Computing the SVD of a quaternion matrix

By Stephen J. Sangwine*
Department of Electronic Systems Engineering, University of Essex, UK
AND Nicolas Le Bihan
Laboratoire des Images et des Signaux, INPG Grenoble, France

Abstract

The practical and accurate computation of the singular value decomposition of a quaternion matrix is of importance in vector signal processing using quaternions. We present a Jacobi algorithm for computing such an SVD, and discuss its utility and accuracy. The algorithm is included in an open-source Matlab toolbox for quaternions where it serves as an accurate reference implementation.

1. Introduction

The singular value decomposition (SVD) is usually represented as $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^\dagger$ where $\mathbf{A}$ is an arbitrary matrix (not necessarily square), $\mathbf{U}$ and $\mathbf{V}$ are unitary matrices ($\mathbf{U}^\dagger\mathbf{U} = \mathbf{I}$), Σ is a diagonal matrix of real singular values, and $\dagger$ denotes a conjugate transpose (using a *quaternion* conjugate in the case of a quaternion matrix, and a complex conjugate in the case of a complex matrix). The singular values are unique, but the columns of $\mathbf{U}$ and $\mathbf{V}$ (the singular vectors) are not. It is usual when computing the decomposition to sort the singular values (and the associated singular vectors) into order so that the largest singular values are stored in the first few rows of Σ.

$\mathbf{A}$ can be reconstructed using all the singular values, or by using a subset (usually from among the largest singular values):

$$\mathbf{A} = \mathbf{U}\Sigma_1\mathbf{V}^\dagger + \mathbf{U}\Sigma_2\mathbf{V}^\dagger + \ldots \mathbf{U}\Sigma_N\mathbf{V}^\dagger$$

where Σ_i denotes a diagonal matrix with only one non-zero element in the i^{th} diagonal position (a singular value).

2. Motivation for computing the quaternion SVD

At a simple level, the SVD can be used to extract components of a matrix with significant singular values. This has been exploited generally in what is known as *reduced rank signal processing* (Scharf 1991) where the idea is to extract, in some sense, the significant parts of a signal. More generally, the SVD is a tool that can be used as part of a signal processing algorithm as for example in (Le Bihan and Mars 2004), where the SVD is used to extract significant signals from noise with particular application to seismic signals captured by vector geophones, or in (Miron et al. 2006) where the SVD of a quaternion matrix is used as part of a quaternion version of the MUSIC algorithm for direction of arrival estimation using a vector sensor array.

Computing the SVD using a Jacobi algorithm promises more accurate results than any other known algorithm, and therefore a quaternion Jacobi algorithm is useful as a reference implemen-

* The work presented in this paper was supported by the UK Engineering and Physical Sciences Research Council (Grant number GR/S16881/01); and by the Centre National de la Recherche Scientifique and the Royal Academy of Engineering, UK.

tation. The full details of the algorithm are given in Le Bihan and Sangwine (2006) which was based on a compiled code implementation as well as an early coding in more or less raw Matlab code. In this paper we present a simplified explanation of the algorithm, and discuss an open source Matlab implementation which we have released, to coincide with this paper, as part of a quaternion toolbox for Matlab which we developed and published last year (Sangwine and Le Bihan 2005).

We have also implemented and published a much faster method for computing the quaternion SVD based on bidiagonalisation to a real or complex bidiagonal matrix using Householder transformations. This algorithm is less accurate than the Jacobi algorithm (Sangwine and Le Bihan 2006) but its speed is a significant advantage for larger matrices and therefore it is this algorithm that is used as the default quaternion SVD algorithm in our toolbox.

3. Review of the Jacobi algorithm

The Jacobi algorithm is a classical iterative algorithm for computing the SVD (Golub and van Loan 1996). It computes the singular values more accurately than any other known algorithm.

Given a matrix $\mathbf{A}$ the algorithm works by *diagonalising* an implicit Hermitian matrix $\mathbf{B} = \mathbf{A}^{\dagger}\mathbf{A}$. The algorithm nulls one pair of off-diagonal elements of $\mathbf{B}$ at a time, and it can therefore be described in terms of operations on 2×2 matrices composed from elements of $\mathbf{B}$ taken from the i^{th} and j^{th} rows and columns:

$$\mathbf{b} = \begin{bmatrix} b_{ii} & b_{ij} \\ b_{ji} & b_{jj} \end{bmatrix}$$

Here the two off-diagonal elements are quaternion conjugates, and the on-diagonal elements are real (because $\mathbf{B}$ is Hermitian). The off-diagonal elements are nulled by unitary transformations known as Jacobi rotations, which take the following form:

$$\Omega = \begin{bmatrix} c & -s \\ \overline{s} & c \end{bmatrix}$$

where the overbar denotes a quaternion conjugate. The product $\Omega^{\dagger}\mathbf{b}\Omega = \mathbf{r}$ where $\mathbf{r}$ is real and takes the form $\begin{bmatrix} r_{ii} & 0 \\ 0 & r_{jj} \end{bmatrix}$. c and s are computed using essentially the same algorithm as in the complex case described in Forsythe and Henrici (1960) (this is because the implicit matrix $\mathbf{B}$ is Hermitian and therefore the 2×2 matrices $\mathbf{b}$ are isomorphic to a complex Hermitian matrix):

$$\tau = \frac{(b_{jj} - b_{ii})}{2|b_{ij}|}, \qquad t = \frac{\operatorname{sign} \tau}{|\tau| + \sqrt{1 + \tau^2}}, \qquad c = \frac{1}{\sqrt{1 + t^2}}, \qquad s = b_{ij}\frac{tc}{|b_{ij}|}$$

The full details are given in Le Bihan and Sangwine (2006).

Diagonalization of $\mathbf{B}$ is achieved by iteratively processing upper diagonal elements, and for each one computing a Jacobi rotation matrix Ω. This matrix is used to modify two columns of $\mathbf{A}$, which is equivalent to nulling the off-diagonal elements of $\mathbf{B}$ used to compute the rotation matrix Ω. $\mathbf{A}$ converges towards $\mathbf{U}\Sigma$. $\mathbf{V}$ is accumulated from the product of the Jacobi rotation matrices. The iterative process is stopped when the implicit matrix $\mathbf{B}$ is sufficiently close to diagonal. In practice this requires a little less than 10 iterations over the upper diagonal part of $\mathbf{B}$.

There are various schemes for iterating over the upper triangular part of $\mathbf{B}$. The simplest to implement is called cyclic sweep. All the elements of the upper triangular part are processed in turn, and at the end of each pass over all such elements, the off-diagonal sum is recomputed, and the algorithm is stopped if this is small enough.

4. Implementation of the quaternion Jacobi SVD algorithm

Our most recent implementation of the Jacobi SVD algorithm is based on an open-source Matlab toolbox for quaternions which we developed in 2005 (Sangwine and Le Bihan 2005) which overloads a large number of fundamental Matlab operations and functions for quaternion matrices including the colon operator and similar for indexing, as well as arithmetic operators and matrix functions such as `sum` and `mean`. The toolbox defines a *class* for quaternions (in fact quaternion matrices, since all objects in Matlab are matrices and quaternions are no exception). The internal representation of a quaternion is, not surprisingly, a structure with four components, one for each of the **W**, **X**, **Y** and **Z** components of the quaternion (matrix). Normally, each of these matrices is of type **double**. Quaternions with complex components are handled quite naturally by the toolbox, since Matlab permits double matrices to take complex values. The toolbox overloads many Matlab functions for quaternion matrices, but most importantly, it also overloads the colon operator and the various forms of brackets which permit indexing of matrices. This means that normal Matlab notation can be used with quaternion matrices. In this way, we can now write Matlab code which will work unchanged on double, complex and quaternion matrices, and indeed, our Jacobi SVD implementation is able to compute the SVD of real, complex and quaternion matrices using *the same code*. In order to make this possible we have used one or two 'tricks', one of which will be explained here to show that the coding is not difficult. In our implementation of the Jacobi SVD algorithm, the output matrix **V** must be initialised to an identity matrix since the rotations will be accumulated by multiplication. In order for the code to work with real, complex and quaternion matrices, **V** must be initialized to a double or quaternion identity matrix as appropriate. The following two lines of code achieve this:

```
F = str2func(class(A)); % F is a function handle.
V = F(eye(N));
```

The first line is standard Matlab code, and it creates a function handle with the name of the data type of the input matrix **A**. This function handle will be either **double** or **quaternion**. (If **A** is complex, `F` will still be **double**.) The second line, in effect, now reads either `V = double(eye(N));` or `V = quaternion(eye(N));` depending on the type of **A**. In the first case, since the `eye` function returns a double matrix, calling `double` has no effect, and **V** is initialised to be a double identity matrix, as required. In the second case, the `eye` function creates a double identity matrix of size N, and the quaternion constructor function converts this to a quaternion matrix (the imaginary parts are all zero of course). This is because the quaternion constructor function has been defined so that, if given a single argument which is not a quaternion matrix, it will create a quaternion matrix, and supply zero matrices for the three imaginary components. Therefore, in this second case, **V** is initialised to be a quaternion matrix, with the scalar (**W**) part an identity matrix, and the vector parts, zero matrices.

We believe this ability to write code that will work for all three cases can be exploited in signal processing algorithms, where the ability to generalize from complex code could be a useful test feature.

5. Some numerical results and comparisons

We present here some numerical results obtained using our Matlab implementation on random matrices. We compare the Jacobi algorithm on real and complex matrices against the standard Matlab implementation of the SVD (which uses LAPACK), and we compare the Jacobi algorithm on a quaternion matrix against the default quaternion algorithm in our toolbox, which is based on

	Real Maximum	Real Mean	Complex Maximum	Complex Mean	Quaternion Maximum	Quaternion Mean
Matlab SVD	2.3×10^{-14}	3.9×10^{-15}	1.5×10^{-14}	4.2×10^{-15}		
QTFM SVD					2.2×10^{-14}	7.4×10^{-15}
Jacobi SVD	8.0×10^{-15}	3.8×10^{-15}	1.1×10^{-14}	4.2×10^{-15}	1.2×10^{-14}	5.6×10^{-15}

TABLE 1. Numerical results. (Each entry gives the maximum and mean errors over 50 tests.)

diagonalization to a real matrix, and the Matlab SVD of the real matrix (Sangwine and Le Bihan 2006).

The random matrices tested were created with the Matlab `randn` function. In the case of complex matrices, this function is used to construct the real and imaginary matrices, and the case of quaternion matrices, it is used to construct all four quaternion components.

Table 1 shows the errors between the original matrix and a reconstruction computed from the singular value decomposition computed as follows, where $\mathbf{A}$ is the original random matrix and $\mathbf{B} = \mathbf{U}\Sigma\mathbf{V}^{\dagger}$ is computed from the results of the decomposition: $\mathbf{E} = \mathrm{abs}(\mathbf{A} - \mathbf{B})$. For each test matrix, the element of $\mathbf{E}$ with the largest modulus is found, and the results in the table are the maximum and mean values found across 50 tests. The test matrices were 8×8.

6. Conclusions

The direct implementation of the Jacobi algorithm for the SVD has been a useful step to take because it provides an accurate reference implementation for verifying other algorithms.

Although the Jacobi SVD algorithm itself is not difficult to code, its implementation was made a lot easier by the existence of our quaternion toolbox which now eliminates the need to consider many low-level programming issues and permits the development of quaternion algorithms using the same high-level abstractions as can be employed when coding real and complex algorithms.

REFERENCES

G. E. Forsythe and P. Henrici. The cyclic Jacobi method for computing the principal values of a complex matrix. *Transactions of the American Mathematical Society*, 94(1):1–23, January 1960.

Gene H. Golub and Charles F. van Loan. *Matrix Computations*. Johns Hopkins studies in the Mathematical Sciences. The Johns Hopkins University Press, Baltimore and London, third edition, 1996. ISBN 0-8018-5413-X and 0-8018-5414-8 (pbk.).

N. Le Bihan and J. Mars. Singular value decomposition of quaternion matrices: A new tool for vector-sensor signal processing. *Signal Processing*, 84(7):1177–1199, 2004.

N. Le Bihan and S. J. Sangwine. Jacobi method for quaternion matrix singular value decomposition. *Applied Mathematics and Computation*, 2006. doi: 10.1016/j.amc.2006.09.055. Available online 20 October.

S. Miron, N. Le Bihan, and J. I. Mars. Quaternion-MUSIC for vector-sensor array processing. *IEEE Transactions on Signal Processing*, 54(4):1218–1229, April 2006.

S. J. Sangwine and N. Le Bihan. Quaternion singular value decomposition based on bidiagonalization to a real or complex matrix using quaternion householder transformations. *Applied Mathematics and Computation*, 182(1):727–738, 1 November 2006. doi: 10.1016/j.amc.2006.04.032.

S. J. Sangwine and N. Le Bihan. Quaternion Toolbox for Matlab®, 2005. URL `http://qtfm.sourceforge.net/`. Software library, licensed under the GNU General Public License.

Louis L. Scharf. The SVD and reduced rank signal processing. *Signal Processing*, 25(2):113–133, November 1991. doi: 10.1016/0165-1684(91)90058-Q.