

UCLA

Department of Statistics Papers

Title

Using Jacobi Plane Rotations in R

Permalink

<https://escholarship.org/uc/item/5zh6d1cg>

Author

Leeuw, Jan de

Publication Date

2009-10-06

Peer reviewed

USING JACOBI PLANE ROTATIONS IN R

JAN DE LEEUW

ABSTRACT. Matrix techniques for various types of diagonalizations of matrices and three-dimensional arrays are implemented in R using Jacobi plane rotations.

1. INTRODUCTION

Many problems in multivariate analysis can be formulated as optimizing a function $f(K)$ over the rotation matrices of order n , i.e. the square matrices that satisfy $K'K = KK' = I$. Or, more generally, to optimize a function $f(K_1, \dots, K_n)$ over a number of rotation matrices.

Recently, much interesting theory has been developed on how to design gradient type optimization methods for such problems, using the differential geometry of Grassman and Stiefel manifolds [Edelman et al., 1998]. (Edelman, Book – gradient methods).

In this paper we go a different route, however. We use the rotation matrix version of one-dimensional coordinate-wise optimization, by building up the optimal rotation matrices iteratively from a sequence of one-parameter plane rotations.

2. PLANE ROTATIONS

Two-by-two rotation matrices can be written in the one-parameter form

$$(1) \quad K(\theta) = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix}.$$

Note that $K(\theta)' = K(-\theta)$ and $K(0) = I$.

Jacobi plane rotations of order n are the matrices $K_{ij}(\theta)$, which are equal to the identity matrix of order n , but with elements $(i, j), (j, j), (i, j)$ and (j, i) replaced

by the four elements of $K(\theta)$. We use s_θ and c_θ as abbreviations for $\sin(\theta)$ and $\cos(\theta)$. Thus

$$(2a) \quad \{K_{ij}(\theta)\}_{ii} = c_\theta,$$

$$(2b) \quad \{K_{ij}(\theta)\}_{ij} = s_\theta,$$

$$(2c) \quad \{K_{ij}(\theta)\}_{ji} = -s_\theta,$$

$$(2d) \quad \{K_{ij}(\theta)\}_{jj} = c_\theta.$$

Now $K_{ij}(\theta)' = K_{ji}(\theta) = K_{ij}(-\theta)$, and again $K_{ij}(0) = I$.

Suppose X is a rectangular matrix with n rows and m columns. Then $\tilde{X} = K_{ij}(\theta)X$ differs from X only in row i and row j . We have

$$(3a) \quad \tilde{x}_{\alpha\beta} = \begin{cases} c_\theta x_{i\beta} + s_\theta x_{j\beta} & \text{if } \alpha = i, \\ -s_\theta x_{i\beta} + c_\theta x_{j\beta} & \text{if } \alpha = j, \\ x_{\alpha\beta} & \text{otherwise.} \end{cases}$$

Similarly $\tilde{X} = XK_{kl}(\xi)$ differs from X only in columns k and l , with

$$(3b) \quad \tilde{x}_{\alpha\beta} = \begin{cases} c_\xi x_{\alpha j} - s_\xi x_{\alpha l} & \text{if } \beta = j, \\ s_\xi x_{\alpha j} + c_\xi x_{\alpha l} & \text{if } \beta = l, \\ x_{\alpha\beta} & \text{otherwise.} \end{cases}$$

Equations (3a) and (3b) are one-sided left and right Jacobi transformations of X .

The two-sided transform is defined by $\tilde{X} = K_{ij}(\theta)XK_{kl}(\xi)$. We find

$$(4) \quad \tilde{x}_{\alpha\beta} = \begin{cases} c_\xi x_{\alpha k} - s_\xi x_{\alpha l} & \text{if } \beta = k \text{ and } \alpha \neq i, j, \\ s_\xi x_{\alpha k} + c_\xi x_{\alpha l} & \text{if } \beta = l \text{ and } \alpha \neq i, j, \\ c_\theta x_{i\beta} + s_\theta x_{j\beta} & \text{if } \alpha = i \text{ and } \beta \neq k, l, \\ -s_\theta x_{i\beta} + c_\theta x_{j\beta} & \text{if } \alpha = j \text{ and } \beta \neq k, l, \\ c_\theta c_\xi x_{ik} + s_\theta c_\xi x_{jk} - c_\theta s_\xi x_{il} - s_\theta s_\xi x_{jl} & \text{if } \alpha = i \text{ and } \beta = k, \\ c_\theta s_\xi x_{ik} + s_\theta s_\xi x_{jk} + c_\theta c_\xi x_{il} + s_\theta c_\xi x_{jl} & \text{if } \alpha = i \text{ and } \beta = l, \\ -s_\theta c_\xi x_{ik} + c_\theta c_\xi x_{jk} + s_\theta s_\xi x_{il} - c_\theta s_\xi x_{jl} & \text{if } \alpha = j \text{ and } \beta = k, \\ -s_\theta s_\xi x_{ik} + c_\theta s_\xi x_{jk} - s_\theta c_\xi x_{il} + c_\theta c_\xi x_{jl} & \text{if } \alpha = j \text{ and } \beta = l, \\ x_{\alpha\beta} & \text{otherwise.} \end{cases}$$

Of course the one-sided transforms can be recovered by setting either $\theta = 0$ or $\xi = 0$.

3. CYCLES

In our algorithms we update the current rotation matrix K of order n by applying $\frac{1}{2}n(n-1)$ plane rotations $K_{ij}(\theta)$ for all $i < j$, where θ is chosen to optimize some criterion. In all our applications the criterion is a sum of squares of certain matrix or array elements.

Usually we update from the left, i.e. we set $\tilde{K} = K_{ij}(\hat{\theta})K$, with the optimally chosen $\hat{\theta}$. If the function we are optimizing contains a term KX , for some fixed matrix X , then of course $\tilde{X} = K_{ij}(\hat{\theta})KX$. It is usually convenient to update X *in situ*, which can be done by just a few multiplications and additions. Because the criterion is a sum of squares, the optimal $\sin(\hat{\theta})$ and $\cos(\hat{\theta})$ can usually be found by solving a 2×2 eigenvalue problem for the eigenvector corresponding with the smallest or largest eigenvalue.

In the symmetric matrix case we use two-sided updates. Because of symmetry we still only deal with a one parameter problem, and trigonometric identities can be used to again reduce finding the optimal plane rotation to solving a 2×2 eigenvalue problem. Each cycle of the algorithm again passes through all $\frac{1}{2}n(n-1)$ in order.

In the literature there are many variations on how to cycle through the plane rotations. We could choose the largest element contributing to the sum of squares, or we only decide to rotate if an element is above a certain threshold (and then lower the threshold in further cycles). This may be important for really large examples, but our `R` code is intended for fairly small ones. In future versions of our work we will optimize by code by translating the loops in the core of the algorithm to `C`.

4. APPLICATIONS

4.1. **Eigenvalues.** In the classical eigenvalue problem we have a symmetric matrix A , and we want to find a rotation matrix K such that $K'AK$ is diagonal. In the classical Jacobi method we build up K by successive plane rotations. Thus the matrix X is symmetric, and we use the symmetric two-sided transform

$$\tilde{A} = K_{ij}(\xi)'AK_{ij}(\xi)$$

In the formulas from the previous section we have $\theta = -\xi$. Using this, and the symmetry of A , we find

$$(5) \quad \tilde{a}_{\alpha\beta} = \begin{cases} c_\xi a_{\alpha i} - s_\xi a_{\alpha j} & \text{if } \beta = i \text{ and } \alpha \neq i, j, \\ s_\xi a_{\alpha i} + c_\xi a_{\alpha j} & \text{if } \beta = j \text{ and } \alpha \neq i, j, \\ c_\xi a_{i\beta} - s_\xi a_{j\beta} & \text{if } \alpha = i \text{ and } \beta \neq i, j, \\ s_\xi a_{i\beta} + c_\xi a_{j\beta} & \text{if } \alpha = j \text{ and } \beta \neq i, j, \\ c_\xi^2 a_{ii} - 2s_\xi c_\xi a_{ij} + s_\xi^2 a_{jj} & \text{if } \alpha = i \text{ and } \beta = i, \\ s_\xi c_\xi (a_{ii} - a_{jj}) + (c_\xi^2 - s_\xi^2) a_{ij} & \text{if } \alpha = i \text{ and } \beta = j, \\ s_\xi c_\xi (a_{ii} - a_{jj}) + (c_\xi^2 - s_\xi^2) a_{ij} & \text{if } \alpha = j \text{ and } \beta = i, \\ s_\xi^2 a_{ii} + 2s_\xi c_\xi a_{ij} + c_\xi^2 a_{jj} & \text{if } \alpha = j \text{ and } \beta = j, \\ a_{\alpha\beta} & \text{otherwise.} \end{cases}$$

Now

$$\begin{aligned} (c_\xi a_{\alpha i} - s_\xi a_{\alpha j})^2 + (s_\xi a_{\alpha i} + c_\xi a_{\alpha j})^2 &= a_{\alpha i}^2 + a_{\alpha j}^2, \\ (c_\xi a_{i\beta} - s_\xi a_{j\beta})^2 + (s_\xi a_{i\beta} + c_\xi a_{j\beta})^2 &= a_{i\beta}^2 + a_{j\beta}^2, \end{aligned}$$

and thus

$$\sum_{\alpha < \beta} \tilde{a}_{\alpha\beta}^2 = \sum_{\alpha < \beta} a_{\alpha\beta}^2 - a_{ij}^2 + \tilde{a}_{ij}^2.$$

We minimize the sum of squares of the off-diagonal elements of \tilde{A} by solving

$$\tilde{a}_{ij} = s_\xi c_\xi (a_{ii} - a_{jj}) + (c_\xi^2 - s_\xi^2) a_{ij} = \sin(2\xi) d_{ij} + \cos(2\xi) a_{ij} = 0,$$

with $d_{ij} = \frac{1}{2}(a_{ii} - a_{jj})$. One solution for the vector $(\sin(2\xi), \cos(2\xi))$ is

$$u = \frac{1}{\sqrt{a_{ij}^2 + d_{ij}^2}} \begin{bmatrix} a_{ij} \\ -d_{ij} \end{bmatrix}.$$

We now solve for any s_ξ and c_ξ such that $2s_\xi c_\xi = u_1$, $c_\xi^2 - s_\xi^2 = u_2$, and $c_\xi^2 + s_\xi^2 = 1$.

Thus

$$\begin{aligned} c_\xi &= \sqrt{\frac{1+u_2}{2}}, \\ s_\xi &= \mathbf{sign}(u_1) \sqrt{\frac{1-u_2}{2}}. \end{aligned}$$

Note that we can also update the matrix of eigenvectors K by starting with $K = I$, and by updating to \tilde{K} after each plane rotation using

$$(6) \quad \tilde{k}_{\alpha\beta} = \begin{cases} c_{\xi}k_{\alpha i} - s_{\xi}k_{\alpha j} & \text{if } \beta = i, \\ s_{\xi}k_{\alpha i} + c_{\xi}k_{\alpha j} & \text{if } \beta = j, \\ k_{\alpha\beta} & \text{otherwise.} \end{cases}$$

The code for the algorithm is given in the Appendix. Note that we have implemented the cyclic Jacobi method, without any searching for a largest pivot element. In applying our algorithm to various matrices, we do see the fast quadratic convergence. In this form, however, our method is not intended to be competitive with the `eigen()` routine in R, which is optimized compiled FORTRAN code from LAPACK.

The comparison with `eigen()` will become more interesting by rewriting critical sections in C, although it is well-established that Jacobi is slower than the combination of Givens-Householder tri-diagonalization and inverse iteration. But Jacobi has the advantage that it is more easily parallelized [Sameh, 1971; Pourzandi and Tourancheau, 1995], and we intend to use OpenMP to see if we can make these routines competitive on SMP machines.

4.2. Singular Values. The existence theorem for the singular value decomposition says that for any rectangular X there exist rotation matrices K and L such that $\tilde{X} = KXL$ is diagonal. We can compute the singular value decomposition by one-sided Jacobi rotations, minimizing the sum of squares of the off-diagonal elements. We can first pivot through all left planar rotations, then pivot through all one-sided right planar rotations, then do the left ones again, and so on. Note that this algorithm can also be applied to symmetric matrices, in which case it gives us an alternative method to compute eigenvalues and eigenvectors.

Instead of minimizing the sum of squares of the off-diagonal elements we may as well maximize the sum of squares of the diagonal elements. For a left Jacobi transformation $K_{ij}(\theta)$ this means maximizing

$$(\cos(\theta)x_{ii} + \sin(\theta)x_{ji})^2 + (-\sin(\theta)x_{ij} + \cos(\theta)x_{jj})^2.$$

Define a 2×2 matrix V with

$$\begin{aligned} v_{11} &= x_{ij}^2 + x_{ji}^2, \\ v_{12} &= v_{21} = x_{ii}x_{ji} - x_{jj}x_{ij}, \\ v_{22} &= x_{ii}^2 + x_{jj}^2, \end{aligned}$$

and a two element vector u with $u_1 = \sin(\theta)$ and $u_2 = \cos(\theta)$. Then we must maximize $u'Vu$ over $u'u = 1$. Thus \hat{u} is the normalized eigenvector corresponding with the largest eigenvalue of V . There is no need to actually compute the optimal θ because u has all the information we need.

For a right Jacobi rotation $K_{ij}(\theta)$ we minimize

$$(\cos(\theta)x_{ii} - \sin(\theta)x_{ij})^2 + (\sin(\theta)x_{ji} + \cos(\theta)x_{jj})^2,$$

and we compute the largest eigenvalue and corresponding eigenvector of

$$\begin{aligned} v_{11} &= x_{ij}^2 + x_{ji}^2, \\ v_{12} &= v_{21} = x_{jj}x_{ji} - x_{ii}x_{ij}, \\ v_{22} &= x_{ii}^2 + x_{jj}^2. \end{aligned}$$

The code for the algorithm in the Appendix, in the function `jSVD()`. There are various ways to deal with the fact that X is not square, and that consequently some of the singular vectors correspond with zero singular values. An obvious, although somewhat wasteful, way is to make X square by appending rows or columns with zeroes.

In our numerical experiments with `jSVD()` we find that the algorithm exhibits slow linear or even sublinear convergence. It performs reliably, but it is dreadfully slow. We are far better off applying the quadratically convergent Jacobi eigenvalue algorithm to $X'X$ or XX' , or even to

$$\begin{bmatrix} I & X \\ X' & I \end{bmatrix}.$$

On the other hand the algorithm can be generalized very simply to approximate simultaneous diagonalization of a number of rectangular matrices (see 4.4 below).

4.3. Simultaneous Diagonalization. If there are m symmetric matrices A_v they generally cannot be simultaneously diagonalized by an orthogonal transformation K . We can find K such that $K'A_vK$ is diagonal for all v if and only if the A_v commute in pairs, i.e. if and only if $A_vA_\mu = A_\muA_v$ for all v, μ . We can find K , however, such that the transformed $K'A_vK$ are as diagonal as possible in the least squares sense. For this we minimize the sum of squares of all off-diagonal elements.

As Subsection 4.1 shows, for $K_{ij}(\xi)$ we must choose the plane rotation angle ξ such that

$$\sum_{v=1}^m \left[\frac{1}{2} \sin(2\xi)(a_{iiv} - a_{jjv}) + \cos(2\xi)a_{ijv} \right]^2$$

is minimized. Let $d_{ijv} = \frac{1}{2}(a_{iiv} - a_{jjv})$. Define a 2×2 matrix V with

$$\begin{aligned} v_{11} &= \sum_{v=1}^m d_{ijv}^2, \\ v_{12} = v_{21} &= \sum_{v=1}^m d_{ijv}a_{ijv}, \\ v_{22} &= \sum_{v=1}^m a_{ijv}^2, \end{aligned}$$

and a two element vector u with $u_1 = \sin(2\xi)$ and $u_2 = \cos(2\xi)$. Then we must minimize $u'Vu$ over $u'u = 1$, and thus the optimal u is any eigenvector corresponding with the smallest eigenvalue of V . Again, as in the Jacobi eigenvector method,

$$\begin{aligned} c_\xi &= \sqrt{\frac{1+u_2}{2}}, \\ s_\xi &= \mathbf{sign}(u_1) \sqrt{\frac{1-u_2}{2}}. \end{aligned}$$

This solution was first derived by De Leeuw and Pruzansky [1978], although in an unnecessarily complicated way. It was implemented in a convenient FORTRAN routine by Clarkson [1988]. Ten Berge [1984] has shown, in an interesting paper, that Kaiser's VARIMAX rotation method [Kaiser, 1958] can be formulated as a simultaneous diagonalization problem, and that the algorithm proposed by Kaiser is the same as the one in De Leeuw and Pruzansky [1978]. The implementation of the function `jSimDiag()` in the Appendix converges quite rapidly.

4.4. Tucker Models. Suppose X_1, \dots, X_m are rectangular matrices of the same dimensions. The problem is to find K and L such that $\tilde{X}_j = KX_jL$ are as diagonal as

possible in the least squares sense. But for this we can straightforwardly use all the results from 4.2. This gives an orthogonal version of the TUCKER-2 model [Kroonenberg and De Leeuw, 1980]. The function `jSimSVD()` is in the Appendix.

A more general problem is to transform the three-dimensional array $X = \{x_{ijk}\}$, using three rotation matrices K, L and M , by

$$\tilde{x}_{abc} = \sum_p \sum_q \sum_r k_{ap} l_{bq} m_{cr} x_{pqr}.$$

We can define different criteria to optimize. An interesting one is to maximize the sum of squares of the body diagonal, i.e. of the elements for which $a = b = c$. This fits an orthonormal version of the INDSCAL/PARAFAC model. It is implemented in `jTucker3Diag()`. Another option is to maximize the sum of squares of the leading principal block with $a \leq A, b \leq B$ and $c \leq C$. This equivalent to fitting the original TUCKER-3 model [Kroonenberg and De Leeuw, 1980]. The function `jTucker3Block()` is also given in the Appendix. Both approaches are conceptually straightforward generalizations of Principal Component Analysis and the Singular Value Decomposition.

All techniques discussed in this section can be generalized to arrays in more than three dimensions. In \mathbb{R} this is most easily done by using the machinery developed by (APL). The code will be presented in a subsequent publication.

4.5. PREHOM. In De Leeuw [1982]; Bekker [1982] and in Bekker and De Leeuw [1988] a variation of multiple correspondence analysis using Jacobi plane rotations is proposed. The matrix to be analyzed is the Burt matrix [Burt, 1950] of m categorical variables. Variable j has k_j categories. If $G = \begin{bmatrix} G_1 & \dots & G_m \end{bmatrix}$ are the concatenated indicator matrices (dummies) of the variables, with G_j of dimension $n \times k_j$, then the Burt matrix is $C = G'G$. Matrix C has submatrices $C_{k\ell}$ of dimension $k_j \times k_\ell$. If $j \neq \ell$ then $C_{j\ell}$ contains the bivariate marginals (cross table) of variables j and ℓ . If $j = \ell$ then $C_{j\ell}$ is a diagonal matrix with the univariate marginals of variable j on the diagonal. Also define $D = \mathbf{diag}(C)$, and the scaled Burt matrix $A = D^{-\frac{1}{2}}CD^{-\frac{1}{2}}$.

Multiple correspondence analysis can be defined as diagonalization of the matrix A . Thus we could use the classical Jacobi method of Subsection 4.1. In De Leeuw [1982] a three-step approximate diagonalization of the scaled Burt matrix is proposed, which is more revealing from a data analysis point of view. In the first step

Jacobi plane rotations are used to approximately diagonalize all of the $k_j \times k_\ell$ submatrices $A_{j\ell}$ simultaneously. Note that the diagonal submatrices A_{jj} are equal to the identity matrix of order k_j , and are thus already diagonal. After the Jacobi step has finished we permute rows and columns to collect the diagonal elements of the $A_{j\ell}$ into a direct sum of diagonal blocks. If all k_j are equal to k , then there are k blocks of order m . Each diagonal block is a correlation matrix. The first block corresponds to the $(1, 1)$ elements of all $A_{j\ell}$, the second block to the $(2, 2)$ elements, and so on. If some variables have fewer categories, they will not occur in the later blocks. The third step of the approximate diagonalization computes the eigenvectors of the diagonal blocks, and uses them to diagonalize these blocks.

Thus the eigenvectors of the scaled Burt matrix are approximated by the product KPL , where K is the cumulative product of the Jacobi rotations that approximately diagonalize the $A_{j\ell}$, P is the permutation matrix that permutes the elements to approximate block-diagonal form, and L is the direct sum of the eigenvectors that diagonalize the correlation matrices along the diagonal. For m variables with k categories each we use $\frac{1}{2}mk(k + m - 2)$ parameters to do the approximate diagonalization, instead of the $\frac{1}{2}mk(mk - 1)$ parameters for the full diagonalization.

The code for the R implementation is the function `jMCA()` in the Appendix.

REFERENCES

- P. Bekker. Varianten van Niet-lineaire Principale Componenten Analyse. Master's thesis, Department of Psychology, University of Leiden, 1982.
- P. Bekker and J. De Leeuw. Relation between Variants of Nonlinear Principal Component Analysis. In J.L.A. Van Rijkevorsel and J. De Leeuw, editors, *Component and Correspondence Analysis*. Wiley, Chichester, England, 1988.
- C. Burt. The Factorial Analysis of Qualitative Data. *British Journal of Statistical Psychology*, 3:166–185, 1950.
- D.B. Clarkson. Remark AS R74: A Least Squares Version of Algorithm AS 211: The F-G Diagonalization Algorithm. *Applied Statistics*, 37:317–321, 1988.
- J. De Leeuw. Nonlinear Principal Component Analysis. In H. Caussinus et al., editor, *COMPSTAT 1982*, pages 77–86, Vienna, Austria, 1982. Physika Verlag.
- J. De Leeuw and S. Pruzansky. A New Computational Method to fit the Weighted Euclidean Distance Model. *Psychometrika*, 43:479–490, 1978.

- A. Edelman, T.A. Arias, and S.T. Smith. The Geometry of Algorithms with Orthogonality Constraints. *SIAM Journal of Matrix Analysis and Applications*, 20: 303–353, 1998.
- H.F. Kaiser. The Varimax Criterion of Analytic Rotation in Factor Analysis. *Psychometrika*, 23:187–200, 1958.
- P.M. Kroonenberg and J. De Leeuw. Principal Component Analysis of Three-Mode Data by Means of Alternating Least Squares Algorithms. *Psychometrika*, 45:69–97, 1980.
- M. Pourzandi and B. Tourancheau. Parallel Performance of Jacobi Eigenvalue Solution. *Computing Systems in Engineering*, 6:377–383, 1995.
- A.H. Sameh. On Jacobi and Jacobi-Like Algorithms for a Parallel Computer. *Mathematics of Computation*, 25(579–590), 1971.
- J.M.F. Ten Berge. A Joint Treatment of Varimax Rotation and the Problem of Diagonalizing Symmetric Matrices Simultaneously in the Least Squares Sense. *Psychometrika*, 49(347–358), 1984.

APPENDIX A. CODE

```

1 #
2 # jacobi package
3 # Copyright (C) 2008 Jan de Leeuw <deleeuw@stat.ucla.edu>
4 # UCLA Department of Statistics, Box 951554, Los Angeles, CA 90095-1554
5 #
6 # This program is free software; you can redistribute it and/or modify
7 # it under the terms of the GNU General Public License as published by
8 # the Free Software Foundation; either version 2 of the License, or
9 # (at your option) any later version.
10 #
11 # This program is distributed in the hope that it will be useful,
12 # but WITHOUT ANY WARRANTY; without even the implied warranty of
13 # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
14 # GNU General Public License for more details.
15 #
16 # You should have received a copy of the GNU General Public License
17 # along with this program; if not, write to the Free Software
18 # Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
19 #
20 #####
21 #
22 # version 0.0.1, 2008-12-05 Eigen and SVD
23 # version 0.1.0, 2008-12-06 Simultaneous Diagonalization
24 # version 0.2.0, 2008-12-08 Simultaneous SVD
25 # version 0.2.1, 2008-12-08 Various Bugfixes
26 # version 0.2.2, 2008-12-08 Small efficiency gains
27 # version 0.3.0, 2008-12-10 PREHOM in jMCA added
28 # version 1.0.0, 2008-12-10 Tucker3 Methods added
29 #
30
31 jEigen<-function(a, eps1=1e-6, eps2=1e-10, itmax=100, vectors=TRUE, verbose=FALSE) {
32 n<-nrow(a); k<-diag(n); itel<-1; mx<-0; saa<-sum(a^2)
33 repeat {
34   for (i in 1:(n-1)) for (j in (i+1):n) {
35     aij<-a[i,j]; bij<-abs(aij);
36     if (bij < eps1) next()
37     mx<-max(mx,bij)
38     am<-(a[i,i]-a[j,j])/2
39     u<-c(aij,-am); u<-u/sqrt(sum(u^2))
40     c<-sqrt((1+u[2])/2); s<-sign(u[1])*sqrt((1-u[2])/2)
41     ss<-s^2; cc<-c^2; sc<-s*c
42     ai<-a[i,]; aj<-a[,j]
43     aii<-a[i,i]; ajj<-a[j,j]
44     a[i,]<-a[i,]-c*ai-s*aj
45     a[,j]<-a[,j]-s*ai+c*aj
46     a[i,j]<-a[j,i]<-0
47     a[i,i]<-aii*cc+ajj*ss-2*sc*aij
48     a[j,j]<-ajj*cc+aii*ss+2*sc*aij
49     if (vectors) {
50       ki<-k[,i]; kj<-k[,j]
51       k[,i]<-c*ki-s*kj

```

```

52         k[,j]<-s*ki+c*kj
53     }
54 }
55 ff<-sqrt(saa-sum(diag(a)^2))
56 if (verbose)
57     cat("Iteration ",formatC(itel,digits=4),"maxel ",formatC(mx,width=10),"
        loss ",formatC(ff,width=10),"\n")
58 if ((mx < eps1) || (ff < eps2) || (itel == itmax)) break()
59 itel<-itel+1; mx<-0
60 }
61 d<-diag(a); o<-order(d,decreasing=TRUE)
62 if (vectors) return(list(values=d[o],vectors=k[,o]))
63 else return(values=d[o])
64 }
65
66 jSVD<-function(x,eps1=1e-6,eps2=1e-6,itmax=1000,vectors=TRUE,verbose=FALSE) {
67 n<-nrow(x); m<-ncol(x); itel<-1; mx<-0
68 kkk<-diag(n); lll<-diag(m);
69 sxx<-sum(x^2); sxm<-sqrt(sxx/(n*m))
70 repeat {
71     for (i in 1:(n-1)) {
72         if (i > m) next()
73         for (j in (i+1):n) {
74             xi<-x[i,]; xj<-x[j,]
75             xij<-ifelse(j > m,0,x[i,j])
76             xjj<-ifelse(j > m,0,x[j,j])
77             xii<-x[i,i]; xji<-x[j,i]
78             mx<-max(mx,abs(xij)/sxm,abs(xji)/sxm)
79             v<-matrix(0,2,2)
80             v[1,1]<-xij^2+xji^2
81             v[1,2]<-v[2,1]<-xii*xji-xjj*xij
82             v[2,2]<-xii^2+xjj^2
83             u<-eigen(v)$vectors[,1]
84             x[i,]<-u[2]*xi+u[1]*xj
85             x[j,]<-u[2]*xj-u[1]*xi
86             if (vectors) {
87                 ki<-kkk[i,]; kj<-kkk[j,]
88                 kkk[i,]<-u[2]*ki+u[1]*kj
89                 kkk[j,]<-u[2]*kj-u[1]*ki
90             }
91         }
92     }
93     ff<-sqrt((sxx-sum(diag(x)^2))/sxx)
94     if (verbose)
95         cat(" Left iteration ",formatC(itel,digits=4),"maxel ",formatC(mx,width
        =10),"loss ",formatC(ff,width=10),"\n")
96     for (k in 1:(m-1)) {
97         if (k > n) next()
98         for (l in (k+1):m) {
99             xk<-x[,k]; xl<-x[,l]
100            xlk<-ifelse(l > n,0,x[l,k])
101            xll<-ifelse(l > n,0,x[l,l])
102            xkk<-x[k,k]; xkl<-x[k,l]

```

```

103     mx<-max(mx,abs(xk1)/sxm,abs(xlk)/sxm)
104     v<-matrix(0,2,2)
105     v[1,1]<-xk1^2+xlk^2
106     v[1,2]<-v[2,1]<-xll*xlk-xkk*xk1
107     v[2,2]<-xkk^2+xll^2
108     u<-eigen(v)$vectors[,1]
109     x[,k]<-u[2]*xk-u[1]*xl
110     x[,l]<-u[1]*xk+u[2]*xl
111     if (vectors) {
112         lk<-l11[,k]; ll<-l11[,l]
113         l11[,k]<-u[2]*lk-u[1]*ll
114         l11[,l]<-u[1]*lk+u[2]*ll
115     }
116     }
117     }
118     ff<-sqrt((sxx-sum(diag(x)^2))/sxx)
119     if (verbose)
120         cat("Right iteration ",formatC(itel,digits=4),"maxel ",formatC(mx,width
121           =10),"loss ",formatC(ff,width=10),"\\n")
122     if ((mx < eps1) || (ff < eps2) || (itel == itmax)) break()
123     itel<-itel+1; mx<-0
124 }
125 }
126
127 jSimDiag<-function(a,eps=1e-10,itmax=100,vectors=TRUE,verbose=FALSE) {
128     n<-dim(a)[1]; kk<-diag(n); m<-dim(a)[3]; itel<-1; saa<-sum(a^2)
129     fold<-saa-sum(apply(a,3,function(x) sum(diag(x^2))))
130     repeat {
131         for (i in 1:(n-1)) for (j in (i+1):n) {
132             ad<-(a[i,i]-a[j,j])/2
133             av<-a[i,j,]
134             v<-matrix(0,2,2)
135             v[1,1]<-sum(ad^2)
136             v[1,2]<-v[2,1]<-sum(av*ad)
137             v[2,2]<-sum(av^2)
138             u<-eigen(v)$vectors[,2]
139             c<-sqrt((1+u[2])/2); s<-sign(u[1])*sqrt((1-u[2])/2)
140             for (k in 1:m) {
141                 ss<-s^2; cc<-c^2; sc<-s*c
142                 ai<-a[i,,k]; aj<-a[j,,k]
143                 aii<-a[i,i,k]; ajj<-a[j,j,k]; aij<-a[i,j,k]
144                 a[i,,k]<-a[i,i,k]*c*ai-s*aj
145                 a[j,,k]<-a[j,j,k]*s*ai+c*aj
146                 a[i,j,k]<-a[j,i,k]*u[1]*(aii-ajj)/2+u[2]*aij
147                 a[i,i,k]<-aii*cc+ajj*ss-2*sc*aij
148                 a[j,j,k]<-ajj*cc+aii*ss+2*sc*aij
149             }
150             if (vectors) {
151                 ki<-kk[,i]; kj<-kk[,j]
152                 kk[,i]<-c*ki-s*kj
153                 kk[,j]<-s*ki+c*kj
154             }

```

```

155     }
156     fnew<-saa-sum(apply(a,3,function(x) sum(diag(x^2))))
157     if (verbose)
158         cat("Iteration ",formatC(itel,digits=4),"old loss ",formatC(fold,width=10)
159             ,"new loss ",formatC(fnew,width=10),"\n")
159     if (((fold-fnew) < eps) || (itel == itmax)) break()
160     itel<-itel+1; fold<-fnew
161     }
162     return(list(a=a,d<-apply(a,3,diag),k=kk))
163 }
164
165 jSimSVD<-function(x,eps=1e-6,itmax=1000,vectors=TRUE,verbose=FALSE) {
166 n<-dim(x)[1]; m<-dim(x)[2]; nmat<-dim(x)[3]; itel<-1
167 kkk<-diag(n); lll<-diag(m); sxx<-sum(x^2); fold<-Inf
168 print(dim(x))
169 repeat {
170     for (i in 1:(n-1)) {
171         if (i > m) next()
172         for (j in (i+1):n) {
173             v<-matrix(0,2,2)
174             for (imat in 1:nmat) {
175                 xij<-ifelse(j > m,0,x[i,j,imat])
176                 xjj<-ifelse(j > m,0,x[j,j,imat])
177                 xii<-x[i,i,imat]; xji<-x[j,i,imat]
178                 v[1,1]<-v[1,1]+(xij^2+xji^2)
179                 v[1,2]<-v[2,1]<-v[1,2]+(xii*xji-xjj*xij)
180                 v[2,2]<-v[2,2]+(xii^2+xjj^2)
181             }
182             u<-eigen(v)$vectors[,1]
183             for (imat in 1:nmat) {
184                 xi<-x[i,,imat]; xj<-x[j,,imat]
185                 x[i,,imat]<-u[2]*xi+u[1]*xj
186                 x[j,,imat]<-u[2]*xj-u[1]*xi
187             }
188             if (vectors) {
189                 ki<-kkk[i,]; kj<-kkk[j,]
190                 kkk[i,]<-u[2]*ki+u[1]*kj
191                 kkk[j,]<-u[2]*kj-u[1]*ki
192             }
193         }
194     }
195     ss<-sum(apply(x,3,diag)^2); fnew<-sqrt((sxx-ss)/sxx)
196     if (verbose)
197         cat(" Left iteration ",formatC(itel,digits=4),"loss ",formatC(fnew,digits
198             =6,width=10),"\n")
199     for (k in 1:(m-1)) {
200         if (k > n) next()
201         for (l in (k+1):m) {
202             v<-matrix(0,2,2)
203             for (imat in 1:nmat) {
204                 xlk<-ifelse(l > n,0,x[l,k,imat])
205                 xll<-ifelse(l > n,0,x[l,l,imat])
206                 xkl<-x[k,l,imat]; xkk<-x[k,k,imat]

```

```

206     v[1,1]<-v[1,1]+(xk1^2+x1k^2)
207     v[1,2]<-v[2,1]<-v[1,2]+(x11*x1k-xk1*xk1)
208     v[2,2]<-v[2,2]+(xkk^2+x11^2)
209     }
210     u<-eigen(v)$vectors[,1]
211     for (imat in 1:nmat) {
212         xk<-x[,k,imat]; x1<-x[,1,imat]
213         x[,k,imat]<-u[2]*xk-u[1]*x1
214         x[,1,imat]<-u[1]*xk+u[2]*x1
215     }
216     if (vectors) {
217         lk<-l11[,k]; l1<-l11[,1]
218         l11[,k]<-u[2]*lk-u[1]*l1
219         l11[,1]<-u[1]*lk+u[2]*l1
220     }
221     }
222 }
223 ss<-sum(apply(x,3,diag)^2); fnew<-sqrt((sxx-ss)/sxx)
224 if (verbose)
225     cat("Right iteration ",formatC(itel,digits=4),"loss ",formatC(fnew,digits
      =6,width=10),"\n")
226 if (((fold - fnew) < eps) || (itel == itmax)) break()
227 itel<-itel+1; fold<-fnew
228 }
229 return(list(d=apply(x,3,diag),u=t(kkk),v=l11))
230 }
231
232 jTucker3Diag<-function(a,eps=1e-6,itmax=100,vectors=TRUE,verbose=TRUE) {
233 n<-dim(a)[1]; m<-dim(a)[2]; k<-dim(a)[3]; nmk<-min(n,m,k)
234 kn<-diag(n); km<-diag(m); kk<-diag(k); ossq<-0; itel<-1
235 repeat {
236     for (i in 1:(n-1)) for (j in (i+1):n) {
237         ai<-a[i,,]; aj<-a[,j,,]
238         acc<-ass<-asc<-0
239         if (i <= min(m,k)) {
240             acc<-acc+ai[i,i,i]^2
241             ass<-ass+aj[j,i,i]^2
242             asc<-asc+ai[i,i,i]*aj[j,i,i]
243         }
244         if (j <= min(m,k)) {
245             acc<-acc+aj[j,j,j]^2
246             ass<-ass+ai[i,j,j]^2
247             asc<-asc-aj[j,j,j]*ai[i,j,j]
248         }
249         u<-eigen(matrix(c(acc,asc,asc,ass),2,2))$vectors[,1]
250         c<-u[1]; s<-u[2]
251         a[i,,]<-c*ai+s*aj
252         a[,j,,]<-c*aj-s*ai
253         if (vectors) {
254             ki<-kn[i,]; kj<-kn[j,]
255             kn[i,]<-c*ki+s*kj
256             kn[,j]<-c*kj-s*ki
257         }

```



```

258     }
259     for (i in 1:(m-1)) for (j in (i+1):m) {
260         ai<-a[,i,]; aj<-a[,j,]
261         acc<-ass<-asc<-0
262         if (i <= min(n,k)) {
263             acc<-acc+a[i,i,i]^2
264             ass<-ass+a[i,j,i]^2
265             asc<-asc+a[i,i,i]*a[i,j,i]
266         }
267         if (j <= min(n,k)) {
268             acc<-acc+a[j,j,j]^2
269             ass<-ass+a[j,i,j]^2
270             asc<-asc-a[j,j,j]*a[j,i,j]
271         }
272         u<-eigen(matrix(c(acc,asc,asc,ass),2,2))$vectors[,1]
273         c<-u[1]; s<-u[2]
274         a[,i,]<-c*ai+s*aj
275         a[,j,]<-c*aj-s*ai
276         if (vectors) {
277             ki<-km[i,]; kj<-km[j,]
278             km[i,]<-c*ki+s*kj
279             km[j,]<-c*kj-s*ki
280         }
281     }
282     for (i in 1:(k-1)) for (j in (i+1):k) {
283         ai<-a[,,i]; aj<-a[,,j]
284         acc<-ass<-asc<-0
285         if (i <= min(n,m)) {
286             acc<-acc+a[i,i,i]^2
287             ass<-ass+a[i,i,j]^2
288             asc<-asc+a[i,i,i]*a[i,i,j]
289         }
290         if (j <= min(n,m)) {
291             acc<-acc+a[j,j,j]^2
292             ass<-ass+a[j,j,i]^2
293             asc<-asc-a[j,j,j]*a[j,j,i]
294         }
295         u<-eigen(matrix(c(acc,asc,asc,ass),2,2))$vectors[,1]
296         c<-u[1]; s<-u[2]
297         a[,,i]<-c*ai+s*aj
298         a[,,j]<-c*aj-s*ai
299         if (vectors) {
300             ki<-kk[i,]; kj<-kk[j,]
301             kk[i,]<-c*ki+s*kj
302             kk[j,]<-c*kj-s*ki
303         }
304     }
305     nssq<-0; for (v in 1:nmk) nssq<-nssq+a[v,v,v]^2
306     if (verbose)
307         cat("Iteration ",formatC(itel,digits=4), "ssq ",formatC(nssq,digits=10,
308             width=15),"\n")
309     if ((nssq - ossq) < eps) || (itel == itmax) break()
310     itel<-itel+1; ossq<-nssq

```

```

310     }
311     d<-rep(0,nmk); for (v in 1:nmk) d[v]<-a[v,v]
312     return(list(a=a,d=d,kn=kn,km=km,kk=kk))
313 }
314
315 jTucker3Block<-function(a,dims,eps=1e-6,itmax=100,vectors=TRUE,verbose=TRUE) {
316 n<-dim(a)[1]; m<-dim(a)[2]; k<-dim(a)[3]; nmk<-min(n,m,k)
317 p<-dims[1]; q<-dims[2]; r<-dims[3]
318 kn<-diag(n); km<-diag(m); kk<-diag(k); ossq<-0; itel<-1
319 repeat {
320     for (i in 1:(n-1)) for (j in (i+1):n) {
321         ai<-a[i,]; aj<-a[j,,]
322         acc<-ass<-asc<-0
323         if (i <= p)
324             for (u in 1:q) for (v in 1:r) {
325                 acc<-acc+a[i,u,v]^2
326                 ass<-ass+a[j,u,v]^2
327                 asc<-asc+a[i,u,v]*a[j,u,v]
328             }
329         if (j <= p)
330             for (u in 1:q) for (v in 1:r) {
331                 acc<-acc+a[j,u,v]^2
332                 ass<-ass+a[i,u,v]^2
333                 asc<-asc-a[j,u,v]*a[i,u,v]
334             }
335         u<-eigen(matrix(c(acc,asc,asc,ass),2,2))$vectors[,1]
336         c<-u[1]; s<-u[2]
337         a[i,,]<-c*ai+s*aj
338         a[j,,]<-c*aj-s*ai
339         if (vectors) {
340             ki<-kn[i,]; kj<-kn[j,]
341             kn[i,]<-c*ki+s*kj
342             kn[j,]<-c*kj-s*ki
343         }
344     }
345     for (i in 1:(m-1)) for (j in (i+1):m) {
346         ai<-a[,i,]; aj<-a[,j,]
347         acc<-ass<-asc<-0
348         if (i <= q)
349             for (u in 1:p) for (v in 1:r) {
350                 acc<-acc+a[u,i,v]^2
351                 ass<-ass+a[u,j,v]^2
352                 asc<-asc+a[u,i,v]*a[u,j,v]
353             }
354         if (j <= q)
355             for (u in 1:p) for (v in 1:r) {
356                 acc<-acc+a[u,j,v]^2
357                 ass<-ass+a[u,i,v]^2
358                 asc<-asc-a[u,i,v]*a[u,j,v]
359             }
360         u<-eigen(matrix(c(acc,asc,asc,ass),2,2))$vectors[,1]
361         c<-u[1]; s<-u[2]
362         a[,i,]<-c*ai+s*aj

```

```

363     a[,j] <- c*aj - s*ai
364     if (vectors) {
365         ki <- km[i,]; kj <- km[j,]
366         km[i,] <- c*ki + s*kj
367         km[j,] <- c*kj - s*ki
368     }
369 }
370 for (i in 1:(k-1)) for (j in (i+1):k) {
371     ai <- a[,i]; aj <- a[,j]
372     acc <- ass <- asc <- 0
373     if (i <= r)
374         for (u in 1:p) for (v in 1:g) {
375             acc <- acc + a[u,v,i]^2
376             ass <- ass + a[u,v,j]^2
377             asc <- asc + a[u,v,i]*a[u,v,j]
378         }
379     if (j <= r)
380         for (u in 1:p) for (v in 1:g) {
381             acc <- acc + a[u,v,j]^2
382             ass <- ass + a[u,v,i]^2
383             asc <- asc - a[u,v,i]*a[u,v,j]
384         }
385     u <- eigen(matrix(c(acc, asc, asc, ass), 2, 2))$vectors[,1]
386     c <- u[1]; s <- u[2]
387     a[,i] <- c*ai + s*aj
388     a[,j] <- c*aj - s*ai
389     if (vectors) {
390         ki <- kk[i,]; kj <- kk[j,]
391         kk[i,] <- c*ki + s*kj
392         kk[j,] <- c*kj - s*ki
393     }
394 }
395 nssq <- 0; for (i in 1:p) for (j in 1:g) for (l in 1:r) nssq <- nssq + a[i,j,l]^2
396 if (verbose)
397     cat("Iteration ", formatC(itel, digits=4), "ssq ", formatC(nssq, digits=10,
398         width=15), "\n")
399     if ((nssq - ossq) < eps) || (itel == itmax) break()
400     itel <- itel + 1; ossq <- nssq
401 }
402 d <- a[1:p, 1:g, 1:r]
403 return(list(a=a, d=d, kn=kn, km=km, kk=kk))
404 }
405 jMCA <- function(burt, k, eps=1e-6, itmax=500, verbose=TRUE, vectors=TRUE) {
406     m <- length(k); burt <- m*m*burt/sum(burt); sk <- sum(k)
407     db <- diag(burt); ll <- kk <- ww <- diag(sk); itel <- 1; ossq <- 0
408     klw <- 1 + cumsum(c(0, k))[1:m]; kup <- cumsum(k)
409     ind <- lapply(1:m, function(i) klw[i]:kup[i])
410     sburt <- burt/sqrt(outer(db, db))
411     for (i in 1:m)
412         kk[ind[[i]], ind[[i]]] <- t(svd(sburt[ind[[i]],])$u)
413     kbk <- kk%*%sburt%*%t(kk)

```

```

414 for (i in 1:m) for (j in 1:m)
415   ww[ind[[i]],ind[[j]]]<-ifelse(outer(1:k[i],1:k[j],"=="),1,0)
416 repeat {
417   for (l in 1:m) {
418     if (k[l] == 2) next()
419     li<-ind[[l]]
420     for (i in (klw[l]+1):(kup[l]-1)) for (j in (i+1):kup[l]) {
421       bi<-kbb[i,-li]; bj<-kbb[j,-li]
422       wi<-ww[i,-li]; wj<-ww[j,-li]
423       acc<-sum(wi*bi^2)+sum(wj*bj^2)
424       acs<-sum((wi-wj)*bi*bj)
425       ass<-sum(wi*bj^2)+sum(wj*bi^2)
426       u<-eigen(matrix(c(acc,acs,acs,ass),2,2))$vectors[,1]
427       c<-u[1]; s<-u[2]
428       kbb[-li,i]<-kbb[i,-li]<-c*s*bi+s*bj
429       kbb[-li,j]<-kbb[j,-li]<-c*s*bj-s*bi
430       if (vectors) {
431         ki<-kk[i,li]; kj<-kk[j,li]
432         kk[i,li]<-c*s*ki+s*kj
433         kk[j,li]<-c*s*kj-s*ki
434       }
435     }
436   }
437   nssq<-sum(ww*kbb^2)-sum(diag(kbb)^2)
438   if (verbose)
439     cat("Iteration ",formatC(itel,digits=4),"ssq ",formatC(nssq,digits=10,
440       width=15),"\n")
441   if ((nssq - ossq) < eps) || (itel == itmax) break()
442   itel<-itel+1; ossq<-nssq
443 }
444 k1<-unlist(sapply(k,function(i) 1:i))
445 pp<-ifelse(outer(1:sk,order(k1),"=="),1,0)
446 pkbkp<-t(pp)%*%kbb%*%pp
447 pk<-t(pp)%*%kk
448 km<-as.vector(table(k1)); nm<-length(km)
449 klw<-1+cumsum(c(0,km))[1:nm]; kup<-cumsum(km)
450 for (i in 1:length(km)) {
451   if (km[i]==1) next()
452   ind<-klw[i]:kup[i]
453   ll[ind,ind]<-eigen(pkbkp[ind,ind])$vectors
454 }
455 lpbkpl<-t(ll)%*%pkbkp%*%ll
456 lpk<-t(ll)%*%pk
457 return(list(kbb=kbb,pkbkp=pkbkp,lpbkpl=lpbkpl,kk=t(kk),pp=pp,ll=ll,kp1=t(lpk)))

```

DEPARTMENT OF STATISTICS, UNIVERSITY OF CALIFORNIA, LOS ANGELES, CA 90095-1554

E-mail address, Jan de Leeuw: deleeuw@stat.ucla.edu

URL, Jan de Leeuw: <http://gifi.stat.ucla.edu>