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 $f(K_1, \cdots, K_n)$ 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

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

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

Date: Wednesday 10th December, 2008 — 18h 5min — Typeset in TIMES ROMAN.

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

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

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

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

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

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

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

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

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

Equations (3a) and (3b) are one-sided left and right Jacobi transformations of X. The two-sided transform is defined by $\tilde{X} = K_{ij}(\theta)XK_{k\ell}(\xi)$. We find

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

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

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

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 $2 s_\xi c_\xi = u_1$, $c_\xi^2 - s_\xi^2 = u_2$, 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

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

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_ν they generally cannot be simultaneously diagonalized by an orthogonal transformation K. We can find K such that $K'A_\nu K$ is diagonal for all ν if and only if the A_ν commute in pairs, i.e. if and only if $A_\nu A_\mu = A_\mu A_\nu$ for all ν, μ. We can find K, however, such that the transformed $K'A_\nu K$ 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

$$\sum_{\nu=1}^{m}[\frac{1}{2}\sin(2\xi)(a_{ii\nu}-a_{jj\nu})+\cos(2\xi)a_{ij\nu}]^2$$

is minimized. Let $d_{ij\nu} = \frac{1}{2}(a_{ii\nu}-a_{jj\nu})$. Define a 2×2 matrix V with

$$v_{11} = \sum_{\nu=1}^{m} d_{ij\nu}^2,$$

$$v_{12} = v_{21} = \sum_{\nu=1}^{m} d_{ij\nu}a_{ij\nu},$$

$$v_{22} = \sum_{\nu=1}^{m} a_{ij\nu}^2,$$

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,

$$c_\xi = \sqrt{\frac{1+u_2}{2}},$$

$$s_\xi = \mathbf{sign}(u_1)\sqrt{\frac{1-u_2}{2}}.$$

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 $X_1, \cdots, X_m$ 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 `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 varables. Variable j has k_j categories. If $G = \begin{bmatrix} G_1 & \cdots & G_m \end{bmatrix}$ 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 varable j on the diagonal. Also define $D = \mathbf{diag}(C)$, and the scaled Burt matrix $A = D^{-\frac{1}{2}} C D^{-\frac{1}{2}}$.

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 pont 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 Komponenten 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 Rijckevorsel 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 Lest 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

```
#
#   jacobi package
#   Copyright (C) 2008  Jan de Leeuw <deleeuw@stat.ucla.edu>
#   UCLA Department of Statistics, Box 951554, Los Angeles, CA 90095-1554
#
#   This program is free software; you can redistribute it and/or modify
#   it under the terms of the GNU General Public License as published by
#   the Free Software Foundation; either version 2 of the License, or
#   (at your option) any later version.
#
#   This program is distributed in the hope that it will be useful,
#   but WITHOUT ANY WARRANTY; without even the implied warranty of
#   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#   GNU General Public License for more details.
#
#   You should have received a copy of the GNU General Public License
#   along with this program; if not, write to the Free Software
#   Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
#
###################################################################
#
# version 0.0.1, 2008-12-05   Eigen and SVD
# version 0.1.0, 2008-12-06   Simultaneous Diagonalization
# version 0.2.0, 2008-12-08   Simultaneous SVD
# version 0.2.1, 2008-12-08   Various Bugfixes
# version 0.2.2, 2008-12-08   Small efficiency gains
# version 0.3.0, 2008-12-10   PREHOM in jMCA added
# version 1.0.0, 2008-12-10   Tucker3 Methods added
#

jEigen<-function(a,eps1=1e-6,eps2=1e-10,itmax=100,vectors=TRUE,verbose=FALSE) {
n<-nrow(a); k<-diag(n); itel<-1; mx<-0; saa<-sum(a^2)
repeat {
    for (i in 1:(n-1)) for (j in (i+1):n) {
        aij<-a[i,j]; bij<-abs(aij);
        if (bij < eps1) next()
        mx<-max(mx,bij)
        am<-(a[i,i]-a[j,j])/2
        u<-c(aij,-am); u<-u/sqrt(sum(u^2))
        c<-sqrt((1+u[2])/2); s<-sign(u[1])*sqrt((1-u[2])/2)
        ss<-s^2; cc<-c^2; sc<-s*c
        ai<-a[i,]; aj<-a[j,]
        aii<-a[i,i]; ajj<-a[j,j]
        a[i,]<-a[,i]<-c*ai-s*aj
        a[j,]<-a[,j]<-s*ai+c*aj
        a[i,j]<-a[j,i]<-0
        a[i,i]<-aii*cc+ajj*ss-2*sc*aij
        a[j,j]<-ajj*cc+aii*ss+2*sc*aij
        if (vectors) {
            ki<-k[,i]; kj<-k[,j]
            k[,i]<-c*ki-s*kj
```

```
            k[,j]<-s*ki+c*kj
            }
        }
    ff<-sqrt(saa-sum(diag(a)^2))
    if (verbose)
        cat("Iteration ",formatC(itel,digits=4),"maxel ",formatC(mx,width=10),"loss ",formatC(ff,width=10),"\n")
    if ((mx < eps1) || (ff < eps2) || (itel == itmax)) break()
    itel<-itel+1; mx<-0
    }
d<-diag(a); o<-order(d,decreasing=TRUE)
if (vectors) return(list(values=d[o],vectors=k[,o]))
    else return(values=d[o])
}

jSVD<-function(x,eps1=1e-6,eps2=1e-6,itmax=1000,vectors=TRUE,verbose=FALSE) {
n<-nrow(x); m<-ncol(x); itel<-1; mx<-0
kkk<-diag(n); lll<-diag(m);
sxx<-sum(x^2); sxm<-sqrt(sxx/(n*m))
repeat {
    for (i in 1:(n-1)) {
                if (i > m) next()
                for (j in (i+1):n) {
                xi<-x[i,]; xj<-x[j,]
                xij<-ifelse(j > m,0,x[i,j])
                xjj<-ifelse(j > m,0,x[j,j])
                        xii<-x[i,i]; xji<-x[j,i]
                mx<-max(mx,abs(xij)/sxm,abs(xji)/sxm)
                v<-matrix(0,2,2)
                v[1,1]<-xij^2+xji^2
                v[1,2]<-v[2,1]<-xii*xji-xjj*xij
                v[2,2]<-xii^2+xjj^2
                u<-eigen(v)$vectors[,1]
                x[i,]<-u[2]*xi+u[1]*xj
                x[j,]<-u[2]*xj-u[1]*xi
                if (vectors) {
                    ki<-kkk[i,]; kj<-kkk[j,]
                    kkk[i,]<-u[2]*ki+u[1]*kj
                    kkk[j,]<-u[2]*kj-u[1]*ki
                    }
                }
                }
    ff<-sqrt((sxx-sum(diag(x)^2))/sxx)
    if (verbose)
        cat(" Left iteration ",formatC(itel,digits=4),"maxel ",formatC(mx,width=10),"loss ",formatC(ff,width=10),"\n")
    for (k in 1:(m-1)) {
        if (k > n) next()
                for (l in (k+1):m) {
                xk<-x[,k]; xl<-x[,l]
                xlk<-ifelse(l > n,0,x[l,k])
                xll<-ifelse(l > n,0,x[l,l])
                xkk<-x[k,k]; xkl<-x[k,l]
```

```
                mx<-max(mx,abs(xkl)/sxm,abs(xlk)/sxm)
                v<-matrix(0,2,2)
                v[1,1]<-xkl^2+xlk^2
                v[1,2]<-v[2,1]<-xll*xlk-xkk*xkl
                v[2,2]<-xkk^2+xll^2
                u<-eigen(v)$vectors[,1]
                x[,k]<-u[2]*xk-u[1]*xl
                x[,l]<-u[1]*xk+u[2]*xl
                if (vectors) {
                    lk<-lll[,k]; ll<-lll[,l]
                    lll[,k]<-u[2]*lk-u[1]*ll
                    lll[,l]<-u[1]*lk+u[2]*ll
                    }
                }
                }
    ff<-sqrt((sxx-sum(diag(x)^2))/sxx)
    if (verbose)
        cat("Right iteration ",formatC(itel,digits=4),"maxel ",formatC(mx,width
             =10),"loss ",formatC(ff,width=10),"\n")
    if ((mx < eps1) || (ff < eps2) || (itel == itmax)) break()
    itel<-itel+1; mx<-0
    }
return(list(d=diag(x),u=t(kkk),v=lll))
}

jSimDiag<-function(a,eps=1e-10,itmax=100,vectors=TRUE,verbose=FALSE) {
n<-dim(a)[1]; kk<-diag(n); m<-dim(a)[3]; itel<-1; saa<-sum(a^2)
fold<-saa-sum(apply(a,3,function(x) sum(diag(x^2))))
repeat {
    for (i in 1:(n-1)) for (j in (i+1):n) {
    ad<-(a[i,i,]-a[j,j,])/2
    av<-a[i,j,]
    v<-matrix(0,2,2)
    v[1,1]<-sum(ad^2)
    v[1,2]<-v[2,1]<-sum(av*ad)
    v[2,2]<-sum(av^2)
    u<-eigen(v)$vectors[,2]
    c<-sqrt((1+u[2])/2); s<-sign(u[1])*sqrt((1-u[2])/2)
    for (k in 1:m) {
            ss<-s^2; cc<-c^2; sc<-s*c
            ai<-a[i,,k]; aj<-a[j,,k]
            aii<-a[i,i,k]; ajj<-a[j,j,k]; aij<-a[i,j,k]
            a[i,,k]<-a[,i,k]<-c*ai-s*aj
            a[j,,k]<-a[,j,k]<-s*ai+c*aj
            a[i,j,k]<-a[j,i,k]<-u[1]*(aii-ajj)/2+u[2]*aij
            a[i,i,k]<-aii*cc+ajj*ss-2*sc*aij
            a[j,j,k]<-ajj*cc+aii*ss+2*sc*aij
            }
    if (vectors) {
        ki<-kk[,i]; kj<-kk[,j]
        kk[,i]<-c*ki-s*kj
        kk[,j]<-s*ki+c*kj
        }
```

```
    }
    fnew<-saa-sum(apply(a,3,function(x) sum(diag(x^2))))
    if (verbose)
        cat("Iteration ",formatC(itel,digits=4),"old loss ",formatC(fold,width=10)
             ,"new loss ",formatC(fnew,width=10),"\n")
    if (((fold-fnew) < eps) || (itel == itmax)) break()
    itel<-itel+1; fold<-fnew
    }
return(list(a=a,d<-apply(a,3,diag),k=kk))
}

jSimSVD<-function(x,eps=1e-6,itmax=1000,vectors=TRUE,verbose=FALSE) {
n<-dim(x)[1]; m<-dim(x)[2]; nmat<-dim(x)[3]; itel<-1
kkk<-diag(n); lll<-diag(m); sxx<-sum(x^2); fold<-Inf
print(dim(x))
repeat {
    for (i in 1:(n-1)) {
        if (i > m) next()
                for (j in (i+1):n) {
                v<-matrix(0,2,2)
                for (imat in 1:nmat) {
                    xij<-ifelse(j > m,0,x[i,j,imat])
                    xjj<-ifelse(j > m,0,x[j,j,imat])
                                xii<-x[i,i,imat]; xji<-x[j,i,imat]
                    v[1,1]<-v[1,1]+(xij^2+xji^2)
                    v[1,2]<-v[2,1]<-v[1,2]+(xii*xji-xjj*xij)
                    v[2,2]<-v[2,2]+(xii^2+xjj^2)
                    }
                u<-eigen(v)$vectors[,1]
                for (imat in 1:nmat) {
                    xi<-x[i,,imat]; xj<-x[j,,imat]
                    x[i,,imat]<-u[2]*xi+u[1]*xj
                    x[j,,imat]<-u[2]*xj-u[1]*xi
                    }
                if (vectors) {
                    ki<-kkk[i,]; kj<-kkk[j,]
                    kkk[i,]<-u[2]*ki+u[1]*kj
                    kkk[j,]<-u[2]*kj-u[1]*ki
                    }
                }
        }
    ss<-sum(apply(x,3,diag)^2); fnew<-sqrt((sxx-ss)/sxx)
    if (verbose)
        cat(" Left iteration ",formatC(itel,digits=4),"loss ",formatC(fnew,digits
             =6,width=10),"\n")
    for (k in 1:(m-1)) {
        if (k > n) next()
                for (l in (k+1):m) {
                v<-matrix(0,2,2)
                for (imat in 1:nmat) {
                    xlk<-ifelse(l > n,0,x[l,k,imat])
                    xll<-ifelse(l > n,0,x[l,l,imat])
                    xkl<-x[k,l,imat]; xkk<-x[k,k,imat]
```

```
                    v[1,1]<-v[1,1]+(xkl^2+xlk^2)
                    v[1,2]<-v[2,1]<-v[1,2]+(xll*xlk-xkk*xkl)
                    v[2,2]<-v[2,2]+(xkk^2+xll^2)
                    }
                u<-eigen(v)$vectors[,1]
                for (imat in 1:nmat) {
                    xk<-x[,k,imat]; xl<-x[,l,imat]
                    x[,k,imat]<-u[2]*xk-u[1]*xl
                    x[,l,imat]<-u[1]*xk+u[2]*xl
                    }
                if (vectors) {
                    lk<-lll[,k]; ll<-lll[,l]
                    lll[,k]<-u[2]*lk-u[1]*ll
                    lll[,l]<-u[1]*lk+u[2]*ll
                    }
                }
        }
    ss<-sum(apply(x,3,diag)^2); fnew<-sqrt((sxx-ss)/sxx)
    if (verbose)
        cat("Right iteration ",formatC(itel,digits=4),"loss ",formatC(fnew,digits=6,width=10),"\n")
    if (((fold - fnew) < eps) || (itel == itmax)) break()
    itel<-itel+1; fold<-fnew
    }
return(list(d=apply(x,3,diag),u=t(kkk),v=lll))
}

jTucker3Diag<-function(a,eps=1e-6,itmax=100,vectors=TRUE,verbose=TRUE) {
n<-dim(a)[1]; m<-dim(a)[2]; k<-dim(a)[3]; nmk<-min(n,m,k)
kn<-diag(n); km<-diag(m); kk<-diag(k); ossq<-0; itel<-1
repeat {
        for (i in 1:(n-1)) for (j in (i+1):n) {
                ai<-a[i,,]; aj<-a[j,,]
                acc<-ass<-asc<-0
                if (i <= min(m,k)) {
                        acc<-acc+a[i,i,i]^2
                        ass<-ass+a[j,i,i]^2
                        asc<-asc+a[i,i,i]*a[j,i,i]
                        }
                if (j <= min(m,k)) {
                        acc<-acc+a[j,j,j]^2
                        ass<-ass+a[i,j,j]^2
                        asc<-asc-a[j,j,j]*a[i,j,j]
                        }
                u<-eigen(matrix(c(acc,asc,asc,ass),2,2))$vectors[,1]
                c<-u[1]; s<-u[2]
                a[i,,]<-c*ai+s*aj
                a[j,,]<-c*aj-s*ai
                if (vectors) {
                    ki<-kn[i,]; kj<-kn[j,]
                    kn[i,]<-c*ki+s*kj
                    kn[j,]<-c*kj-s*ki
                        }
```

```
            }
        for (i in 1:(m-1)) for (j in (i+1):m) {
            ai<-a[,i,]; aj<-a[,j,]
            acc<-ass<-asc<-0
            if (i <= min(n,k)) {
                acc<-acc+a[i,i,i]^2
                ass<-ass+a[i,j,i]^2
                asc<-asc+a[i,i,i]*a[i,j,i]
                }
            if (j <= min(n,k)) {
                acc<-acc+a[j,j,j]^2
                ass<-ass+a[j,i,j]^2
                asc<-asc-a[j,j,j]*a[j,i,j]
                }
            u<-eigen(matrix(c(acc,asc,asc,ass),2,2))$vectors[,1]
            c<-u[1]; s<-u[2]
            a[,i,]<-c*ai+s*aj
            a[,j,]<-c*aj-s*ai
            if (vectors) {
              ki<-km[i,]; kj<-km[j,]
              km[i,]<-c*ki+s*kj
              km[j,]<-c*kj-s*ki
                }
            }
        for (i in 1:(k-1)) for (j in (i+1):k) {
            ai<-a[,,i]; aj<-a[,,j]
            acc<-ass<-asc<-0
            if (i <= min(n,m)) {
                acc<-acc+a[i,i,i]^2
                ass<-ass+a[i,i,j]^2
                asc<-asc+a[i,i,i]*a[i,i,j]
                }
            if (j <= min(n,m)) {
                acc<-acc+a[j,j,j]^2
                ass<-ass+a[j,j,i]^2
                asc<-asc-a[j,j,j]*a[j,j,i]
                }
            u<-eigen(matrix(c(acc,asc,asc,ass),2,2))$vectors[,1]
            c<-u[1]; s<-u[2]
            a[,,i]<-c*ai+s*aj
            a[,,j]<-c*aj-s*ai
            if (vectors) {
              ki<-kk[i,]; kj<-kk[j,]
              kk[i,]<-c*ki+s*kj
              kk[j,]<-c*kj-s*ki
                }
            }
        nssq<-0; for (v in 1:nmk) nssq<-nssq+a[v,v,v]^2
    if (verbose)
        cat("Iteration ",formatC(itel,digits=4),"ssq ",formatC(nssq,digits=10,
            width=15),"\n")
    if (((nssq - ossq) < eps) || (itel == itmax)) break()
    itel<-itel+1; ossq<-nssq
```

```
    }
d<-rep(0,nmk); for (v in 1:nmk) d[v]<-a[v,v,v]
return(list(a=a,d=d,kn=kn,km=km,kk=kk))
}

jTucker3Block<-function(a,dims,eps=1e-6,itmax=100,vectors=TRUE,verbose=TRUE) {
n<-dim(a)[1]; m<-dim(a)[2]; k<-dim(a)[3]; nmk<-min(n,m,k)
p<-dims[1]; q<-dims[2]; r<-dims[3]
kn<-diag(n); km<-diag(m); kk<-diag(k); ossq<-0; itel<-1
repeat {
        for (i in 1:(n-1)) for (j in (i+1):n) {
                ai<-a[i,,]; aj<-a[j,,]
                acc<-ass<-asc<-0
                if (i <= p)
                        for (u in 1:q) for (v in 1:r) {
                                acc<-acc+a[i,u,v]^2
                                ass<-ass+a[j,u,v]^2
                                asc<-asc+a[i,u,v]*a[j,u,v]
                                }
                if (j <= p)
                        for (u in 1:q) for (v in 1:r) {
                                acc<-acc+a[j,u,v]^2
                                ass<-ass+a[i,u,v]^2
                                asc<-asc-a[j,u,v]*a[i,u,v]
                                }
                u<-eigen(matrix(c(acc,asc,asc,ass),2,2))$vectors[,1]
                c<-u[1]; s<-u[2]
                a[i,,]<-c*ai+s*aj
                a[j,,]<-c*aj-s*ai
                if (vectors) {
                    ki<-kn[i,]; kj<-kn[j,]
                    kn[i,]<-c*ki+s*kj
                    kn[j,]<-c*kj-s*ki
                        }
                }
        for (i in 1:(m-1)) for (j in (i+1):m) {
                ai<-a[,i,]; aj<-a[,j,]
                acc<-ass<-asc<-0
                if (i <= q)
                        for (u in 1:p) for (v in 1:r) {
                                acc<-acc+a[u,i,v]^2
                                ass<-ass+a[u,j,v]^2
                                asc<-asc+a[u,i,v]*a[u,j,v]
                                }
                if (j <= q)
                        for (u in 1:p) for (v in 1:r) {
                                acc<-acc+a[u,j,v]^2
                                ass<-ass+a[u,i,v]^2
                                asc<-asc-a[u,i,v]*a[u,j,v]
                                }
                u<-eigen(matrix(c(acc,asc,asc,ass),2,2))$vectors[,1]
                c<-u[1]; s<-u[2]
                a[,i,]<-c*ai+s*aj
```

```
                a[,j,]<-c*aj-s*ai
                if (vectors) {
                    ki<-km[i,]; kj<-km[j,]
                    km[i,]<-c*ki+s*kj
                    km[j,]<-c*kj-s*ki
                        }
                }
        for (i in 1:(k-1)) for (j in (i+1):k) {
                ai<-a[,,i]; aj<-a[,,j]
                acc<-ass<-asc<-0
                if (i <= r)
                        for (u in 1:p) for (v in 1:q) {
                                acc<-acc+a[u,v,i]^2
                                ass<-ass+a[u,v,j]^2
                                asc<-asc+a[u,v,i]*a[u,v,j]
                                }
                if (j <= r)
                        for (u in 1:p) for (v in 1:q) {
                                acc<-acc+a[u,v,j]^2
                                ass<-ass+a[u,v,i]^2
                                asc<-asc-a[u,v,i]*a[u,v,j]
                                }
                u<-eigen(matrix(c(acc,asc,asc,ass),2,2))$vectors[,1]
                c<-u[1]; s<-u[2]
                a[,,i]<-c*ai+s*aj
                a[,,j]<-c*aj-s*ai
                if (vectors) {
                    ki<-kk[i,]; kj<-kk[j,]
                    kk[i,]<-c*ki+s*kj
                    kk[j,]<-c*kj-s*ki
                        }
                }
        nssq<-0; for (i in 1:p) for (j in 1:q) for (l in 1:r) nssq<-nssq+a[i,j,l]^2
    if (verbose)
        cat("Iteration ",formatC(itel,digits=4),"ssq ",formatC(nssq,digits=10,width=15),"\n")
    if (((nssq - ossq) < eps) || (itel == itmax)) break()
    itel<-itel+1; ossq<-nssq
    }
d<-a[1:p,1:q,1:r]
return(list(a=a,d=d,kn=kn,km=km,kk=kk))
}

jMCA<-function(burt,k,eps=1e-6,itmax=500,verbose=TRUE,vectors=TRUE) {
m<-length(k); burt<-m*m*burt/sum(burt); sk<-sum(k)
db<-diag(burt); ll<-kk<-ww<-diag(sk); itel<-1; ossq<-0
klw<-1+cumsum(c(0,k))[1:m]; kup<-cumsum(k)
ind<-lapply(1:m,function(i) klw[i]:kup[i])
sburt<-burt/sqrt(outer(db,db))
for (i in 1:m)
    kk[ind[[i]],ind[[i]]]<-t(svd(sburt[ind[[i]],])$u)
kbk<-kk%*%sburt%*%t(kk)
```

```
for (i in 1:m) for (j in 1:m)
    ww[ind[[i]],ind[[j]]]<-ifelse(outer(1:k[i],1:k[j],"=="),1,0)
repeat {
    for (l in 1:m) {
        if (k[l] == 2) next()
        li<-ind[[l]]
        for (i in (klw[l]+1):(kup[l]-1)) for (j in (i+1):kup[l]) {
            bi<-kbk[i,-li]; bj<-kbk[j,-li]
            wi<-ww[i,-li]; wj<-ww[j,-li]
            acc<-sum(wi*bi^2)+sum(wj*bj^2)
            acs<-sum((wi-wj)*bi*bj)
            ass<-sum(wi*bj^2)+sum(wj*bi^2)
            u<-eigen(matrix(c(acc,acs,acs,ass),2,2))$vectors[,1]
            c<-u[1]; s<-u[2]
            kbk[-li,i]<-kbk[i,-li]<-c*bi+s*bj
            kbk[-li,j]<-kbk[j,-li]<-c*bj-s*bi
            if (vectors) {
                ki<-kk[i,li]; kj<-kk[j,li]
                kk[i,li]<-c*ki+s*kj
                kk[j,li]<-c*kj-s*ki
                }
            }
        }
    nssq<-sum(ww*kbk^2)-sum(diag(kbk)^2)
    if (verbose)
        cat("Iteration ",formatC(itel,digits=4),"ssq ",formatC(nssq,digits=10,
             width=15),"\n")
    if (((nssq - ossq) < eps) || (itel == itmax)) break()
    itel<-itel+1; ossq<-nssq
    }
kl<-unlist(sapply(k,function(i) 1:i))
pp<-ifelse(outer(1:sk,order(kl),"=="),1,0)
pkbkp<-t(pp)%*%kbk%*%pp
pk<-t(pp)%*%kk
km<-as.vector(table(kl)); nm<-length(km)
klw<-1+cumsum(c(0,km))[1:nm]; kup<-cumsum(km)
for (i in 1:length(km)) {
        if (km[i]==1) next()
        ind<-klw[i]:kup[i]
        ll[ind,ind]<-eigen(pkbkp[ind,ind])$vectors
        }
lpkbkpl<-t(ll)%*%pkbkp%*%ll
lpk<-t(ll)%*%pk
return(list(kbk=kbk,pkbkp=pkbkp,lpkbkpl=lpkbkpl,kk=t(kk),pp=pp,ll=ll,kpl=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