds an SVD such that $U = Q\,\mathrm{sign}(\Lambda)$, $\Sigma = |\Lambda|$ and $V^T = Q^T$. In words the singular values are the absolute values of eigenvalues and the singular vectors are the eigenvectors (same norm and direction, but not necessary the same versus). For a non symmetric matrix the singular values and the singular vectors are not directly related to the eigenvalues and eigenvectors, instead there is a strict relationship with eigenvalues and eigenvectors of the symmetric matrices $A^TA \in \mathbb{R}^{M\times M}$ and $AA^T \in \mathbb{R}^{N\times N}$. In fact it can be easily shown that

$$\begin{aligned} A^TA &= V\Sigma^2V^T \\ AA^T &= U\Sigma^2U^T \end{aligned} \tag{12}$$

where A is decomposed as $A = U\Sigma V^T$. From (12) it results that the singular values of A are eigenvalues of the matrices A^TA and AA^T (except those equal to zero for the latter). Additionally the right and left singular vectors are the eigenvectors of A^TA and AA^T respectively, which can differ for a sign (note that the matrix A can be correctly reconstructed provided the correct sign is known). Therefore, on the basis of previous considerations, the singular value decomposition reduces to the eigenvalue problem of a symmetric matrix. In general, the methods for solving such a problem are iterative methods that include two stages:

(i) in the first stage the matrix A is transformed to a matrix B whose structure makes the computation of the eigenvalues and eigenvectors easier. A typical choice, that is assumed here, is a tridiagonal form.

(ii) in the second stage an iterative method is applied to determine the eigenvalues and eigenvectors.

With reference to the second stage the divide and conquer algorithm aims at reducing a complex problem to a singular one [33]. The algorithm is intended to be applied to a tridiagonal and symmetric matrix T of dimension $N \times N$.

The pseudo-code of the algorithm is reported in Algorithm 5.

Algorithm 5 Divide and conquer

Require: $T \in \mathbb{R}^{n \times n}$ tridiagonal symmetric

Divide T as $T = \begin{bmatrix} T_1 & 0 \\ 0 & T_2 \end{bmatrix} + \rho u u^T$

for $i = 1, 2$ **do**

Compute Λ_i, Q_i eigenvalues/vectors of T_i as follows:

if T_i suitably small **then**

Compute eigenvalues/vectors directly

else

Apply recursively divide and conquer algorithm to T_i

end if

end for

Use factorized T_1, T_2 to compute $D + \rho v v^T$, where

$$D = \begin{bmatrix} \Lambda_1 & 0 \\ 0 & \Lambda_2 \end{bmatrix}, v = \begin{bmatrix} Q_1^T & 0 \\ 0 & Q_2^T \end{bmatrix} u \tag{13}$$

Compute eigenvalues Λ by solving the secular equation $D + \rho v v^T = Q \Lambda Q^T$ through Li's algorithm

Compute eigenvectors as $\begin{bmatrix} Q_1 & 0 \\ 0 & Q_2 \end{bmatrix} Q$

3. Experimental Results

The five algorithms described in the previous section have been implemented both in MATLAB on a desktop computer and on a Cortex-M4F microcontroller.

The algorithms have been tested with several matrices $A \in \mathbb{R}^{m \times n}$. To ensure that the algorithms worked in the most general cases, several sets of matrices of increasing size have been chosen, each set with a different row/column ratio. Considering the limited memory of the microcontroller, three sets of matrices have been tested, with a ratio of $m/n = 1, \frac{4}{3}$ and 2, respectively. Further increasing the ratio would have caused the matrices to not fit into the limited memory of the microcontroller without significantly reducing their rank (memory occupation must take into account not only the input and output matrices, but also the intermediate matrices and vectors needed by the various algorithms). Indeed, preference has been given to maintaining comparable ranks between the set of matrices. The low amount of RAM memory also limits the absolute size of matrices, so smaller matrices have been used for the experiments on the microcontroller, with respect to the tests in MATLAB.

The biggest matrix A has been randomly generated, and all the other matrices used are the top-left portion of the full matrix A.

For algorithms requiring a symmetric matrix, the input matrix is converted to a symmetric form as follows. First it is converted to bidiagonal form by applying Algorithm A1, then, being B the upper triangular portion of the bidiagonal matrix, the SVD algorithm is applied to BB^T. In such a way the singular values of the original matrix are the square roots of eigenvalues of the latter tridiagonal matrix.

3.1. MATLAB Implementation

The algorithms have initially been implemented in MATLAB to test their correctness, on 6-core, 12-thread Intel i7 CPU with 32 GB RAM (Intel, Santa Clara, CA, USA).

Figure 1 shows, as a matter of comparison, the timings of the various algorithms as implemented in MATLAB, as functions of the number of columns n for a set of matrices having $m/n = \frac{4}{3}$. Table 1 reports the accuracy of a sample of the same tests, with the algorithms implemented in MATLAB with respect to the MATLAB built-in SVD function. The errors reported are computed as the average of relative errors between the singular value matching pairs, as computed by MATLAB built-in function and the given routines.

As you can see, all the algorithms, with the exception of Demmel–Kahan, ensure the same accuracy of the MATLAB built-in SVD function. This could be expected, as the Demmel–Kahan algorithm was specifically designed to deal well with very small singular values, while the random-generated matrices we employed in our experiments have singular values close to unity.

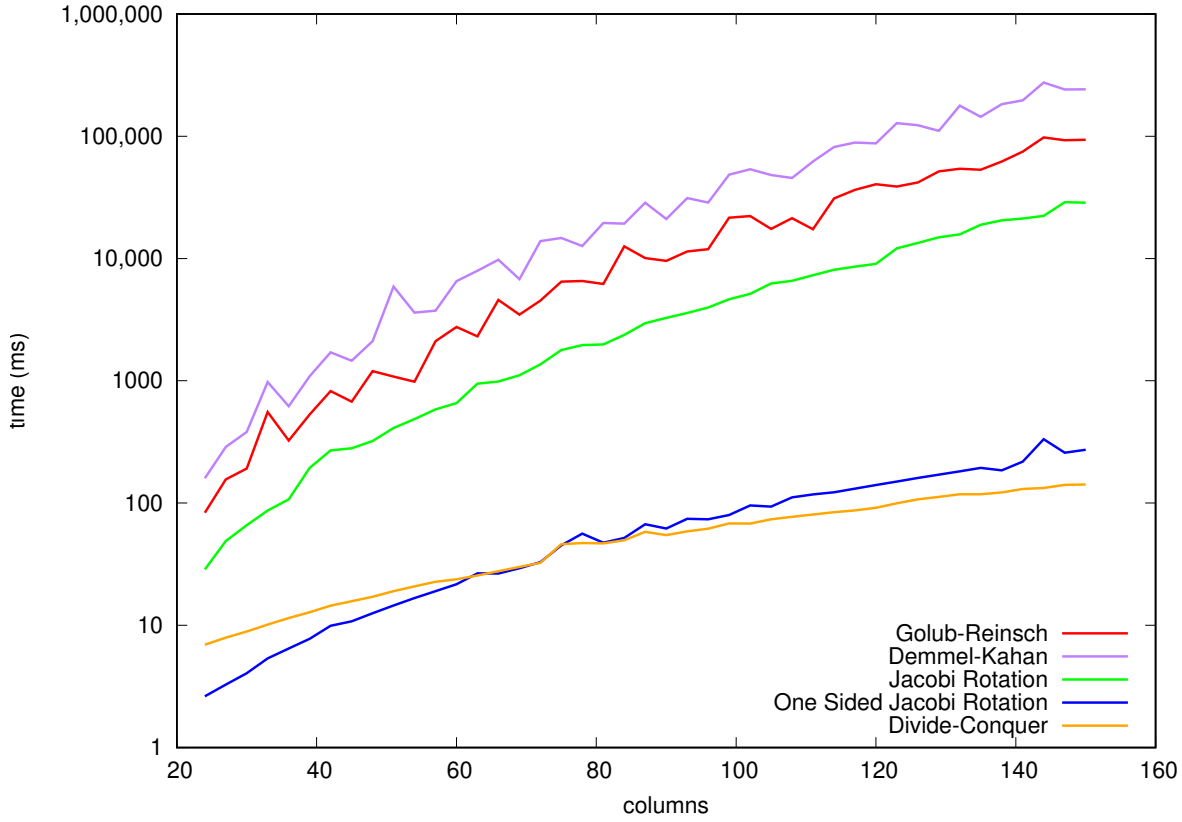


Figure 1. Timings of singular value decomposition (SVD) algorithms implemented in MATLAB for matrices $A \in \mathbb{R}^{m \times n}$ with $m/n = \frac{4}{3}$.

Table 1. Average errors of SVD algorithms implemented in MATLAB vs. MATLAB built-in.

Size	Golub-Reinsch	Demmel-Kahan	Jacobi Rot.	One-Sided J.	Divide-Conquer
32×24	0	2.0×10^{-8}	0	0	0
48×36	0	2.7×10^{-8}	0	0	0
96×72	0	5.4×10^{-8}	0	0	0
128×96	0	7.9×10^{-8}	0	0	0
160×120	0	8.6×10^{-8}	0	0	0
200×150	0	1.9×10^{-7}	0	0	0

3.2. Cortex-M4F Implementation

Cortex-M4F implementations of the different algorithms were tested on a STM32F429ZI microcontroller from STMicroelectronics, Geneva, Switzerland, (https://www.st.com/en/microcontrollers-microprocessors/stm32f429zi.html), mounted on an 32F429IDISCOVERY evaluation board (https://www.st.com/en/evaluation-tools/32f429idiscovery.html).

The STM32F429ZI microcontroller is based on an ARM 32-bit Cortex-M4F [60] CPU clocked at 180 MHz, with 2 MB of flash memory for code and read-only data, and 256 KB of RAM. In addition it has several hardware peripherals that are not relevant to this work.

A main feature of the Cortex-M4F core is the presence of a 32-bit hardware floating-point unit (FPU), as implied by the additional "F" in its name. An FPU is essential for any kind of heavy computational work in the floating-point domain, as is the case for the experiments on SVD performed in this article. The Cortex-M4F FPU is limited to 32 bits (https://developer.arm.com/docs/ddi0439/latest/floating-point-unit/about-the-fpu), so the algorithms have been implemented using single-precision (32 bits) values. Implementing this kind of algorithms on a CPU with no FPU, or with larger precision than that managed by the hardware, would require the use of software floating-point mathematical libraries, that would be prohibitive in an already resource-constrained system.

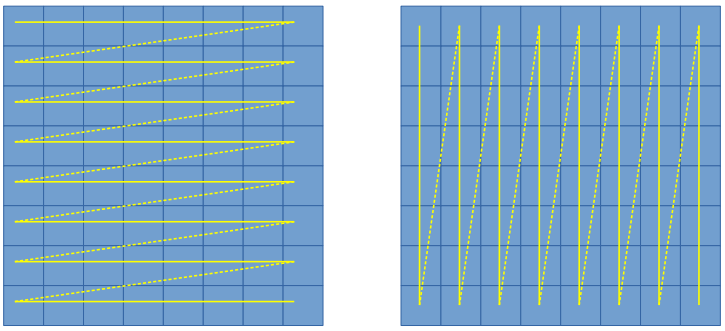
The algorithms were implemented in C language. No particular development environment was used, the code was compiled with the GCC software suite for ARM on a GNU-Linux machine, using a custom makefile and with the aid of the STM32F4xx Standard Peripherals Drivers (STMicroelectronics, Geneva, Switzerland), a set of libraries provided by ST for their specific microcontrollers and encompassing all aspects of hardware management, from low-level initialization to use of hardware peripherals. The firmware is of the "bare-metal" kind, so no real-time operating system (RTOS) or other middlewares have been added.

The hardware system requested no particular design besides what was already provided by the 32F429IDISCOVERY board (STMicroelectronics, Geneva, Switzerland). The device has been clocked at its maximum speed of 180 MHz. The board also integrates all the hardware needed for programming and debugging the microcontroller, namely the ST-LINK/V2 interface (STMicroelectronics, Geneva, Switzerland), offering USB connectivity with the computer. On the computer side, communication with such interface has been established by using OpenOCD (http://openocd.org), a free software for debugging and programming of ARM and other systems. OpenOCD finally acts as a server for GDB, the standard debugger from the GCC suite, used when needed to transfer the code to the device and examine its memory for the results of the tests.

Regarding input and output, read-only data for the program, like the bigger matrix from which smaller ones are generated, or the reference vectors of singular values to compute the accuracy, are stored in the program memory (flash memory) that is more abundant than RAM. Once the program is run for a series of tests, the numerical outputs can be examined through the debugger, by interrupting the program in convenient points. The timing of the single routines is computed by the software itself, using the SysTick timer built in the Cortex-M core.

Besides the optimizations performed by the compiler, special care has been exercised in trying to optimize the critical mathematical routines, while keeping a low usage of program memory and needed RAM, in order to speed up the computation as much as possible while fitting in the constrained resources of the system.

A first optimization comes from choosing the best arrangement of data in memory. Bidimensional matrices can be stored in RAM (which is a linear array of bytes) in two different ways: row-major or column-major order, that is storing data sequentially by row or column respectively (see Figure 2). The former one is the most common way in the C language or anywhere the mathematical convention of having the row as the first index is followed. This has a much more crucial impact in CPUs with cache memory, where cache is filled with sequential data from RAM, and so it gives a huge speed boost in accessing sequential data with respect to sparse ones. As said, a microcontroller has no cache memory, so this is not directly the case; nevertheless a non-negligible advantage exists in accessing data sequentially, due to the load/store assembly instructions with auto-increment, that is instructions that read or write data from memory and increment the address register in a single execution cycle.



(**a**) (**b**)

Figure 2. Matrix stored in memory in (**a**) row-major order and (**b**) column-major order.

As a quantitative example of the effect of row-major or column-major data storing, we can consider the one-sided Jacobi rotation algorithm, that differs from the other ones for accessing the input matrix exclusively by column. Table 2 shows the different timings of the algorithm for the biggest set of matrices, both in absolute times and in percentage of speed increase. As said, the increase in speed is minimal, even if appreciable. The last column shows the time needed to transpose the matrix, in the case it is stored in the opposite way, which is approximately one order of magnitude less than the time improvement obtained. Moreover, often the input matrix to SVD is generated from previous computations, and so it can be generated already in the more convenient order.

Table 2. Timing differences for one-sided Jacobi rotation with respect to data arrangement in memory.

Matrix Size	Row-Major	Column-Major	Speed Increase	Transposition
48×24	19.6 ms	19.1 ms	2.6%	0.07 ms
72×36	73.1 ms	71.9 ms	1.7%	0.15 ms
96×48	192 ms	189.7 ms	1.2%	0.25 ms
120×60	370.6 ms	366.8 ms	1%	0.39 ms
144×72	634 ms	628.6 ms	0.9%	0.55 ms

Besides accessing data in the convenient order, that results in a modest speed increment, and lacking hardware resources to be further exploited, other optimizations must be necessarily obtained by reducing the unneeded computations as much as possible. A significant example is matrix multiplication, one of the most computationally expensive

operations in matrix algebra. Generally speaking, if $C = A \cdot B$, the generic element of C is given by

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j} \tag{14}$$

where n is the number of columns of A. Computing all the elements of C in this way requires a triple nested loop which is very computationally expensive, especially for large matrices.

The number of operations performed for a matrix multiplication can be reduced by observing the properties of the specific matrices. For example a recurrent operation in the exposed algorithms requires the multiplication of a square matrix by its transpose, like $C = A \cdot A'$. In this case (14) becomes

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} a_{j,k}. \tag{15}$$

First we can notice that in the inner loop A is always traversed by row, so we can have the advantage of reading data always in the most convenient order if the matrix is stored in row-major order. Most importantly it can easily be seen that $A \cdot A'$ is a symmetric matrix, so we can actually compute approximately half of its elements, sensibly reducing the number of operations (Figure 3).

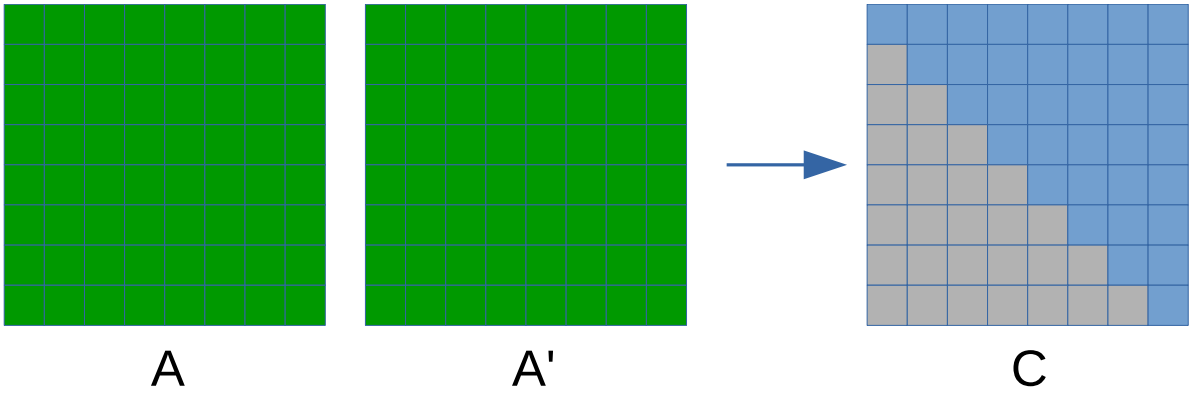


Figure 3. Matrix product of $A \cdot A'$. Green = cells used in computation, blue = cells to be computed, grey = duplicated cells.

A further reduction of the number of operations is possible in the case of $C = A \cdot A'$ where A is an upper-diagonal matrix, another common case in the given algorithms. Given that $a_{i,j} \neq 0$ only for $j = i$ or $j = i + 1$, from (15) follows that $c_{i,j} \neq 0$ only for $j = i - 1$, i or $i + 1$ (indeed the resulting matrix is tridiagonal) and that the only non-zero terms in the sum are those for which $a_{i,k}$ and $a_{j,k}$ are both non-zero . The resulting formula is:

$$c_{i,j} = \begin{cases} a_{i,i} a_{j,i} + a_{i,i+1} a_{j,i+1}, & j = i \\ a_{i,i+1} a_{j,i+1}, & j = i + 1 \\ a_{i,i} a_{j,i}, & j = i - 1 \\ 0, & \text{otherwise} \end{cases} \tag{16}$$

where the $(i + 1)^{th}$ element may not exist. Being also symmetric, the reduction of the previous case also applies (Figure 4).

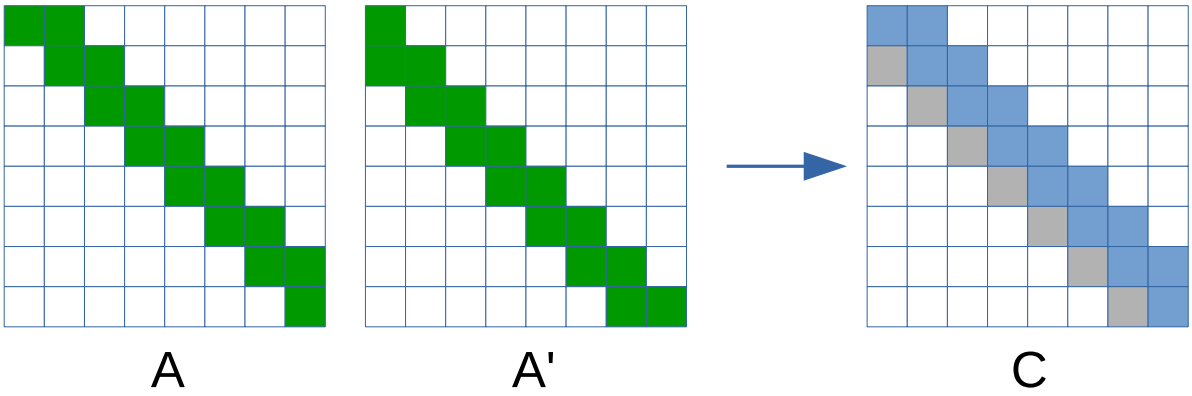


Figure 4. Matrix product of $A \cdot A'$, with A upper-diagonal. Green = cells used in computation, blue = cells to be computed, grey = duplicated cells.

Another special example is the multiplication of a matrix by a Givens matrix, in the form of (A14), to perform a Givens rotation. Let us call G the Givens matrix to avoid confusion with indices, and let's limit for simplicity to the case of left multiplication, as in $C = G \cdot A$. If we initially set $C = A$, it is clear from the definition of G that only rows p and q of C are affected by the multiplication. Moreover, only elements p and q of a given column of A are used in the computation (Figure 5). So the only elements of C that need to be updated are those at rows p and q, and their values are:

$$c_{p,j} = g_{p,p}a_{p,j} + g_{p,q}a_{q,j}, \qquad c_{q,j} = g_{q,p}a_{p,j} + g_{q,q}a_{q,j} \tag{17}$$

for every column j. A similar formula holds for right-side multiplication.

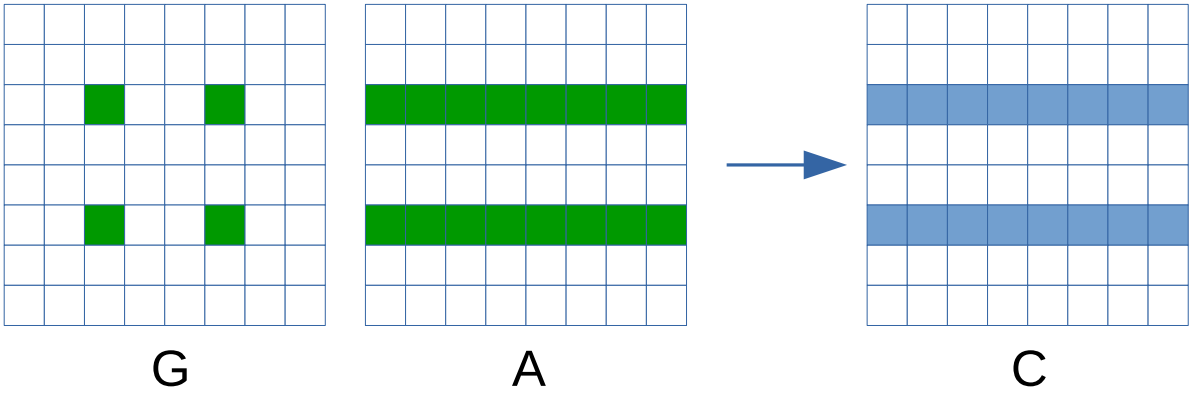


Figure 5. Matrix product for Givens rotation. Green = cells used in computation, blue = cells to be computed.

The complexity of the previous computations, corresponding to matrix products for different special cases of input matrices, can be compared in terms of number of scalar multiplications with respect to the size of input matrices. Results are shown in Table 3, together with the code size of the specific software routines.

Table 3. Number of scalar multiplications in matrix products for different types of $n \times n$ input matrices.

Matrix Product	Scalar Products	Code Size (Bytes)
$C = A \cdot B$, generic	n^3	148
$C = A \cdot A'$	$n^3/2 + n^2/2$	210
$C = A \cdot A'$, A upper-diagonal	$3n - 1$	304
$C = G \cdot A$, Givens rotation	$4n$	108

The impact of such optimizations on computation speed can be measured, in particular those from (15) and (16) and from the property of symmetry (Givens rotation is never actually implemented as a matrix multiplication, since its matrix is expressly constructed to only modify a few elements in the result). Table 4 shows the speed increase for the set of biggest matrices when using two algorithms where these optimizations are mostly relevant.

Table 4. Speed increase from optimized matrix multiplication algorithms.

Matrix Size	Golub-Reinsch	Demmel-Kahan
48×24	21%	16%
72×36	18%	16%
96×48	36%	23%
120×60	44%	16%
144×72	18%	16%

Besides trying to optimize mathematical routines, it is important to avoid the problems arising from the limited power and precision of the microcontroller's FPU. In case of operations that can be carried out in different ways, choosing the right procedure can make a substantial difference. For example the assembly multiplication instruction (VMUL.F32) takes 1 clock cycle to execute, while the division (VDIV.F32) takes 14 cycles (https://developer.arm.com/docs/ddi0439/b/floating-point-unit/fpu-functional-description/fpu-instruction-set). In some cases the compiler can automatically substitute an expensive operation with a cheaper one during the optimization phase of compilation, for example when dividing by a constant.

Another problem arising from the limits of the FPU is the loss of precision in certain operations in the 32-bit domain. For example a recurring problem in the given algorithms is computing $1/\sqrt{1+x^2}$. This kind of operation causes the loss of many significant bits in the original value of x when $x \ll 1$. It must be verified experimentally when this is tolerable and when not. In some cases the loss of precision causes a worsening of the final accuracy by an order of magnitude. A possible solution is switching temporarily to 64-bit precision, then converting back to single precision when the sensitive computation is done. Of course this sensibly increases the execution time, using software libraries instead of the hardware FPU. A better solution is applying the logarithm to the value to be computed, performing intermediate computations in the logarithm domain and finally applying exponentiation. In this case $\log\left(1/\sqrt{1+x^2}\right) = -0.5\log\left(1+x^2\right)$, which can take advantage of a special function in the C mathematical library, called "log1pf", that is optimized to compute $\log\left(1+x\right)$ with high accuracy even if the value of x is near zero. Tests show that using logarithm and exponentiation software libraries while remaining in the 32-bit domain is faster and gives similar or better results than computing the root square in software in double precision.

Figure 6 shows the timings of the full SVD algorithms implemented on the microcontroller for the three sets of matrices of different row/column ratio, with respect to the total matrix size ($m \times n$). The same results are also reported in Table 5, but with the three matrix sets listed separately.

From the experimental results it can be seen that the time of the algorithms is roughly dependent on the total matrix size, but with irregularities suggesting that other factors, like the matrix rank, affect the total time.

On a comparative basis, as you can see the one-sided Jacobi rotation algorithm gives the lowest execution time, compared to the others.

Table 6 reports the accuracy of the specific tests, implemented on the microcontroller, with respect to the MATLAB built-in SVD function. As in Section 3.1, the errors reported are computed as the average of relative errors of matching pairs of singular values, as computed

by MATLAB built-in function and the given routines. As you can see, the accuracy of Cortex-M4F implementation is significantly lower than the equivalent MATLAB code; this is due to the lower precision (32 bits) of the Cortex-M4F hardware floating-point unit. In this case, both the one-sided Jacobi rotation and divide and conquer algorithms achieve a better accuracy than the others.

Finally, Table 7 reports the energy consumption of the tests relative to one matrix set, measured by sampling the voltage and current with an INA226 from Texas Instruments (https://www.ti.com/product/INA226). The INA226 is a current and voltage monitor with I²C interface, with a maximum gain error of 0.1%, maximum input offset of 10 µV and 16-bit resolution. A 0.1 Ω shunt resistor has been used in series with the microcontroller power supply, and the data have been acquired through an external single-board computer.

As you can see, the results are coherent with execution times. As a matter of comparison, Figure 7 shows side-by-side the current consumption of the five algorithms for one of the matrices. The one-sided Jacobi rotation algorithm, besides being faster, clearly has the lowest average current consumption. It is worth noting that the other algorithms exhibit the same pattern at the beginning, corresponding to the Householder's bidiagonalization step. This step has therefore a significant relevance, both in time and energy, for the algorithms that need it.

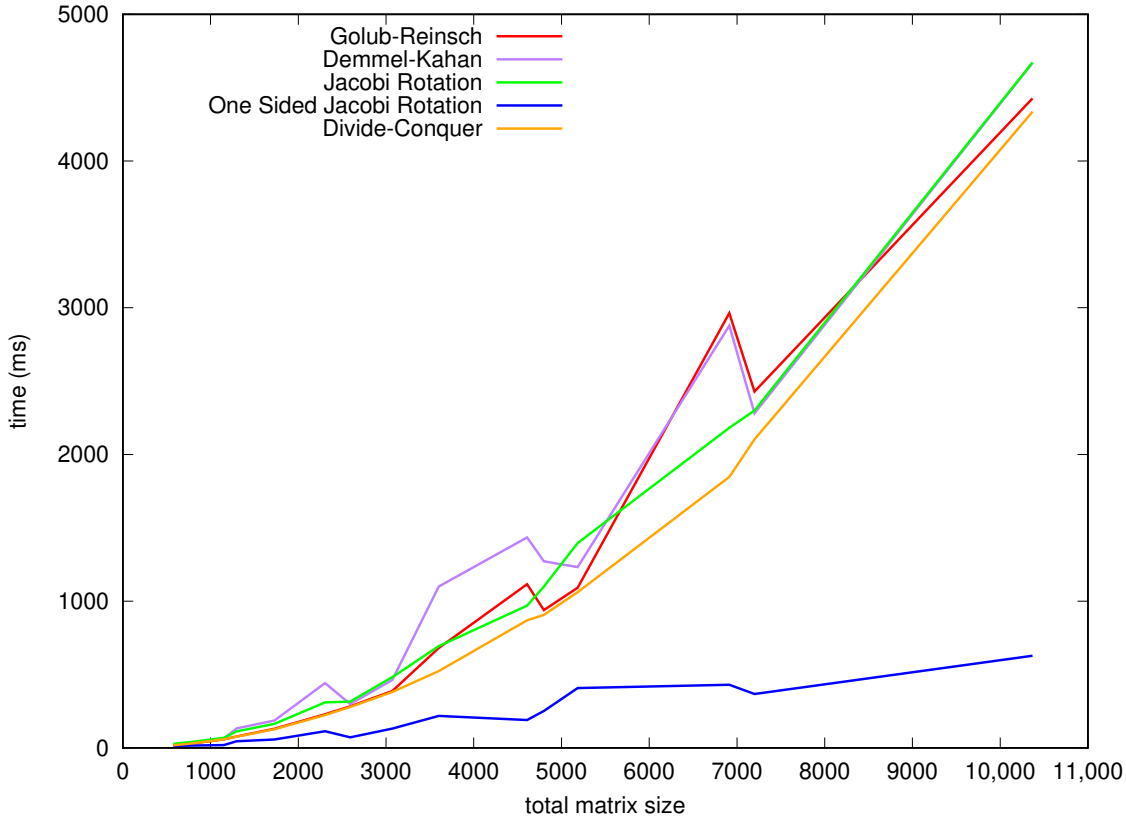


Figure 6. Timings of SVD algorithms on Cortex-M4F with respect to total matrix size $m \times n$.

Table 5. Timings of SVD algorithms on Cortex-M4F.

Size	Golub-Reinsch	Demmel-Kahan	Jacobi Rot.	One-Sided J.	Divide-Conquer
24×24	17 ms	22 ms	28 ms	13 ms	18 ms
36×36	78 ms	133 ms	112 ms	45 ms	76 ms
48×48	229 ms	441 ms	310 ms	114 ms	223 ms
60×60	681 ms	1100 ms	693 ms	218 ms	523 ms
72×72	1092 ms	1232 ms	1396 ms	408 ms	1061 ms
32×24	28 ms	34 ms	39 ms	16 ms	29 ms
48×36	130 ms	186 ms	164 ms	57 ms	127 ms
64×48	388 ms	463 ms	482 ms	131 ms	381 ms
80×60	939 ms	1272 ms	1100 ms	252 ms	906 ms
96×72	2963 ms	2878 ms	2182 ms	430 ms	1847 ms
48×24	58 ms	64 ms	69 ms	19 ms	58 ms
72×36	283 ms	299 ms	316 ms	72 ms	279 ms
96×48	1115 ms	1434 ms	970 ms	190 ms	870 ms
120×60	2429 ms	2280 ms	2299 ms	367 ms	2103 ms
144×72	4427 ms	4669 ms	4672 ms	628 ms	4336 ms

Table 6. Average errors of SVD algorithms on Cortex-M4F vs. MATLAB built-in.

Size	Golub-Reinsch	Demmel-Kahan	Jacobi Rot.	One-Sided J.	Divide-Conquer
24×24	3.8×10^{-7}	7.8×10^{-7}	1.0×10^{-6}	1.9×10^{-7}	2.2×10^{-6}
36×36	4.8×10^{-7}	2.3×10^{-6}	1.7×10^{-6}	3.5×10^{-7}	7.0×10^{-6}
48×48	4.7×10^{-7}	7.1×10^{-6}	3.0×10^{-6}	2.4×10^{-7}	6.6×10^{-6}
60×60	5.3×10^{-7}	4.8×10^{-6}	7.3×10^{-7}	3.0×10^{-7}	4.3×10^{-6}
72×72	4.6×10^{-7}	2.6×10^{-6}	1.7×10^{-6}	3.4×10^{-7}	4.7×10^{-6}
32×24	2.7×10^{-7}	5.7×10^{-7}	2.2×10^{-7}	1.7×10^{-7}	2.9×10^{-7}
48×36	3.6×10^{-7}	2.4×10^{-6}	4.0×10^{-7}	1.7×10^{-7}	2.6×10^{-7}
64×48	2.9×10^{-7}	2.7×10^{-6}	2.9×10^{-7}	1.7×10^{-7}	1.9×10^{-7}
80×60	4.1×10^{-7}	5.9×10^{-6}	5.4×10^{-7}	2.4×10^{-7}	4.0×10^{-7}
96×72	1.6×10^{-6}	6.5×10^{-6}	6.0×10^{-7}	2.7×10^{-7}	3.5×10^{-7}
48×24	3.0×10^{-7}	1.7×10^{-6}	1.6×10^{-7}	1.7×10^{-7}	9.5×10^{-8}
72×36	2.7×10^{-7}	6.5×10^{-7}	3.7×10^{-7}	1.5×10^{-7}	1.4×10^{-7}
96×48	1.0×10^{-6}	8.7×10^{-6}	4.1×10^{-7}	1.8×10^{-7}	1.5×10^{-7}
120×60	5.9×10^{-7}	3.9×10^{-6}	4.8×10^{-7}	2.0×10^{-7}	1.5×10^{-7}
144×72	4.8×10^{-7}	5.3×10^{-6}	5.8×10^{-7}	3.1×10^{-7}	1.3×10^{-7}

Table 7. Energy consumption of SVD algorithms on Cortex-M4F.

Size	Golub-Reinsch	Demmel-Kahan	Jacobi Rot.	One-Sided J.	Divide-Conquer
32×24	5.7 mJ	7.1 mJ	10.9 mJ	1.2 mJ	5.8 mJ
48×36	26.7 mJ	30.6 mJ	38.3 mJ	9.0 mJ	26.1 mJ
64×48	79.7 mJ	87.8 mJ	113.6 mJ	26.5 mJ	78.8 mJ
80×60	193.5 mJ	217.5 mJ	250.7 mJ	50.9 mJ	187.2 mJ
96×72	594.6 mJ	1150.9 mJ	487.8 mJ	86.7 mJ	382.1 mJ

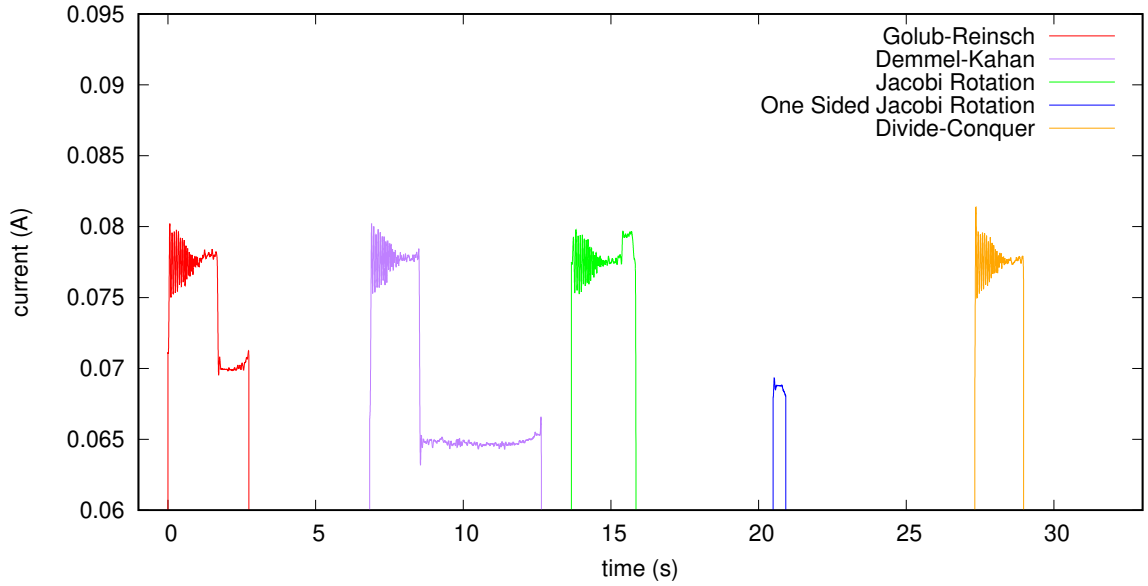


Figure 7. Current consumption of SVD algorithms on Cortex-M4F for the 96×72 matrix, low-pass filtered.

4. Application: Kalman Filtering of Inertial System Data

As a practical application to an embedded system, and as a proof of the advantage of implementing the SVD on this class of systems, we will show how SVD can be used to improve the numerical accuracy of Kalman filtering applied to an inertial measurement unit (IMU). Sensors included in an IMU are typically accelerometers and gyroscopes. Data from such sensors, with proper filtering and data-fusion techniques, allow the computation of the spatial orientation of a moving body (its attitude), namely the relative angles of the body with respect to the spatial axes.

Accelerometers have the problem of high-frequency noise added to the signal, together with additional acceleration terms when a force is applied to the body. Gyroscopes measure the angular rate of change around each axis, that must be integrated to get angular positions; this leads to a sensible drift in the resulting data, because gyroscopes are affected by a bias in the measurement that is non-constant and unpredictable, resulting in a drift increasing with time.

Kalman filter [61] is a common technique used to estimate the state of such a system (namely the current angles along the chosen axes) given a set of discrete measurements over time that are subject to statistical noise. To that end, the system dynamics must be written in form of space-state equations:

$$x_t = Ax_{t-1} + Bu_{t-1} + w_{t-1} \tag{18}$$

$$y_t = Cx_t + v_t \tag{19}$$

where x_t is the unknown system state vector to be estimated, vector u_t is the control input to the system and y_t is the measurement vector. w and v are additive noise on process and measurement data respectively, both assumed to be zero-mean Gaussian processes: $w_t \sim \mathcal{N}(0, Q_t)$, $v_t \sim \mathcal{N}(0, R_t)$. Initial state condition x_0 is also a random variable: $x_0 \sim \mathcal{N}(0, \Pi_0)$.

It is shown in [22] that traditional Kalman filter implementations, like square-root algorithms based on the Cholesky decomposition, are effective with well-conditioned problems but may fail if equations are ill-conditioned, due to roundoff errors, as those algorithms rely on matrix inversion in several places. On the other hand [22] shows how an SVD-based Kalman filter variant works much better in ill-conditioned cases.

In the IMU example, a case of ill-conditioned problem is when multiple sets of the same measurements are available. To better show a practical case, we report results obtained with

an X-NUCLEO-IKS01A2 (https://www.st.com/en/ecosystems/x-nucleo-iks01a2.html) board, a sensor board from ST equipped, among others, with a gyroscope-accelerometer combined sensor and a magnetometer-accelerometer one. The board can be used with a microcontroller unit, in our case we used a NUCLEO-F411RE (https://www.st.com/en/evaluation-tools/nucleo-f411re.html), with a Cortex-M4F-based microcontroller similar to the one used in the previous experiments.

For this hardware system the firmware has been developed using the STM32CubeIDE development platform, version 1.4.0. An additional software package, X-CUBE-MEMS1, has been installed in the IDE through its integrated software package manager. This package supports the communication with the specific sensors in the X-NUCLEO-IKS01A2 inertial system. In this case the microcontroller (STM32F411RE) is clocked at 84 MHz. Like in the previous experiment, the firmware is "bare-metal", with no RTOS or other middlewares added. Communication with the board for programming and debugging is entirely managed by the IDE in this case.

Having two accelerometers available, it is convenient to use both data to get a better estimate of the state, given the noisy nature of the device. Mathematically, matrices in Equations (18) and (19) can be expressed as:

$$A = \begin{bmatrix} 1 & -\Delta t & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -\Delta t \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad B = \begin{bmatrix} \Delta t & 0 \\ 0 & 0 \\ 0 & \Delta t \\ 0 & 0 \end{bmatrix} \quad C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1+\delta & 0 & 0 & 0 \\ 0 & 0 & 1+\delta & 0 \end{bmatrix} \tag{20}$$

and variables as:

$$x = \begin{bmatrix} \hat{\phi} \\ \hat{b}_\phi \\ \hat{\theta} \\ \hat{b}_\theta \end{bmatrix} \quad u = \begin{bmatrix} \dot{\phi}_G \\ \dot{\theta}_G \end{bmatrix} \quad y = \begin{bmatrix} \hat{\phi}_{A1} \\ \hat{\theta}_{A1} \\ \hat{\phi}_{A2} \\ \hat{\theta}_{A2} \end{bmatrix} \tag{21}$$

Here the system state includes the estimated roll and pitch angles ($\hat{\phi}$, $\hat{\theta}$) along x and y axes respectively and the estimated gyroscope bias on both angles ($\hat{b}_\phi$, $\hat{b}_\theta$); system inputs include angular rates from gyroscope ($\dot{\phi}_G, \dot{\theta}_G$); Δt is the period of acquired data; system measurements are given by estimated angles from both accelerometer data ($\hat{\phi}_{A1}, \hat{\theta}_{A1}, \hat{\phi}_{A2}, \hat{\theta}_{A2}$).

The δ factor accounts for the fact that the two series of accelerometer data are very similar numerically, leading to an ill-formed problem: the matrix is now nearly singular, due to redundancy of system measurement variables. As shown in [22], we can expect that resolution of Kalman filter equations by means of conventional algorithms will suffer from roundoff errors when δ becomes small enough to be comparable to machine precision. We can also expect that the SVD-based algorithm will work better in such cases.

The firmware on the board has been used to export a batch of measurements (gyroscope, accelerometer 1, accelerometer 2) to the computer, so that the data could be examined in Matlab in a reproducible way. The data have a duration of about 90 s, with a Δt of 10 ms, and have been obtained by moving the device through a set of reference positions, namely 0, 90 and -90 degrees on the three axes.

Then the Kalman filter has been computed in Matlab with both algorithms (conventional and SVD-based), repeating the experiment for various decreasing values of δ, until approaching values comparable to machine precision.

Finally the accuracy of the estimated state has been evaluated by computing the mean absolute error (in degrees) with respect to the reference intervals of known orientations, for all the different values of δ.

Table 8 shows that the two algorithms give very similar results down to a certain value of δ, but at lower values the conventional algorithm fails ("NaN" means Not-a-Number, an error condition indicating an invalid numeric value). The SVD-based algorithm, conversely,

keeps working with neglectable difference for other two orders of magnitude of δ, starting to degrade at a much lower value.

Table 8. Matlab: mean absolute errors (ϕ, θ) of estimated angles (degrees).

Algorithm	$\delta = 10^{-8}$	$\delta = 10^{-9}$	$\delta = 10^{-10}$	$\delta = 10^{-11}$	$\delta = 10^{-12}$
Conventional	1.7337 1.1066	1.7337 1.1066	NaN NaN	NaN NaN	NaN NaN
SVD-based	1.7337 1.1066	1.7337 1.1066	1.7357 1.1065	1.7113 1.3738	23.6834 27.0080

To perform a similar experiment with the limited capabilities of an embedded system, the same algorithms for the Kalman filter have been implemented in a C program on the same set of off-line data, using the custom SVD algorithms discussed in the previous sections and with floating-point numbers in single precision (32 bit), that is the same precision of the floating-point hardware of the Cortex-M4F architecture. To just focus on the difference between conventional and SVD-based Kalman filter, we used one only SVD algorithm, namely the One-Sided Jacobi, that proved to be the best one under several criteria (Section 3). The other algorithms have shown to give similar results.

Preliminary tests have shown that with limited numeric precision, as in this case, the conventional Kalman algorithm is much more sensible to variations of initial conditions. Actually, the difference between the two Kalman implementations is evident even in the single accelerometer case, by varying the relative values of the covariance matrices involved.

As a clarifying example, a series of tests is shown with fixed values $\Pi_0 = I_4$ and $Q = I_4$, data from a single accelerometer and varying values of R. Results are shown in Table 9, which highlights a trend similar to the Matlab case: the two algorithms are very similar for some cases, but the SVD-based one works uniformly on a wider range of parameters.

Table 9. Single precision C code: mean absolute errors (ϕ, θ) of estimated angles (degrees).

Algorithm	$R = I_2$	$R = 0.8I_2$	$R = 0.6I_2$	$R = 0.4I_2$	$R = 0.2I_2$
Conventional	2.7197 1.3642	2.7191 1.3820	2.7124 1.4480	NaN NaN	NaN NaN
SVD-based	2.7186 1.3477	2.7173 1.3494	2.7159 1.3516	2.7143 1.3543	2.7124 1.3580

About the actual performances on a Cortex-M4F microcontroller, the matrices involved in the SVD-based Kalman algorithm are much smaller than the ones in Table 5, having a maximum size of 8x4 in the case of double accelerometer. An extrapolation of the results in Table 5 leads to an estimate of fractions of millisecond, so the computation of Kalman filter using the SVD-based variant should be possible in real time in most applications.

5. Conclusions

This paper presents a comparative study of five of the most common SVD algorithms, namely Golub–Reinsch, Demmel–Kahan, Jacobi rotation, one-sided Jacobi rotation, divide and conquer. The aim of this comparative study is to find the most suitable algorithm for an embedded system. The chosen algorithms have been investigated starting from a theoretical perspective giving a detailed mathematical description, which provides crucial hints for the practical implementations. The algorithms have been initially written in MATLAB to verify their correctness, then implemented on a Cortex-M4F microcontroller to achieve optimized results for low-resource embedded systems. The comparison of the Cortex-M4F implementations shows that the one-sided Jacobi rotation outperforms the other algorithms in terms of speed, accuracy and energy consumption. Moreover the divide and conquer algorithm shows similar accuracy results. Finally the SVD has been applied to an example case, showing that it can improve the accuracy of algorithms where the data are ill-conditioned, while traditional implementations may fail.

Author Contributions: Conceptualization, M.A., G.B., L.F., L.M. and C.T.; investigation, M.A., L.F., L.M. and C.T.; methodology, M.A., G.B., L.F., L.M. and C.T.; project administration, P.C. and C.T.; software, M.A., G.B., L.F. and L.M.; supervision, P.C. and C.T.; validation, M.A., G.B. and L.M.; visualization, M.A. and L.F.; writing—original draft, M.A., L.F., L.M. and C.T.; writing—review and editing, M.A., G.B., P.C., L.F., L.M. and C.T. 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: Not applicable.

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

Appendix A. Matrix Algebra Operations

Appendix A.1. Householder Transformation

It is based on an $(n \times n)$ symmetric matrix of the form

$$U = I - 2\,\frac{uu^T}{u^T u} \tag{A1}$$

where u is the Householder vector. A matrix of this kind is also called a Householder reflection, due to the following specific geometry property. A given vector x can always be represented as

$$x = \alpha u + w,\ \alpha \in \mathbb{R} \tag{A2}$$

where w is orthogonal to u. By applying the transformation U to x one gets the vector

$$Ux = (I - 2\frac{uu^T}{u^T u})(\alpha u + w) = -\alpha u + w\,, \tag{A3}$$

representing the reflected vector of x with respect to the hyperplane spanned by w. Thanks to this property the Householder transformation can be used to zero selected components of a vector. To show this let us choose u such that

$$u = x \pm \|x\| e_1 \tag{A4}$$

where $e_1 = (1, 0, \ldots, 0)^T$, and $\|\cdot\|$ is the Euclidean norm of a vector. By applying the transformation U so obtained to x, then it results

$$Ux = \mp \|x\| e_1 \tag{A5}$$

meaning that all the components of x but the first one are zeroed. A common choice to avoid errors when $x_1 \approx \|x\|$ is to pose $u = x + \text{sign}(x_1)\|x\|e_1$. The method can be generalized to the k-th component as follows. For the generic index $k \in \{1, \ldots, n\}$, let us define

$$u_k = [0, \ldots, 0, x_k \pm s, x_{k+1}, \ldots, x_n]^T \tag{A6}$$

where $s = \sqrt{x_k^2 + \ldots + x_n^2}$. The resulting Householder matrix U_k when applied to x gives

$$U_k\, x = [x_1, x_2, \ldots, x_{k-1}, \mp s, 0, \ldots, 0]^T\,, \tag{A7}$$

that is U_k leaves the first $k-1$ components unchanged, changes the k-th component and zeroes all the residual $n-k$ components. It can be easily shown that U_k has the block form

$$U_k = \begin{bmatrix} I_{k-1} & 0 \\ 0 & \widehat{U} \end{bmatrix} \tag{A8}$$

where $\widehat{U}$ acts only on the last $n-k+1$ components of x by zeroing them all but the k-th component. Similarly, the transformation

$$x^T V = x^T (I - 2\frac{v^T v}{v v^T}) \quad , \tag{A9}$$

where v is a row vector, acts on the row vector x^T by zeroing some of its components. A useful property of U is that it is not necessary for the matrix to be explicitly derived, indeed the transformation

$$Ux = (I - 2\frac{uu^T}{u^T u})x = x - 2\frac{u^T x}{u^T u}u \tag{A10}$$

can be written in terms of u alone.

A mathematical description of the algorithm is shown in Algorithm A1.

Algorithm A1 Householder bidiagonalization

Require: $A \in \mathbb{R}^{m \times n}$

for $k = 1, \ldots, n-1$ **do**

- Determine Householder matrix U_k such that

$U_k\, x = [x_1, x_2, \ldots, x_{k-1}, \mp s, 0, \ldots, 0]^T$

$A \leftarrow U_k\, A$

if $k < n-1$ **then**

- Determine Householder matrix V_k such that

$x^T\, V_k = [x_1, x_2, \ldots, x_k, \mp s, 0, \ldots, 0]$

$A \leftarrow A\, V_k$

end if

end for

Appendix A.2. Householder's Bidiagonalization

Given the matrix $A \in \mathbb{R}^{n \times m}$ $(n > m)$ the bidiagonal form

$$A = UBV^T,\ U \in \mathbb{R}^{n \times n},\ V \in \mathbb{R}^{m \times m} \tag{A11}$$

with

$$B = \begin{bmatrix} \widehat{B} \\ 0 \end{bmatrix} \in \mathbb{R}^{n \times m},$$

$$\widehat{B} = \begin{bmatrix} \psi_1 & \phi_1 & 0 & \cdots & 0 \\ 0 & \psi_2 & \phi_2 & & \\ \vdots & & \ddots & \ddots & \\ & & & \psi_{m-1} & \phi_{m-1} \\ 0 & & & & \psi_m \end{bmatrix} \in \mathbb{R}^{m \times m}, \tag{A12}$$

exists. The matrix B can be obtained from A by the successive orthogonal transformation

$$B = U_m \cdots U_1\ A\ V_1 \cdots V_{m-2} \tag{A13}$$

where U_k, V_k are Householder's matrices. In particular for the k-th step:

(i) an Householder matrix U_k can be defined for zeroing all the last $n-k$ components of the k-th column of B;

(ii) an Householder matrix V_k can be defined for zeroing all the $m-k-1$ components of the k-th row of B.

At the end of the process the diagonalization (A11) is achieved with $U = U_m \cdots U_1$, $V = V_1 \cdots V_{m-2}$. The computational cost of this process is about $\mathcal{O}(n^3)$.

Appendix A.3. Jacobi Rotation

Householder transformation is useful for zeroing a number of components of a vector. However when it is necessary to zero elements more selectively, Jacobi (or Givens) rotation is able to zero a selected component of a vector. It is based on the Jacobi matrix, also called Givens matrix, denoted by $J(p,q,\theta)$, of the form

$$J(p,q,\theta) = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & & & & & & \\ 0 & \cdots & c & \cdots & s & \cdots & 0 \\ \vdots & & & & & & \\ 0 & \cdots & -s & \cdots & c & \cdots & 0 \\ \vdots & & & & & & \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \\ & & \text{p} & & \text{q} & & \end{bmatrix} \begin{matrix} \\ \\ \text{p} \\ \\ \text{q} \\ \\ \end{matrix} \tag{A14}$$

where $c = \cos\theta$ and $s = \sin\theta$. Premultiplication of a vector by $J(p,q,\theta)^T$ corresponds to a counterclockwise rotation of θ in the (p,q) plane, that zeroes the q components of the resulting vector y. Indeed, if $x \in \mathbb{R}^n$ and

$$y = J(p,q,\theta)^T x \,, \tag{A15}$$

then

$$y_j = \begin{cases} cx_p - sx_q, & j = p \\ sx_p + cx_q, & j = q \\ x_j, & j \neq p,q \end{cases} \,. \tag{A16}$$

From (A16) it is clear that y_q can be forced to zero by setting

$$c = \frac{x_p}{\sqrt{x_p^2 + x_q^2}}, \; s = \frac{-x_q}{\sqrt{x_p^2 + x_q^2}} \,. \tag{A17}$$

The Jacobi matrix, when applied as a similarity transformation to a symmetric matrix A,

$$B = J(p,q,\theta)^T A \, J(p,q,\theta) \,, \tag{A18}$$

rotates rows and columns p and q of A through the angle θ so that the (p,q) and (q,p) entries are zeroed.

Appendix A.4. QR Factorization

This factorization is a fundamental step in QR iteration algorithms. The QR factorization of an $(m \times n)$ matrix A is given by

$$A = QR \tag{A19}$$

where $Q \in \mathbb{R}^{m \times m}$ is orthogonal and $R \in \mathbb{R}^{m \times n}$ is upper triangular. Having derived the properties of Householder transformation, it is straightforward to show that the upper triangular matrix R can be obtained by successive transformations

$$H_n H_{n-1} \ldots H_1 A = R \tag{A20}$$

where $H_1, H_2, \ldots, H_n$ are Householder matrices, and so by setting $Q = H_1 \ldots H_n$ we obtain $A = QR$.

Appendix A.5. QR Iteration

This algorithm is based on the QR factorization and the "power method". Assuming $A \in \mathbb{R}^{n\times n}$ is a symmetric matrix then, by the symmetric Schur decomposition, there exists a real orthogonal Q such that

$$Q^T A Q = \text{diag}(\lambda_1, \ldots, \lambda_n) \tag{A21}$$

with $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n$. Given a vector $q^{(0)} \in \mathbb{R}^n$, such that $\|q^{(0)}\| = 1$, the power method produces a sequence of vectors $q^{(k)}$ as follows:

$$\begin{aligned} &\textbf{for } k = 1, 2, \ldots \textbf{ do} \\ &\quad z^{(k)} = A\, q^{(k-1)} \\ &\quad q^{(k)} = z^{(k)} / \|z^{(k)}\| \\ &\quad \lambda^{(k)} = [q^{(k)}]^T A\, q^{(k)} \\ &\textbf{end for} \end{aligned} \tag{A22}$$

The power method states that if $q^{(0)} \neq 0$ and $\lambda_1 > \lambda_2 \geq \ldots \geq \lambda_n$ then $q^{(k)}$ converges to an eigenvector and $\lambda^{(k)}$ to the corresponding eigenvalue.

This method can be generalized to solve the eigenvalue problem of a symmetric matrix. To this end let us consider an $(n \times n)$ matrix Q_0 with orthonormal columns and a sequence of matrices $\{Q_k\}$ generated as follows:

$$\begin{aligned} &\textbf{for k} = 1, 2, \ldots \textbf{ do} \\ &\quad Z_k = A\, Q_{k-1} \\ &\quad Q_k\, R_k = Z_k \ (QR \textit{ factorization}) \\ &\textbf{end for} \end{aligned} \tag{A23}$$

where the QR factorization is applied at each step to obtain the matrices Q_k and R_k, then (A23) defines the so-called orthogonal iteration. It can be shown [62] that the matrices T_k defined by

$$T_k = Q_k^T A\, Q_k \tag{A24}$$

are converging to a diagonal form whose values $\{\lambda_1^{(k)}, \ldots, \lambda_n^{(k)}\}$ converge to $\{\lambda_1, \ldots, \lambda_n\}$.

From definition of T_{k-1} we have

$$T_{k-1} = Q_{k-1}^T A\, Q_{k-1} = Q_{k-1}^T (A\, Q_{k-1}) = Q_{k-1}^T (Q_k R_k) \tag{A25}$$

where the QR factorization of $A\, Q_{k-1}$ has been applied. Similarly, using (A23) and orthogonality of Q_{k-1}, one gets

$$T_k = Q_k^T A Q_k = (Q_k^T A Q_{k-1})(Q_{k-1}^T Q_k) = R_k (Q_{k-1}^T Q_k) \quad . \tag{A26}$$

Defining $U_k = Q_{k-1}^T Q_k$ the algorithm (A23) can be rewritten as

$$\begin{aligned} &T_0 = U_0^T A U_0 \\ &\textbf{for } k = 1, 2, \ldots \textbf{ do} \\ &\quad U_k R_k = T_{k-1} \ \ (QR \textit{ factorization}) \\ &\quad T_k = R_k U_k \\ &\textbf{end for} \end{aligned} \tag{A27}$$

where $U_0 \in \mathbb{R}^{n \times n}$ is orthogonal. Since $T_k = R_k U_k = U_k^T (U_k R_k) U_k = U_k^T T_{k-1} U_k$, it follows by induction that

$$T_k = (U_0 U_1 \ldots U_k)^T A \, (U_0 U_1 \ldots U_k) \tag{A28}$$

and T_k converges to the diagonal form (A21). The iteration (A27) establishes the so-called QR *iteration algorithm* for symmetric matrices.

The main limitation of QR algorithm is that it is only valid for symmetric matrices, such as $A^T A$, thus the method cannot be directly applied to the matrix A.

Appendix A.6. Implicit QR Method with Shift

The symmetric QR algorithm (A27) can be made more efficient in two ways:

(i) by choosing U_0 such that $U_0 A \, U_0 = T_0$ is tridiagonal. In this way all T_k in (A27) are tridiagonal and this reduces the complexity of the algorithm to $\mathcal{O}(n^2)$; once the Householder's algorithm is applied to the matrix A giving the bidiagonal matrix $\hat{B}$, the tridiagonal form T_0 can be easily obtained as $T_0 = \hat{B}^T \hat{B}$.

(ii) by introducing a shift in the iteration of (A27): with this change the convergence to diagonal form proceeds at a cubic rate. This result is based on the following facts:

   (a) if $s \in \mathbb{R}$ and $T - sI = QR$ is the shifted version of T then $T_+ = RQ + sI$ is also tridiagonal;

   (b) if s is an eigenvalue of T, $s \in \lambda(T)$, the last column of T_+ equals $s\, e_n = s(0 \ldots 1)^T$, that is $T_+(n, n) = s$.

With regard to the second point the algorithm (A27) modifies to the following

$$\begin{aligned} &T = \hat{B}^T \hat{B} \ \ (\textit{tridiagonal}) \\ &\textbf{for } k = 0, 1, \ldots \textbf{ do} \\ &\quad \textit{Determine real shift } \mu \\ &\quad UR = T - \mu I \ \ (QR \textit{ factorization}) \\ &\quad T = RU + \mu I \\ &\textbf{end for} \end{aligned} \tag{A29}$$

where μ is a good approximate eigenvalue and

$$T = \begin{bmatrix} a_1 & b_1 & \cdots & 0 \\ b_1 & a_2 & \cdots & 0 \\ & \ddots & \ddots & \\ & & & b_{n-1} \\ & & b_{n-1} & a_n \end{bmatrix} . \tag{A30}$$

An effective choice is to shift by the eigenvalue of

$$\begin{bmatrix} a_{n-1} & b_{n-1} \\ b_{n-1} & a_n \end{bmatrix} \tag{A31}$$

known as the Wilkinson shift and given by

$$\mu = a_n + d - \text{sign}(d)\sqrt{d^2 + b_{n-1}^2},\ d = (a_{n-1} - a_n)/2 \quad . \tag{A32}$$

If μ is a good approximation of the eigenvalue s, then the term b_{n-1} will be smaller after a QR step with shift μ. It has been shown [63] that with this shift strategy, (A28) is cubically convergent.

A pseudo-code of the algorithm in shown in Algorithm A2.

Algorithm A2 QR iteration with shift

Require: $A \in \mathbb{R}^{m \times n}$

Apply Algorithm A1 to obtain bidiagonal $\hat{B}$

$T = \hat{B}^T \hat{B}$ tridiagonal

for $k = 1, \ldots$ **do**

- Select $B_{22}(2 \times 2)$: block matrix at the right bottom of $\hat{B}^T \hat{B}$
- Compute eigenvalues λ_1, λ_2 of B_{22}
- Determine shift $\mu = \min(\lambda_1, \lambda_2)$

$T = \mu T = UR$ *(QR factorization)*

$T = RU + \mu I$

end for

It is possible to execute the transition to $T = RU + \mu I$ without explicitly forming the matrix $T - \mu I$, thus giving the implicit shift version [62]. This is achieved by a Given rotation matrix in which $c = \cos(\theta)$ and $s = \sin(\theta)$ are such that

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T \begin{bmatrix} a_1 - \mu \\ b_1 \end{bmatrix} = \begin{bmatrix} x \\ 0 \end{bmatrix} \quad . \tag{A33}$$

However if we set $J_1 = J(1, 2, \theta)$ we have

$$T \leftarrow J_1^T T J_1 = \begin{bmatrix} x & x & + & 0 & \cdots & 0 \\ x & x & x & & & \vdots \\ + & x & x & & & 0 \\ & & & \ddots & & \\ & & & & x & x \\ 0 & \cdots & \cdots & 0 & x & x \end{bmatrix} \quad . \tag{A34}$$

where the two nonzero elements "+" out of the tridiagonals appears. To "chase" these unwanted elements, we can apply rotations $J_2, \ldots, J_{n-1}$ of the form $J_i = J(i, i+1, \theta_i)$, $i = 2, \ldots, n-1$, such that if $z = J_1 J_2 \ldots J_{n-1}$ then $Z^T T Z$ is tridiagonal. In such a way, it can be shown that the tridiagonal matrix produced by this implicit shift technique is the same as the tridiagonal matrix obtained by the explicit method.

A description of implicit QR method with shift is reported in Algorithm A3, while the pseudo-code for Golub–Reinsch algorithm is described in Algorithm 1.

Algorithm A3 QR iteration with implicit shift

Require: $A \in \mathbb{R}^{m\times n}$

Apply Algorithm A1 to obtain bidiagonal $\hat{B}$

$T = \hat{B}^T\hat{B}$ tridiagonal

Compute the eigenvalue μ of $\begin{bmatrix} T_{m-1,m-1} & T_{m-1,m} \\ T_{m,m-1} & T_{m,m} \end{bmatrix}$ that is closer to $T_{m,m}$.

Choose the Givens matrix $J_1 = J(1,2,\theta)$ such that

$$J_1^T \begin{bmatrix} a_1 - \mu \\ b_1 \end{bmatrix} = \begin{bmatrix} x \\ 0 \end{bmatrix}$$

$T = J_1^T T J_1$

for $k = 2, \ldots, m-1$ **do**

$J_k = J(k, k+1, \theta_k)$

$Z = J_1 J_2 \ldots J_k$

$T = Z^T T Z$

end for

Appendix A.7. QR Iteration with Zero-Shift

This algorithm is a variation of the QR standard method with shift, called implicit zero-shift QR algorithm, since it corresponds to the standard algorithm when $\sigma = 0$, which computes all the singular values of a bidiagonal matrix, with guaranteed high relative accuracy.

To show the algorithm, let us take $\sigma = 0$ and refer to a 4×4 matrix example. From (A26) one gets $\tan\theta_1 = -b_{12}/b_{11}$ so that the result of the first rotation is

$$B^{(1)} = BJ_1 = \begin{bmatrix} b_{11}^{(1)} & 0 & & \\ b_{21}^{(1)} & b_{22}^{(1)} & b_{23} & \\ & & b_{33} & b_{34} \\ & & & b_{44} \end{bmatrix}. \tag{A35}$$

We see that $(1,2)$ entry is zero and, as it will propagate through the rest of the algorithm, this is the key of its effectiveness. After the rotation by J_2 we have

$$B^{(2)} = J_2 B J_1 = \begin{bmatrix} b_{11}^{(2)} & b_{12}^{(2)} & b_{13}^{(2)} & \\ 0 & b_{22}^{(2)} & b_{23}^{(2)} & \\ & & b_{33} & b_{34} \\ & & & b_{44} \end{bmatrix} \tag{A36}$$

where

$$\begin{bmatrix} b_{12}^{(2)} & b_{13}^{(2)} \\ b_{22}^{(2)} & b_{23}^{(2)} \end{bmatrix} = \begin{bmatrix} \sin\theta_2 b_{22}^{(1)} & \sin\theta_2 b_{23} \\ \cos\theta_2 b_{22}^{(1)} & \cos\theta_2 b_{23} \end{bmatrix} \tag{A37}$$

is a rank one matrix. Postmultiplication by J_3 to zero out the $(1,3)$ entry will also zero out the $(2,3)$ entry:

$$B^{(3)} = J_2 B J_1 J_3 = \begin{bmatrix} b_{11}^{(2)} & b_{12}^{(3)} & 0 & \\ 0 & b_{22}^{(3)} & 0 & \\ & b_{32}^{(3)} & b_{33}^{(3)} & b_{34} \\ & & & b_{44} \end{bmatrix} . \tag{A38}$$

Rotation by J_4 just repeats the situation: the submatrix of $J_4 J_2 B J_1 J_3$ consisting of row 2 and 3 and columns 3 and 4 is rank one, and rotation by J_5 zeroes out the $(3,4)$ entry as well as the $(2,4)$ entry. This engine repeats itself for the length of the matrix. Thus at each step of zero-shift algorithm a transformation is applied which takes f and g as input and returns r, $cs = \cos\theta$ and $sn = \sin\theta$ such that

$$\begin{bmatrix} cs & sn \\ -sn & cs \end{bmatrix} \begin{bmatrix} f \\ g \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix} . \tag{A39}$$

References

1. MacDuffee, C.C. *The Theory of Matrices*; Springer Science & Business Media: Berlin/Heidelberg, Germany, 2012; Volume 5.
2. Autonne, L. Sur les groupes linéaires, réels et orthogonaux. *Bull. Soc. Math. Fr.* **1902**, *30*, 121–134. [CrossRef]
3. Eckart, C.; Young, G. A principal axis transformation for non-Hermitian matrices. *Bull. Am. Math. Soc.* **1939**, *45*, 118–121. [CrossRef]
4. Yücelbaş, Ş.; Yücelbaş, C.; Tezel, G.; Özşen, S.; Yosunkaya, Ş. Automatic sleep staging based on SVD, VMD, HHT and morphological features of single-lead ECG signal. *Expert Syst. Appl.* **2018**, *102*, 193–206. [CrossRef]
5. Sreeja, S.; Sahay, R.R.; Samanta, D.; Mitra, P. Removal of eye blink artifacts from EEG signals using sparsity. *IEEE J. Biomed. Health Inform.* **2017**, *22*, 1362–1372. [CrossRef]
6. Mukhopadhyay, S.K.; Ahmad, M.O.; Swamy, M. SVD and ASCII Character Encoding-Based Compression of Multiple Biosignals for Remote Healthcare Systems. *IEEE Trans. Biomed. Circuits Syst.* **2017**, *12*, 137–150. [CrossRef]
7. Biagetti, G.; Crippa, P.; Falaschetti, L.; Orcioni, S.; Turchetti, C. Reduced complexity algorithm for heart rate monitoring from PPG signals using automatic activity intensity classifier. *Biomed. Signal Process. Control.* **2019**, *52*, 293–301. [CrossRef]
8. Biagetti, G.; Crippa, P.; Falaschetti, L.; Orcioni, S.; Turchetti, C. Human activity recognition using accelerometer and photoplethysmographic signals. In *International Conference on Intelligent Decision Technologies*; Springer International Publishing: Cham, Switzerland, 2017; pp. 53–62.
9. Biagetti, G.; Crippa, P.; Falaschetti, L.; Orcioni, S.; Turchetti, C. An efficient technique for real-time human activity classification using accelerometer data. In *International Conference on Intelligent Decision Technologies*; Springer International Publishing: Cham, Switzerland, 2016; pp. 425–434.
10. Bacà, A.; Biagetti, G.; Camilletti, M.; Crippa, P.; Falaschetti, L.; Orcioni, S.; Rossini, L.; Tonelli, D.; Turchetti, C. CARMA: A robust motion artifact reduction algorithm for heart rate monitoring from PPG signals. In Proceedings of the 2015 23rd European Signal Processing Conference (EUSIPCO), Nice, France, 31 August–4 September 2015; pp. 2646–2650.
11. Yang, G.; Zeng, R.; Dong, A.; Yan, X.; Tan, Z.; Liu, Y. Research and Application of 3D Face Modeling Algorithm Based on ICP Accurate Alignment. In *Journal of Physics: Conference Series*; IOP Publishing: Bristol, UK, 2018.
12. Zear, A.; Singh, A.K.; Kumar, P. A proposed secure multiple watermarking technique based on DWT, DCT and SVD for application in medicine. *Multimed. Tools Appl.* **2018**, *77*, 4863–4882. [CrossRef]
13. Turajlic, E.; Begović, A.; Škaljo, N. Application of Artificial Neural Network for Image Noise Level Estimation in the SVD domain. *Electronics* **2019**, *8*, 163. [CrossRef]
14. Liu, Z.; Dickson, K.; McCanny, J.V. Application-specific instruction set processor for SoC implementation of modern signal processing algorithms. *IEEE Trans. Circuits Syst. Regul. Pap.* **2005**, *52*, 755–765.
15. Jena, J.J.; Patro, M.; Girish, G. A SVD Based Pattern Matching Approach for Color Image Retrieval. In Proceedings of the 2018 Second International Conference on Electronics, Communication and Aerospace Technology (ICECA), Coimbatore, India, 29–31 March 2018; pp. 1771–1776.
16. Guo, Q.; Zhang, C.; Zhang, Y.; Liu, H. An efficient SVD-based method for image denoising. *IEEE Trans. Circuits Syst. Video Technol.* **2015**, *26*, 868–880. [CrossRef]
17. Chung, K.L.; Yang, W.N.; Huang, Y.H.; Wu, S.T.; Hsu, Y.C. On SVD-based watermarking algorithm. *Appl. Math. Comput.* **2007**, *188*, 54–57. [CrossRef]
18. Chang, C.C.; Tsai, P.; Lin, C.C. SVD-based digital image watermarking scheme. *Pattern Recognit. Lett.* **2005**, *26*, 1577–1586. [CrossRef]
19. Baranger, J.; Arnal, B.; Perren, F.; Baud, O.; Tanter, M.; Demené, C. Adaptive spatiotemporal SVD clutter filtering for ultrafast Doppler imaging using similarity of spatial singular vectors. *IEEE Trans. Med. Imaging* **2018**, *37*, 1574–1586. [CrossRef] [PubMed]

20. Tsyganova, J.V.; Kulikova, M.V. SVD-based Kalman filter derivative computation. *IEEE Trans. Autom. Control.* **2017**, *62*, 4869–4875. [CrossRef]
21. Kulikova, M.V. Hyperbolic SVD-based Kalman filtering for Chandrasekhar recursion. *IET Control Theory Appl.* **2019**, *13*, 1525–1531. [CrossRef]
22. Kulikova, M.V.; Tsyganova, J.V. Improved discrete-time Kalman filtering within singular value decomposition. *IET Control Theory Appl.* **2017**, *11*, 2412–2418. [CrossRef]
23. Liu, F.; Du, R.; Cheng, Y.; Sun, Z. LP-W-ℓ_∞-SVD Algorithm for Direction-of-Arrival Estimation. *IEEE Sens. J.* **2016**, *17*, 428–433. [CrossRef]
24. Cheng, Y.; Zhu, J.; Lin, X. An enhanced incremental SVD algorithm for change point detection in dynamic networks. *IEEE Access* **2018**, *6*, 75442–75451. [CrossRef]
25. Kanhe, A.; Aghila, G. A DCT–SVD-Based Speech Steganography in Voiced Frames. *Circuits Syst. Signal Process.* **2018**, *37*, 5049–5068. [CrossRef]
26. Hsu, Y.W.; Huang, S.S.; Perng, J.W. Application of multisensor fusion to develop a personal location and 3D mapping system. *Optik* **2018**, *172*, 328–339. [CrossRef]
27. Deng, T.B. Design of complex-coefficient variable digital filters using successive vector-array decomposition. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2005**, *52*, 932–942. [CrossRef]
28. Hogben, L. *Handbook of Linear Algebra*; Chapman & Hall/CRC: Boca Raton, FL, USA, 2006.
29. Golub, G.H.; Reinsch, C. Singular value decomposition and least squares solutions. *Numer. Math.* **1970**, *14*, 403–420. [CrossRef]
30. Demmel, J.; Kahan, W. Accurate Singular Values of Bidiagonal Matrices. *SIAM J. Sci. Stat. Comput.* **1990**, *11*, 873–912. [CrossRef]
31. Forsythe, G.E.; Henrici, P. The cyclic Jacobi method for computing the principal values of a complex matrix. *Trans. Am. Math. Soc.* **1960**, *94*, 1–23. [CrossRef]
32. Kaiser, H.F. The JK Method: A Procedure for Finding the Eigenvectors and Eigenvalues of a Real Symmetric Matrix. *Comput. J.* **1972**, *15*, 271–273. [CrossRef]
33. Gu, M.; Eisenstat, S.C. A Divide-and-Conquer Algorithm for the Bidiagonal SVD. *SIAM J. Matrix Anal. Appl.* **1995**, *16*, 79–92. [CrossRef]
34. Stewart, G.W. On the early history of the singular value decomposition. *SIAM Rev.* **1993**, *35*, 551–566. [CrossRef]
35. Cline, A.K.; Dhillon, I.S. Computation of the Singular Value Decomposition. In *Handbook of Linear Algebra*; Chapman & Hall/CRC: Boca Raton, FL, USA, 2006.
36. Wu, C.; Tsai, P. An SVD Processor Based on Golub–Reinsch Algorithm for MIMO Precoding With Adjustable Precision. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2019**, *66*, 2572–2583. [CrossRef]
37. Willink, T.J. Efficient adaptive SVD algorithm for MIMO applications. *IEEE Trans. Signal Process.* **2008**, *56*, 615–622. [CrossRef]
38. Zhang, A.; Xia, D. Tensor SVD: Statistical and computational limits. *IEEE Trans. Inf. Theory* **2018**, *64*, 7311–7338. [CrossRef]
39. Kaloorazi, M.F.; de Lamare, R.C. Subspace-Orbit randomized decomposition for low-rank matrix approximations. *IEEE Trans. Signal Process.* **2018**, *66*, 4409–4424. [CrossRef]
40. Yang, Y.; Rao, J. Robust and Efficient Harmonics Denoising in Large Dataset Based on Random SVD and Soft Thresholding. *IEEE Access* **2019**, *7*, 77607–77617. [CrossRef]
41. Fontenla-Romero, O.; Pérez-Sánchez, B.; Guijarro-Berdiñas, B. LANN-SVD: A non-iterative SVD-based learning algorithm for one-layer neural networks. *IEEE Trans. Neural Netw. Learn. Syst.* **2017**, *29*, 3900–3905. [PubMed]
42. Aharon, M.; Elad, M.; Bruckstein, A. K-SVD: An algorithm for designing overcomplete dictionaries for sparse representation. *IEEE Trans. Signal Process.* **2006**, *54*, 4311–4322. [CrossRef]
43. Kviatkovsky, I.; Gabel, M.; Rivlin, E.; Shimshoni, I. On the Equivalence of the LC-KSVD and the D-KSVD Algorithms. *IEEE Trans. Pattern Anal. Mach. Intell.* **2016**, *39*, 411–416. [CrossRef]
44. Eksioglu, E.M.; Bayir, O. K-SVD meets transform learning: Transform K-SVD. *IEEE Signal Process. Lett.* **2014**, *21*, 347–351. [CrossRef]
45. Dumitrescu, B.; Irofti, P. Regularized K-SVD. *IEEE Signal Process. Lett.* **2017**, *24*, 309–313. [CrossRef]
46. Raja, H.; Bajwa, W.U. Cloud K-SVD: A collaborative dictionary learning algorithm for big, distributed data. *IEEE Trans. Signal Process.* **2015**, *64*, 173–188. [CrossRef]
47. Lei, Y.; Fang, Y.; Zhang, L. A Weighted K-SVD-Based Double Sparse Representations Approach for Wireless Channels Using the Modified Takenaka-Malmquist Basis. *IEEE Access* **2018**, *6*, 54331–54342. [CrossRef]
48. Huang, K.J.; Chang, J.C.; Feng, C.W.; Fang, W.C. A parallel VLSI architecture of singular value decomposition processor for real-time multi-channel EEG system. In Proceedings of the 2013 IEEE International Symposium on Consumer Electronics (ISCE), Hsinchu, Taiwan, 3–6 June 2013; pp. 21–22.
49. Fang, W.; Chang, J.; Huang, K.; Feng, C.; Chou, C. An efficient VLSI implementation of SVD processor of on-line recursive ICA for real-time EEG system. In Proceedings of the 2014 IEEE Biomedical Circuits and Systems Conference (BioCAS) Proceedings, Lausanne, Switzerland, 22–24 October 2014; pp. 73–76.
50. Yang, C.H.; Chou, C.W.; Hsu, C.S.; Chen, C.E. A systolic array based GTD processor with a parallel algorithm. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2015**, *62*, 1099–1108. [CrossRef]
51. Hwang, Y.T.; Chen, W.D.; Hong, C.R. A low complexity geometric mean decomposition computing scheme and its high throughput VLSI implementation. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2013**, *61*, 1170–1182. [CrossRef]

52. Guenther, D.; Leupers, R.; Ascheid, G. A scalable, multimode SVD precoding ASIC based on the cyclic Jacobi method. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2016**, *63*, 1283–1294. [CrossRef]
53. Bravo, I.; Vázquez, C.; Gardel, A.; Lazaro, J.L.; Palomar, E. High level synthesis FPGA implementation of the Jacobi algorithm to solve the eigen problem. *Math. Probl. Eng.* **2015**, *2015*. [CrossRef]
54. Wang, Y.; Lee, J.; Ding, Y.; Li, P. A Scalable FPGA Engine for Parallel Acceleration of Singular Value Decomposition. In Proceedings of the 2020 21st International Symposium on Quality Electronic Design (ISQED), Santa Clara, CA, USA, 25–26 March 2020; pp. 370–376.
55. Tian, M.; Sima, M.; McGuire, M. Behavioral Implementation of SVD on FPGA. In Proceedings of the 2018 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT), Louisville, KY, USA, 6–8 December 2018; pp. 495–500.
56. Mansoori, M.A.; Casu, M.R. High Level Design of a Flexible PCA Hardware Accelerator Using a New Block-Streaming Method. *Electronics* **2020**, *9*, 449. [CrossRef]
57. Lahabar, S.; Narayanan, P. Singular value decomposition on GPU using CUDA. In Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing, Rome, Italy, 23–29 May 2009; pp. 1–10.
58. Athi, M.V.; Zekavat, S.R.; Struthers, A.A. Real-time signal processing of massive sensor arrays via a parallel fast converging svd algorithm: Latency, throughput, and resource analysis. *IEEE Sens. J.* **2016**, *16*, 2519–2526. [CrossRef]
59. Yang, W.; Liu, Z. Accelerating Householder bidiagonalization with ARM NEON technology. In Proceedings of the 2012 Asia Pacific Signal and Information Processing Association Annual Summit and Conference, Hollywood, CA, USA, 3–6 December 2012; pp. 1–4.
60. ARM. Cortex™-M4 Devices Generic User Guide. 2011. Available online: https://developer.arm.com/docs/dui0553/b (accessed on 2 November 2020).
61. Kalman, R.E. A New Approach to Linear Filtering and Prediction Problems. *J. Basic Eng.* **1960**, *82*, 35–45. [CrossRef]
62. Golub, G.H.; Van Loan, C.F. *Matrix Computations*; Johns Hopkins: Baltimore, MD, USA, 1983.
63. Wilkinson, J. Global convergene of tridiagonal QR algorithm with origin shifts. *Linear Algebra Appl.* **1968**, *1*, 409–420. [CrossRef]