COMPONENT ANALYSIS USING MULTIVARIATE CUMULANTS

JAN DE LEEUW AND IRINA KUKUYEVA

ABSTRACT. This paper discusses several different algorithms to fit linear component analysis approximations to multivariate data matrices. Our algorithms fit multiway models of the Harshman-Carroll or Tucker type to symmetric arrays of cumulants. We cover both orthogonal and independent component and factor analysis, while allowing for simple linear constraints on the coefficients of the linear combinations.

1. INTRODUCTION

In *component analysis* we assume that an observed vector of random variables $\underline{y}$ is a function of a number of unobserved random variables $\underline{x}$. Note that we use the "Dutch Convention" of underlining random variables [Hemelrijk, 1966]. Also note that we are guilty of abuse of language: there is no such thing in the real world as an "observed random variable". Calculations with random variables take place in the world of mathematics, although of course they can be used to motivate or justify actual calculations with floating point numbers.

The most common forms of component analysis can be exhibited in a small 3×3 table. The rows correspond with the nature of

Date: Tuesday 1st January, 2013 — 9h 53min — Typeset in LUCIDA BRIGHT.

the functional relation between $\underline{x}$ and $\underline{y}$. The columns specify additional assumed relationships between the different components in $\underline{x}$.

	unrestricted	orthogonal	independent
linear	LUCA	LOCA	LICA
polynomial	PUCA	POCA	PICA
nonlinear	NUCA	NOCA	NICA

Classic component analysis concentrates on linear relationships, i.e. LUCA, LOCA, and LICA. If we assume the $\underline{x}$ are joint multinormal, then LOCA and LICA are identical. LUCA and LOCA are basically the same, because we can assume without loss of generality that the components are orthogonal. The only constraint imposed by the models is the number of components p, which will generally be less than the number of variables m.

Both LICA and POCA must be considered computationally convenient approximations to more sensible forms of CA. If we give up the assumption of normality then there is not much reason to continue to assume either linearity or orthogonality. Without normality PICA is more natural, either as separate technique or as an approximation of NICA.

In this paper we will concentrate on LICA without assuming that the $\underline{x}$ are normal. Thus we assume there are m random variables $\underline{y}_j$ which are linear combinations of p independent components $\underline{x}_s$. . There is an $p \times m$ matrix of *loadings* B such that

$$\underline{y}_j = \sum_{s=1}^{p} b_{js}\underline{x}_s. \tag{1}$$

We do not assume anything about the relative size of p and m, either one of the two can be the larger one.

If Y is an $n \times m$ data matrix with rows that are, supposedly, independent realizations of $(\underline{\mathcal{Y}}_1, \cdots, \underline{\mathcal{Y}}_m)$, then the problem we study is how to recover B and/or X from the data.

1.1. **Constraints.** There can be various simple constraints on B in our framework. We can, for instance, fix some elements to zero, require all elements to be non-negative, or require that some elements are equal. In all of these cases the constraints on B are typically linear.

There is an enormous psychometric and statistical literature discussing the differences between *Principal Component Analysis* (PCA) and *Factor Analysis* (FA). In our framework, both PCA and FA are simply special cases of LUCA. In FA there are constraints on B. We have $p > m$ and $b_{js} = 0$ if $p+1 \leq s \leq m$ and $s \neq j$. In *confirmatory* FA there are additional constraints on B.

In some forms of CA we do not require independence or even orthogonality. In psychometric parlance [Kroonenberg, 2008] this leads to Tucker approximation with a non-diagonal core, and to a different classes of loss functions and algorithms. The resulting technique has been discussed recently by Morton and Lim [2009], also see Lim and Morton [2008], as *Principal Cumulant Component Analysis*.

1.2. **Loss.** In De Leeuw and Sorenson [2012] we follow De Leeuw [2004, 2008] and directly approximate the data matrix, using the least squares loss function $\mathbf{SSQ}(Y - F(X)B)$, with orthogonality constraints $X'X = I$. It is much more difficult, however, to directly impose independence constraints on X, because independence is defined, in principle at least, by an infinite number of constraints. As a consequence, in this paper we do not fit our model to the data directly, but, as in the most common implementations of component and factor analysis, we fit data to the *multivariate product moments*, more precisely the *multivariate cumulants*. An interesting

alternative that we do not explicitly consider in this paper would be to use the *multivariate empirical characteristic function*, in discussed in a related context by Van Montfort and De Leeuw [2004].

If we use moments or cumulants we integrate out the mode corresponding with observations or replications, and consequently we are left with a loss function that does not involve X any more. Loss is just a function of the component loadings and the cumulants of the latent variables. There is a large and confusing psychometric literature on factor scores that "cannot be computed but must be estimated". For our purposes it is sufficient to either generate a set of independent random numbers with the estimated cumulants, or to approximate the conditional expectation of the latent variables given the observed ones.

1.3. **Matrix and Array Approximation.** In much recent work on CA the multidimensional arrays of product moments and cumulants are referred to as *tensors*. There is something to be said for using tensor notation and terminology in this context, especially when dealing with multivariate cumulants. See, for example McCullagh [1984, 1987]; Peccati and Taqqu [2011]. For our purposes, however, tensors are overkill. We discuss algorithms to approximate matrices and arrays of statistics by parametrized matrices and arrays of simpler form.For computational purposes, especially with programs in R in mind, the arrays are simply vectors of double precision numbers with a dimension attribute.

A great deal of effort has also gone into developing specialized notation to deal with approximation multidimensional arrays. Again, we see this as largely irrelevant from the computational point of view. At some point the various notational constructs correspond with loops over vectors of indices, and the loops have to be taken care of, either internally by vectorized R operators or by explicit R programming. What we are basically suggesting is that the R program itself provides the appropriate notation in this case.

2. Linear Independent Component Analysis

Suppose $\mathcal{J}_r^m$ is the set of all $r-$tuples of integers between 1 and m. There are m^r such tuples, and the elements of a particular $J \in \mathcal{J}_r^m$ are not necessarily distinct. For any $J \in \mathcal{J}_r^m$ we have the multivariate cumulant

$$\kappa(J) = \sum_{s=1}^{p} \kappa_s^r \prod_{j \in J} b_{js}, \tag{2}$$

with κ_s^r the univariate cumulant of $\underline{x}_s$ of order r.

2.1. **Loss Function.**

We minimize the loss function

$$\sigma(B,K) = \sum_{r=1}^{\overline{r}} \alpha_r \sum_{J \in \mathcal{J}_r} \left(\hat{\kappa}(J) - \sum_{s=1}^{p} \kappa_{rs} \prod_{j \in J} b_{js} \right)^2. \tag{3}$$

Here the α_r are $\overline{r}$ given non-negative weights adding up to one, and the $\hat{\kappa}(J)$ are the observed or estimated multivariate cumulants. Thus we measure loss only for cumulants up to order $\overline{r}$. The univariate cumulants κ_{rs} are collected in a matrix K with $\overline{r}$ rows and p columns.

It is convenient to collect the $\hat{\kappa}(J)$ in an $r-$dimensional $m \times m \times \cdots \times m$ array. The multivariate cumulants define arrays of increasing dimensionality, and each of the arrays is approximated by a symmetric array with diagonal core structure. Using such a restrictive parametrization makes it possible to extract more dimensions than variables, especially with additional constraints on the loadings.

It is tempting to choose all α_r, except one, equal to zero. For instance, we can fit the third-order cross cumulants only. But we have to realize that this will not work for symmetric distributions, in which third-order cumulants are zero. It will not work very well if the distributions are approximately symmetric, because deviations from symmetry may just be sampling errors. This suggests it

is important to include the loss component with second order cumulants (variances and covariances) with a fairly large weight α_r, so that at least variances and covariances will be fitted well.

2.1.1. *Projected Loss.* Define

$$K(B) \triangleq \underset{K}{\textbf{argmin}}\, \sigma(B,K),$$

and

$$\sigma_\star(B) \triangleq \min_K \sigma(B,K) = \sigma(B,K(B)).$$

This is the projected loss, which now is a function only of B.

For the derivatives we find, by a simple special case of Danskin [1966],

$$\mathcal{D}\sigma_\star(B) = \mathcal{D}_1\sigma(B,K(B))$$

For row r of $K(B)$ we find

$$\kappa_r(B) = \left(C^{[r]}\right)^{-1} d^{[r]}$$

(where $C^{[r]}$ is the r-th elementwise or Hadamard power of $C = B'B$ and the vector $d^{[r]}$ has elements

$$d_s^{[r]} = \sum_{J\in\mathcal{J}_r} \hat{\kappa}(J) \prod_{j\in J} b_{js}$$

2.2. **Generating Data.** In section A of the appendix we give R code to generate data according to the LICA model. We use standard normal pseudo-random numbers to create X, but we power them to give them non-vanishing third and fourth order cumulants.

```
1 arti <- make_artificial (n = 1000, m = 9, p = 4,
     pow = 2)
2 cumu <- make_cumulants (arti $ y)
```

If $p < m$ the `make_artificial()` function creates a column-centered X such that $X'X = nI$ and B such that $B'B = I$. Thus $Y = XB'$ is column-centered and satisfies $\frac{1}{n}Y'Y = BB'$, i.e. the covariance matrix of Y is a projector of rank p.

If `err = TRUE` then `make_artificial()` creates an X with $p+m$ columns and an $m\times(p+m)$ matrix B of the form $B=[C\mid D]$, with $C'C=I$ and D diagonal. We will not use these data with common and unique components in this paper.

3. Algorithms

3.1. **Alternating Least Squares.** Consider the case in which we are fitting cumulants of order three only. The loss function is

$$\sigma(B)=\sum_{j=1}^m\sum_{\ell=1}^m\sum_{\nu=1}^m\left(\hat\kappa_{j\ell\nu}-\sum_{s=1}^p b_{js}b_{\ell s}b_{\nu s}\right)^2,\tag{4}$$

where we have absorbed the univariate cumulants κ_{3s} in B. In simple ALS we minimize instead

$$\sigma(A,B,F)=\sum_{j=1}^m\sum_{\ell=1}^m\sum_{\nu=1}^m\left(\hat\kappa_{j\ell\nu}-\sum_{s=1}^p a_{js}b_{\ell s}f_{\nu s}\right)^2,\tag{5}$$

i.e. we ignore the symmetry and minimize (5) by three linear least squares block relaxation steps. Define iteration k by

$$\tilde A^{(k)}=\mathop{\mathbf{argmin}}_A\sigma(A,B^{(k)},F^{(k)}),\tag{6a}$$

$$\tilde B^{(k)}=\mathop{\mathbf{argmin}}_B\sigma(\tilde A^{(k)},B,F^{(k)}),\tag{6b}$$

$$\tilde F^{(k)}=\mathop{\mathbf{argmin}}_C\sigma(\tilde A^{(k)},\tilde B^{(k)},F).\tag{6c}$$

This is the CANDECOMP algorithm of Carroll and Chang [1970], applied to the symmetric third-order array of cumulants. One would then expect the update $A^{(k+1)}=\tilde A^{(k)}$, $B^{(k+1)}=\tilde B(k)$, and $F^{(k+1)}=\tilde F^{(k)}$. This produces a monotone convergent algorithm with at least one subsequential limit $(A^\infty,B^\infty,F^\infty)$, but there is no guarantee that $A^\infty=B^\infty=F^\infty$. Some results in the context of INDSCAL are in Bennani Dosse and Ten Berge [2008]; Ten Berge et al. [2009].

Appendix B gives R code implementing the CANDECOMP algorithm for arrays of arbitrary order. The code allows for options in which

we can require B to be either non-negative or orthonormal, but for now we will not use these constraints.

Appendix E gives an alternative implementation of the CANDECOMP algorithm which use the `apl` package [De Leeuw and Yajima, 2011]. It is faster, because the basic functions in `apl` are written in C.

3.1.1. *Symmetrizing Iterations.* If the CANDECOMP algorithm from the previous section does not converge to a solution with all component matrices equal, then we can try to use (6) in combination with

$$A^{(k+1)} = B^{(k+1)} = F^{(k+1)} = (\tilde{A}^{(k)} + \tilde{B}^{(k)} + \tilde{F}^{(k)})/3. \tag{7}$$

Because (7) is a projection, this symmetrized CANDECOMP algorithm is still locally convergent to a fixed point, but monotone and global convergence is no longer guaranteed.

3.2. **Index Partitioning.** One solution to the symmetry dilemma of CANDECOMP is to partition the index set $\mathcal{J} = \{1, \cdots, m\}$ into three subsets J_1, J_2, and J_3 and to minimize

$$\sigma(A, B, F) = \sum_{j \in J_1} \sum_{\ell \in J_2} \sum_{\nu \in J_3} \left(\hat{\kappa}_{j\ell\nu} - \sum_{s=1}^{p} a_{js} b_{\ell s} f_{\nu s} \right)^2. \tag{8}$$

After convergence we concatenate $(A^\infty, B^\infty, F^\infty)$ to find the solution for B.

The problem with this approach is, of course, that there are many ways to partition $\mathcal{J}$, and each partition will lead to a different solution (unless there is perfect fit).

3.3. **Cyclic Coordinate Descent.** Suppose we replace b_{js} by $\tilde{b}_{js} = b_{js} + \theta \delta^{j\mu} \delta^{st}$, i.e. we leave B as is, but only change element (μ, t). In the case of fitting third-order cumulants we see from (4) that the optimal θ can be found by minimizing a polynomial of degree six, i.e. by finding the real roots of a polynomial of degree five.

For fourth-order cumulants the polynomial to minimize will be of degree eight, and so on.

We cycle through all coordinates, using efficient solvers for finding roots of polynomials. It is important for this algorithm to efficiently update the relevant quantities after computing each coordinate change.

The CCD method has the advantage that it can easily handle fitting second, third, and fourth order cumulants simultaneously. This is more difficult to do with the ALS methods.

3.4. **Two-step methods.** For the second order cumulants we have, assuming without loss of generality that the components have unit variance, $S = BB'$, where S is the covariance matrix. If the eigen decomposition of S is $S = K\Lambda^2 K'$, then it follows that $B = K\Lambda L'$, where L is an $p \times m$ orthonormal matrix (remember that we do not exclude the case $p > m$). The orthonormal matrix L can then be determined from the higher order cumulants. This is similar to the SIMSCAL method of De Leeuw and Pruzansky [1978].

Again, consider the case in which we fit third-order cumulants. We use the second order information to reduce the size of the problem and then use CANDECOMP with orthogonality restrictions to find the optimal rotation. There are some results on INDORT, i.e. orthogonal CANDECOMP in the INDSCAL context, in [Bennani Dosse et al., 2011].

It may be of some interest that this method can be used as a rotation technique for principal component analysis, using the higher order cumulants to rotate to maximum independence. Also note that this is equivalent to minimizing loss function (3) in the limiting case in which α_2 is infinitely large compared to the other α_r.

3.5. **General Purpose Optimization Methods.** R has a variety of add-on packages with optimization routines. A good overview,

with a manual for the package optimx, which wraps several unconstrained optimization methods, is in Nash and Varadhan [2011].

In our case it makes sense to optimize the projected loss function of 2.1.1

4. Example

4.1. **Alternating Least Squares.** We use the data generated in section 2.2.

```
> set.seed(12345)
> x<-matrix(rnorm(36),9,4)
> x3<-list(x,x,x)
> x4<-list(x,x,x,x)
> res3<-candecomp(cumu$c3, x3, verbose = TRUE)
Iteration:    1 Old Loss:  4890.577326 New Loss:
    11.035565
Iteration:    2 Old Loss:   11.035565 New Loss:
     5.819128
Iteration:    3 Old Loss:    5.819128 New Loss:
     1.632454
Iteration:    4 Old Loss:    1.632454 New Loss:
     0.554487
Iteration:    5 Old Loss:    0.554487 New Loss:
     0.020319
Iteration:    6 Old Loss:    0.020319 New Loss:
     0.012989
Iteration:    7 Old Loss:    0.012989 New Loss:
     0.012988
> res4<-candecomp(cumu$c4, x4, verbose = TRUE)
Iteration:    1 Old Loss:  64654.945865 New Loss:
    164.406934
Iteration:    2 Old Loss:  164.406934 New Loss:
   142.497797
```

```
Iteration:    3 Old Loss:  142.497797 New Loss:
   13.202225
Iteration:    4 Old Loss:   13.202225 New Loss:
      2.182829
Iteration:    5 Old Loss:    2.182829 New Loss:
      2.179922
Iteration:    6 Old Loss:    2.179922 New Loss:
      2.179921
Iteration:    7 Old Loss:    2.179921 New Loss:
      2.179921
```

If $\hat{B}$ is the estimate of B, and $\overline{B}$ is the true B used to compute the data, then we can find out how well we recover the solution by computing the matrix $(\hat{B}'\hat{B})^{-1}\hat{B}'\overline{B}(\overline{B}'\overline{B})^{-1}\overline{B}'\hat{B}$, which should be equal to the identity. In R this becomes simply

```
qr.solve(res3$x[[1]], arti$b) %*% qr.solve (arti$b
   , res3$x[[1]])
```

and similarly for the other two solutions in `res3` and the four solutions in `res4`. We do find identity matrices, which also implies that all solutions for B are identical and there is no need to symmetrize the CANDECOMP iterations.

If we fit the four-dimensional data using only two-dimensions we find

```
> set.seed(12345)
> x<-matrix(rnorm(18),9,2)
> x3<-list(x,x,x)
> res3<-candecomp(cumu$c3, x3, verbose = TRUE)
Iteration:    1 Old Loss:  484.866406 New Loss:
   16.830996
Iteration:    2 Old Loss:   16.830996 New Loss:
   14.562874
```

```
Iteration:    3 Old Loss:   14.562874 New Loss:
   14.558700
Iteration:    4 Old Loss:   14.558700 New Loss:
   14.558699
Iteration:    5 Old Loss:   14.558699 New Loss:
   14.558699
> qr.solve(res3$x[[1]], res3$x[[2]]) %*% qr.solve
   (res3$x[[2]], res3$x[[1]])
             [,1]         [,2]
[1,] 1.000000e+00 2.309441e-16
[2,] 2.824184e-15 1.000000e+00
> qr.solve(res3$x[[1]], res3$x[[3]]) %*% qr.solve
   (res3$x[[3]], res3$x[[1]])
             [,1]         [,2]
[1,] 1.000000e+00 3.742154e-16
[2,] 4.228399e-15 1.000000e+00
> qr.solve(res3$x[[2]], res3$x[[3]]) %*% qr.solve
   (res3$x[[3]], res3$x[[2]])
              [,1]         [,2]
[1,]  1.000000e+00 2.674211e-18
[2,] -2.769133e-17 1.000000e+00
```

Again there is no need to symmetrize.

4.2. **Two-step Method.** We first reduce the dimension of the array of third-order cumulants by using the nonzero eigenvalues and corresponding eigenvectors of the second-order cumulants (i.e. the covariance matrix, which is a projector of rank 4 in this case).

```
red <- array (0, c(4, 4, 4))
ev <- eigen (cumu$c2)
a <-  ev $ vectors [, 1 : 4]
b <- sqrt (ev $ values [1 : 4])
for (i in 1 : 4) for (j in 1 : 4) for (k in 1 : 4) {
```

```
    for (p in 1 : 9) for (q in 1 : 9) for (r in 1 :9)
        {
    aa <- a [p, i] * a [q, j] * a [r, k]
    bb <- 1 / (b [i] * b [j] * b [k])
    red [i, j, k] <- red [i, j, k] + cumu$c3 [p, q, r]
        * aa * bb
    }
}
```

We then apply CANDECOMP to the reduced array to find the optimal rotation.

```
> set.seed (12345)
> x <- qr.Q (qr (matrix (rnorm (16), 4, 4)))
> aa <- candecomp(red, list (x, x, x), verbose = TRUE,
    ortho = rep (TRUE, 3))
Iteration:    1 Old Loss:   21.903768 New Loss:   11
   .094943
Iteration:    2 Old Loss:   11.094943 New Loss:   10
   .854350
Iteration:    3 Old Loss:   10.854350 New Loss:   10
   .854349
Iteration:    4 Old Loss:   10.854349 New Loss:   10
   .854349
> bb <- arti$b
> bc <- a %*% diag (b) %*% (aa $ x [[1]])
> qr.solve (bb, bc)
            [,1]         [,2]        [,3]         [,4]
[1,] -0.04561276  0.013822892  0.99834432  0.032202851
[2,] -0.02935202  0.001229333 -0.03358220  0.999004096
[3,]  0.01633056  0.999780665 -0.01305830 -0.001189439
[4,]  0.99839433 -0.015685580  0.04483677  0.030860627
```

This shows that the B we find is essentially the true B, up to a permutation of the dimensions.

4.3. **Cyclic Coordinate Descent.** The code in Appendix C can be used to fit second, third, and/or fourth order cumulants, simultaneously or separately. The algorithm can probably be improved a great deal. What we currently do is compute the loss function at $2R+1$ equally spaced points, where R is the highest order cumulant considered. We then fit a polynomial of degree $2R$ through these points, differentiate it, find the roots of the derivative, and use the real root corresponding with the minimum, using the `polynom` package [Venables et al., 2009].

We give three runs, the first fits third-order cumulants, the second both third-order and fourth-order, and the third just forth-order.

```
> ftf<-candecca(cumu$c2, cumu$c3, cumu$c4, 4, mode = c
    (f2 = FALSE, f3 = TRUE, f4 = FALSE), verbose = TRUE
    )
Iteration:    1 Old Loss:   18.210058 New Loss:    1
    .600939
Iteration:    2 Old Loss:    1.600939 New Loss:    0
    .080718
Iteration:    3 Old Loss:    0.080718 New Loss:    0
    .016456
Iteration:    4 Old Loss:    0.016456 New Loss:    0
    .013149
Iteration:    5 Old Loss:    0.013149 New Loss:    0
    .012995
Iteration:    6 Old Loss:    0.012995 New Loss:    0
    .012988
Iteration:    7 Old Loss:    0.012988 New Loss:    0
    .012988
> ftt<-candecca(cumu$c2, cumu$c3, cumu$c4, 4, mode = c
    (f2 = FALSE, f3 = TRUE, f4 = TRUE), verbose = TRUE)
Iteration:    1 Old Loss:  337.030406 New Loss:   83
    .236050
Iteration:    2 Old Loss:   83.236050 New Loss:   22
    .740455
```

```
Iteration:    3 Old Loss:   22.740455 New Loss:    2.742149
Iteration:    4 Old Loss:    2.742149 New Loss:    2.392112
Iteration:    5 Old Loss:    2.392112 New Loss:    2.354895
Iteration:    6 Old Loss:    2.354895 New Loss:    2.349831
Iteration:    7 Old Loss:    2.349831 New Loss:    2.349162
Iteration:    8 Old Loss:    2.349162 New Loss:    2.349079
Iteration:    9 Old Loss:    2.349079 New Loss:    2.349069
Iteration:   10 Old Loss:    2.349069 New Loss:    2.349068
Iteration:   11 Old Loss:    2.349068 New Loss:    2.349068
> fft<-candecca(cumu$c2, cumu$c3, cumu$c4, 4, mode = c(f2 = FALSE, f3 = FALSE, f4 = TRUE), verbose = TRUE)
Iteration:    1 Old Loss:  318.820348 New Loss:   77.379064
Iteration:    2 Old Loss:   77.379064 New Loss:   49.099478
Iteration:    3 Old Loss:   49.099478 New Loss:    3.016047
Iteration:    4 Old Loss:    3.016047 New Loss:    2.239244
Iteration:    5 Old Loss:    2.239244 New Loss:    2.186204
Iteration:    6 Old Loss:    2.186204 New Loss:    2.180520
Iteration:    7 Old Loss:    2.180520 New Loss:    2.179975
```

```
Iteration:    8 Old Loss:    2.179975 New Loss:    2
   .179927
Iteration:    9 Old Loss:    2.179927 New Loss:    2
   .179922
Iteration:   10 Old Loss:    2.179922 New Loss:    2
   .179921
```

All three runs reproduce B pretty much exactly.

```
> qr.solve(b,ftf$x)%*%qr.solve(ftf$x,b)
              [,1]          [,2]          [,3]
                         [,4]
[1,]  1.000000e+00  1.890758e-12 -1.288589e-11 3
   .071839e-11
[2,]  1.890802e-12  1.000000e+00 -1.480419e-11 1
   .207829e-11
[3,] -1.288599e-11 -1.480441e-11  1.000000e+00 1
   .866793e-10
[4,]  3.071830e-11  1.207784e-11  1.866794e-10 1
   .000000e+00
> qr.solve(b,ftt$x)%*%qr.solve(ftt$x,b)
              [,1]          [,2]          [,3]
                          [,4]
[1,]  1.000000e+00 -7.736867e-14  2.608330e-14  7
   .002177e-13
[2,] -7.738601e-14  1.000000e+00 -3.627409e-13  9
   .881256e-13
[3,]  2.588554e-14 -3.629668e-13  1.000000e+00 -4
   .430525e-13
[4,]  7.000858e-13  9.879007e-13 -4.428942e-13  1
   .000000e+00
> qr.solve(b,fft$x)%*%qr.solve(fft$x,b)
              [,1]          [,2]          [,3]
                         [,4]
[1,]  1.000000e+00 -4.657247e-13 -9.005852e-13 5
   .767553e-12
```

```
16 [2,] -4.657628e-13  1.000000e+00  4.755735e-13 5
       .600477e-12
17 [3,] -9.005071e-13  4.754697e-13  1.000000e+00 7
       .356665e-12
18 [4,]  5.767477e-12  5.600118e-12  7.356799e-12 1
       .000000e+00
```

4.4. **General Optimization Methods.** We use `optimx` with parameter `method="spg"`. The method finds the same loss function values as the CCD methods in the previous section. Code is in Appendix D.

4.5. **Timing.** We give the elapsed time given by `system.time()` for the various optimization methods applied to our three different fits.

Cumulants	CANDECOMP	CCD	OPTIMX
3	0.047	1.392	9.159
3+4		10.458	35.588
4	0.078	7.585	28.679

References

M. Bennani Dosse and J.M.F Ten Berge. The Assumption of Proportional Components when CANDECOMP is Applied to Symmetric Matrices in the Context of INDSCAL. *Psychometrika*, 73:303–307, 2008.

M. Bennani Dosse, J.M.F Ten Berge, and J.N. Tendeiro. Some New Results on Orthogonally Constrained CANDECOMP. *Journal of Classification*, 28:144–155, 2011.

J.D. Carroll and J.J. Chang. Analysis of Individual Differences in Multidimensional Scaling Via an N-way Generalization of Eckart-Young Decomposition. *Psychometrika*, pages 283–319, 1970.

J.M. Danskin. The Theory of Max-Min, with Applications. *SIAM Journal on Applied Mathematics*, 14:641–664, 1966.

J. De Leeuw. Least Squares Optimal Scaling of Partially Observed Linear Systems. In K. van Montfort, J. Oud, and A. Satorra, editors, *Recent Developments on Structural Equation Models*, chapter 7. Kluwer Academic Publishers, Dordrecht, Netherlands, 2004. URL `http://preprints.stat.ucla.edu/360/lserr.pdf`.

J. De Leeuw. Factor Analysis as Matrix Decomposition. Unpublished, 2008. URL `http://www.stat.ucla.edu/~deleeuw/janspubs/2008/notes/deleeuw_U_08g.pdf`.

J. De Leeuw and S. Pruzansky. A New Computational Method to fit the Weighted Euclidean Distance Model. *Psychometrika*, 43: 479–490, 1978.

J. De Leeuw and K. K. Sorenson. Least Squares Orthogonal Polynomial Component Analysis in R. Unpublished, 2012.

J. De Leeuw and M. Yajima. *apl: APL Array Manipulation Functions*, 2011. URL `http://R-Forge.R-project.org/projects/apl/`. R package version 0.01-01/r59.

J. Hemelrijk. Underlining Random Variables. *Statistica Neerlandica*, 20:1–7, 1966.

P.M. Kroonenberg. *Applied Multiway Data Analysis*. Wiley Series in Probability and Statistics. Wiley-Interscience, New York, N.Y., 2008.

L.-H. Lim and J. Morton. Cumulant Component Analysis: A Simultaneous Generalization of PCA and ICA. Unpublished, December 2008. URL http://math.stanford.edu/~jason/lim-morton-abstract.pdf.

P. McCullagh. Tensor Notation and Cumulants of Polynomials. *Biometrika*, 71:461–476, 1984.

P. McCullagh. *Tensor Methods in Statistics*. Chapman & Hall, Boca Raton, Florida, 1987.

J. Morton and L.-H. Lim. Principal Cumulant Component Analysis. Unpublished, 2009. URL http://galton.uchicago.edu/~lekheng/work/pcca.pdf.

J.C. Nash and R. Varadhan. Unifying Optimization Algorithms to Aid Software System Users: optimx for R. *Journal of Statistical Software*, 43(9):1–13, 2011. URL http://www.jstatsoft.org/v43/i09/paper.

G. Peccati and M.S. Taqqu. *Wiener Chaos: Moments, Cumulants and Diagrams*. Springer, Milan, Italy, 2011.

J.M.F Ten Berge, A. Stegeman, and M. Bennani Dosse. The Carroll and Chang Conjecture of Equal INDSCAL Components when CANDECOMP/PARAFAC Gives Perfect Fit. *Linear Algebra and Its Applications*, 430:818–829, 2009.

K. Van Montfort and J. De Leeuw. Factor Analysis of Non-Normal Variables by Fitting Characteristic Functions. Preprint Series 397, UCLA Department of Statistics, Los Angeles, CA, 2004. URL http://www.stat.ucla.edu/~deleeuw/janspubs/2004/reports/vanmontfort_deleeuw_R_04.pdf.

B. Venables, K. Hornik, and M. Maechler. *polynom: A Collection of Functions to Implement a Class for Univariate Polynomial Manipulations*, 2009. URL http://CRAN.R-project.org/package=polynom. R package version 1.3-5. S original by Bill Venables,

packages for R by Kurt Hornik and Martin Maechler.

Appendix A. Computing Cumulants

Code Segment 1 AAA

```
make_cumulants<-function(y) {
    n <- nrow (y)
    m <- ncol (y)
    mm <- 1 : m
    nn <- 1 : n
    r1 <- colSums (y) / n
    r2 <- crossprod (y) / n
    r3 <- array (0, c (m, m, m))
    r4 <- array (0, c (m, m, m, m))
    for (i in nn) {
        r3 <- r3 + outer (outer (y [i, ], y [i, ]), y [i, ])
        r4 <- r4 + outer (outer (y [i, ], y [i, ]), outer (y [i, ], y [i,
             ]))
    }
    r3<-r3 / n
    r4<-r4 / n
    c2<-r2 - outer (r1, r1)
    c3<-r3
    for (i in mm) for (j in mm) for (k in mm) {
        s3 <- r3 [i, j, k]
        s21 <- r2 [i, j] * r1 [k] + r2 [i, k] * r1 [j] + r2 [j, k] * r1 [
            i]
        s111 <- r1 [i] * r1 [j] * r1 [k]
        c3 [i, j, k] <- s3 - s21 + 2 * s111
    }
    c4<-r4
    for (i in mm) for (j in mm) for (k in mm) for (l in mm) {
        s4 <- r4 [i, j, k, l]
        s31 <- r3 [i, j, k] * r1 [l] + r3 [i, j, l] * r1 [k] + r3 [i, k,
            l] * r1 [j] + r3 [j, k, l] * r1 [i]
        s22 <- r2 [i, j] * r2 [k, l] + r2 [i, k] * r2 [j, l] + r2 [j, k]
            * r2 [i, l]
        s211 <- r2 [i, j] * r1 [k] * r1 [l] + r2 [i, k] * r1 [j] * r1 [l]
             + r2 [i, l] * r1 [k] * r1 [j] + r2 [j, k] * r1 [i] * r1 [l]
            + r2 [j, l] * r1 [i] * r1 [k] + r2 [k, l] * r1 [i] * r1 [j]
        s1111 <- r1 [i] * r1 [j] * r1 [k] * r1 [l]
        c4 [i, j, k, l] <- s4 - s31 - s22 + 2 * s211 - 6 * s1111
    }
    return (list(c2 = c2, c3 = c3, c4 = c4))
}

multivariate_raw_moments <- function (x, p) {
    n <- nrow (x)
    m <- ncol (x)
    y <- array (0, rep (m + 1, p))
    for (i in 1:n) {
        xi <- c (1, x[i,])
        z <- xi
        if (p > 1) {
            for (s in 2:p) {
                z <- outer (z, xi)
            }
```

Appendix B. Multiway in R

```
1
2  require("quadprog")
3  require("nnls")
4
5  candecomp <- function(a, x, xopt = rep (0,length (x)),
6          ortho = rep (FALSE, length (dim (a))), ispos =
               FALSE,
7          itmax = 1000, eps = 1e-6, verbose = FALSE) {
8      ndam <- dim (a)
9      nard <- length (ndam)
10     ndim <- ncol (x [[1]])
11     for (k in 1 : nard) {
12         if (ortho [k]) {
13             x [[k]] <- procrustus (x [[k]])
14         }
15     }
16     cp <- lapply (x, crossprod)
17     oloss <- candeValue (a, x) $ loss
18     itel <- 1
19     repeat {
20         for (k in 1:nard) {
21             cc <- arrHadamard (cp [-k])
22             for (p in 1 : ndim) {
23                 y <- lapply (x [-k], function(z) z [,p
                       ])
24                 b <- arrOuter (y)
25                 x [[k]] [,p] <- apply(a, k, function(z
                       ) sum (z * b))
26                 }
27             xx <- x [[k]]
28             if (ortho[k]) {
29                 x [[k]] <- procrustus (xx)
```

```
            }
            else {
                if (!ispos) {
                    x [[k]] <- t (solve (cc, t (xx)))
                }
                else {
                    x [[k]] <- posProj (xx, cc)
                }
            }
            cp [[k]] <- crossprod (x [[k]])
            }
            cval <- candeValue (a,x)
            nloss <- cval $ loss
            ahat <- cval $ ahat
        if (verbose)
            cat ("Iteration: ", formatC (itel, digits
               = 3, width = 3),
                 "Old Loss: ", formatC (oloss, digits
                    = 6, width = 10, format = "f"),
                 "New Loss: ", formatC (nloss, digits
                    = 6, width = 10, format = "f"),
                 "\n")
        if ((itel == itmax) | ((oloss - nloss) < eps))
            break ()
        itel <- itel + 1
        oloss <- nloss
        }
    return (list (x = x, ahat = ahat, loss = nloss))
    }

candeValue<-function(a,x) {
    ndim<-ncol(x[[1]])
    ahat<-array(0,dim(a))
    for (p in 1:ndim) {
        y<-lapply(x,function(z) z[,p])
        b<-arrOuter(y)
```

```
        ahat<-ahat+b
        }
    loss<-sum((a-ahat)^2)
    return(list(ahat=ahat,loss=loss))
    }

tucker<-function(a,x,ident=rep(FALSE,length(x)),ortho=
   rep(FALSE,length(x)),ispos=rep(FALSE,length(x)),
   isposb=FALSE,itmax=1000,eps=1e-6,verbose=FALSE) {
    ndam<-dim(a)
    for (k in 1:nard) if (ident[k]) x[[k]]<-diag(ndam[
       k])
    ndbm<-sapply(x,function(z) ncol(z))
    nard<-length(ndam)
    rard<-rev(1:nard)
    x<-lapply(x,procrustus)
    xx<-arrKronecker(x)
    aa<-as.vector(aperm(a,rard))
    if (!isposb) bb<-lsfit(xx,aa,intercept=FALSE)
        else bb<-nnls(xx,aa)
    b<-aperm(array(bb,rev(ndbm)),rard)
    ahat<-aperm(array(drop(xx%*%bb),rev(ndam)),rard)
    oloss<-sum((a-ahat)^2)
    itel<-1
    repeat {
        for (k in 1:nard) {
            if (!ident[k]) {
                z<-crossprod(flatten(a,k),arrKronecker
                   (x[-k]))%*%flatten(b,k)
                x[[k]]<-procrustus(z)
                }
            }
        xx<-arrKronecker(x)
        aa<-as.vector(aperm(a,rard))
        if (!isposb) bb<-lsfit(xx,aa,intercept=FALSE)
            else bb<-nnls(xx,aa)
```

```
94          b<-aperm(array(bb,rev(ndbm)),rard)
95          ahat<-aperm(array(drop(xx%*%bb),rev(ndam)),
               rev(1:nard))
96          nloss<-sum((a-ahat)^2)
97          if (verbose)
98              cat("Iteration: ",formatC(itel,digits=3,
                   width=3),
99                  "Old Loss: ",formatC(oloss,digits=6,
                       width=10,format="f"),
100                 "New Loss: ",formatC(nloss,digits=6,
                       width=10,format="f"),
101                 "\n")
102         if ((itel == itmax) | ((oloss - nloss) < eps))
                break()
103         itel<-itel+1; oloss<-nloss
104         }
105     return(list(x=x,b=b,ahat=ahat,loss=nloss))
106     }
107
108 # Hadamard product of a list of arrays
109
110 arrHadamard<-function(c,f="*") {
111     nmat<-length(c)
112     fun<-match.fun(f)
113     if (nmat == 0) stop("empty argument in arrHadamard
           ")
114     res<-c[[1]]
115     if (nmat == 1) return(res)
116     for (i in 2:nmat) res<-fun(res,c[[i]])
117     return(res)
118     }
119
120 # outer product of a list of arrays
121
122 arrOuter<-function(x,fun="*") {
123     nmat<-length(x)
```

```
    if (nmat == 0) stop("empty argument in arrOuter")
    res<-x[[1]]
    if (length(x) == 1) return(res)
    for (i in 2:nmat) res<-outer(res,x[[i]],fun)
    return(res)
    }

# kronecker product of a list of arrays

arrKronecker<-function(x,fun="*") {
    nmat<-length(x)
    if (nmat == 0) stop("empty argument in
       arrKronecker")
    res<-x[[1]]
    if (length(x) == 1) return(res)
    for (i in 2:nmat) res<-kronecker(res,x[[i]],fun)
    return(res)
    }

flatten<-function(a,k) {
    nard<-rev(1:(length(dim(a))-1))
    apply(a,k,function(z) as.vector(aperm(z,nard)))
}

procrustus<-function(x) {
    res<-svd(x)
    return(tcrossprod(res$u,res$v))
}

posProj<-function(x,c) {
n<-nrow(x); p<-ncol(x); ip<-diag(p)
u<-matrix(0,n,p)
for (i in 1:n) {
    y<-x[i,]
    u[i,]<-solve.QP(c,y,ip)$solution
    }
```

```
159 return(u)
160 }
```

APPENDIX C. CCD IN R

```
require("polynom")

source("cumulant.R")
arti <- make_artificial (n = 1000, m = 9, p = 4, pow =
    2)
cumu <- make_cumulants (arti $ y)

candecca <- function (cum2, cum3, cum4, p, mode = c (
   f2 = FALSE, f3 = TRUE, f4 = FALSE), itmax = 1000,
   eps = 1e-6, verbose = FALSE) {
    m <- nrow (cum2)
    e <- eigen (cum2)
    x <- e $ vectors [, 1: p] %*% diag (sqrt (e $
       values [1 : p]))
    if (mode ["f4"]) {
        th <- c (-.4, -.3, -.2, -.1, 0, .1, .2, .3, .4
           )
    }
    if ((!mode ["f4"]) & (mode ["f3"])) {
        th <- c (-.3, -.2, -.1, 0, .1, .2, .3)
    }
    if ((!mode ["f4"]) & (!mode ["f3"])) {
        th <- c (-.2, -.1, 0, .1, .2)
    }
    ot <- outer (th, 0 : (length (th) - 1), "^")
    w <- th
    itel <- 1
    kur2 <- rep (1, p)
    kur3 <- rep (1, p)
    kur4 <- rep (1, p)
    oloss <- 0
    if (mode ["f2"]) {
```

```
28          oloss <- oloss + candeValue (cum2, kur2, list
                (x, x)) $ loss
29      }
30      if (mode ["f3"]) {
31          oloss <- oloss + candeValue (cum3, kur3, list
                (x, x, x)) $ loss
32      }
33      if (mode ["f4"]) {
34          oloss <- oloss + candeValue (cum4, kur4, list
                (x, x, x, x)) $ loss
35      }
36      repeat {
37          nloss <- Inf
38          cc <- crossprod (x)
39          if (mode ["f2"]) {
40              kur2 <- solve (cc ^ 2, candeValue (cum2,
                    kur2, list (x, x)) $ d)
41          }
42          if (mode ["f3"]) {
43              kur3 <- solve (cc ^ 3, candeValue (cum3,
                    kur3, list (x, x, x)) $ d)
44          }
45          if (mode ["f4"]) {
46              kur4 <- solve (cc ^ 4, candeValue (cum4,
                    kur4, list (x, x, x, x)) $ d)
47          }
48          for (j in 1 : m) {
49              for (s in 1 : p) {
50                  for (i in 1 : length (th)) {
51                      xt <- x
52                      xt [j, s] <- x [j, s] + th [i]
53                      w [i] <- 0
54                      if (mode ["f2"]) {
55                          w [i] <- w [i] + candeValue (
                                cum2, kur2, list (xt, xt))
                                $ loss
```

```
56                      }
57                      if (mode ["f3"]) {
58                          w [i] <- w [i] + candeValue (
                                cum3, kur3, list (xt, xt,
                                xt)) $ loss
59                      }
60                      if (mode ["f4"]) {
61                          w [i] <- w [i] + candeValue (
                                cum4, kur4, list (xt, xt,
                                xt, xt)) $ loss
62                      }
63                  }
64                  cf <- solve (ot, w)
65                  pf <- polynomial (cf)
66                  rt <- solve (deriv (pf))
67                  rr <- Re (rt [which (abs (Im (rt)) < 1
                        e-10)])
68                  pr <- predict (pf, rr)
69                  nloss <- min (nloss, min (pr))
70                  tp <- rr [which.min (pr)]
71                  x [j, s] <- x [j, s] + tp
72              }
73          }
74          if (verbose) {
75              cat ("Iteration: ", formatC (itel, digits
                    = 3, width = 3),
76                  "Old Loss: ", formatC (oloss, digits
                        = 6, width = 10, format = "f"),
77                  "New Loss: ", formatC (nloss, digits
                        = 6, width = 10, format = "f"),
78                  "\n")
79              }
80          if (((oloss - nloss) < eps) || (itel == itmax)
                ) break()
81          oloss <- nloss
82          itel <- itel + 1
```

```
    }
    return (list(x = x, kur = list (kur2, kur3, kur4),
        loss = nloss))
}

candeValue <- function(a, kur, x) {
    ndim <- ncol (x [[1]])
    ahat <- array (0, dim (a))
    d <- rep (0, ndim)
    for (p in 1 : ndim) {
        y <- lapply (x, function (z) z[,p])
        b <- arrOuter (y)
        d [p] <- sum (a * b)
        ahat <- ahat + kur [p] * b
        }
    loss <- sum ((a - ahat) ^ 2)
    return (list (ahat = ahat, d = d, loss = loss))
    }

arrOuter <- function (x, fun = "*") {
    nmat <- length(x)
    if (nmat == 0) {
        stop ("empty argument in arrOuter")
    }
    res <- x [[1]]
    if (length (x) == 1) {
        return (res)
    }
    for (i in 2 : nmat) {
        res <- outer (res, x [[i]], fun)
    }
    return (res)
    }
```

Appendix D. General Optimization Techniques

```
require("optimx")

source("cumulant.R")
arti <- make_artificial (n = 1000, m = 9, p = 4, pow =
    2)
cumu <- make_cumulants (arti $ y)

cangenopt <- function (cum2, cum3, cum4, p, mode = c (
   f2 = FALSE, f3 = TRUE, f4 = FALSE), itmax = 1000,
   eps = 1e-6, verbose = FALSE) {
    m <- nrow (cum2)
    e <- eigen (cum2)
    x <- e $ vectors [, 1: p] %*% diag (sqrt (e $
       values [1 : p]))
    itel <- 1
    kur2 <- rep (1, p)
    kur3 <- rep (1, p)
    kur4 <- rep (1, p)
    ansout <<- optimx (as.vector (x), proj_loss,
       method = "spg",
            mode = mode, cum = list (cum2, cum3, cum4)
               , kur = list (kur2, kur3, kur4))
    nloss <- ansout [[2]] [[1]]
    x <- matrix (ansout [[1]] [[1]], m, p)
    return (list(x = x, loss = nloss))
}

candeValue <- function(a, kur, x) {
    ndim <- ncol (x [[1]])
    ahat <- array (0, dim (a))
    d <- rep (0, ndim)
    for (p in 1 : ndim) {
```

```
27          y <- lapply (x, function (z) z[,p])
28          b <- arrOuter (y)
29          d [p] <- sum (a * b)
30          ahat <- ahat + kur [p] * b
31          }
32      loss <- sum ((a - ahat) ^ 2)
33      return (list (ahat = ahat, d = d, loss = loss))
34      }
35
36  arrOuter <- function (x, fun = "*") {
37      nmat <- length(x)
38      if (nmat == 0) {
39          stop ("empty argument in arrOuter")
40      }
41      res <- x [[1]]
42      if (length (x) == 1) {
43          return (res)
44      }
45      for (i in 2 : nmat) {
46          res <- outer (res, x [[i]], fun)
47      }
48      return (res)
49      }
50
51  proj_loss <- function (x, mode, cum, kur) {
52      loss <- 0
53      m <- nrow (cum [[1]])
54      x <- matrix (x, m, length (x) / m)
55      cc <- crossprod (x)
56      if (mode ["f2"]) {
57          kur[[1]] <- solve (cc ^ 2, candeValue (cum
                [[1]], kur[[1]], list (x, x)) $ d)
58          loss <- loss + candeValue (cum[[1]], kur[[1]],
                list (x, x)) $ loss
59      }
60      if (mode ["f3"]) {
```

```
61          kur[[2]] <- solve (cc ^ 3, candeValue (cum
                [[2]], kur[[2]], list (x, x, x)) $ d)
62          loss <- loss + candeValue (cum[[2]], kur[[2]],
                list (x, x, x)) $ loss
63      }
64      if (mode ["f4"]) {
65          kur[[3]] <- solve (cc ^ 4, candeValue (cum
                [[3]], kur[[3]], list (x, x, x, x)) $ d)
66          loss <- loss + candeValue (cum[[3]], kur[[3]],
                list (x, x, x, x)) $ loss
67      }
68      return (loss)
69  }
```

APPENDIX E. CANDECOMP USING THE APL PACKAGE

```
require("apl")

candecomp <- function (a, ind, ndim, itmax = 100, eps
   = 1e-6, intercept = FALSE, verbose = TRUE) {
    nn <- dim (a)
    nd <- length (dim (a))
    ni <- sapply (ind, length)
    li <- lapply (ni, function (i) 1:i)
    vv <- list ()
    cc <- list ()
    for (i in 1:nd) {
        vv[[i]] <- matrix (runif (ni[i] * ndim), ni[i
           ], ndim)
        cc[[i]] <- crossprod (vv[[i]])
    }
    itel <- 1
    oloss <- Inf
    adat <- aplSelect (a, ind)
    repeat {
        nloss <- mkLoss (ni, adat, vv)
        if (!verbose) {
            cat("Iteration: ",formatC(itel,digits=3,
               width=3),
                "nloss    : ",formatC(nloss,digits=6,
                   width=10,format="f"),
                "\n")
        }
        for (i in 1:nd) {
            f <- matrix (0, ndim, ni[i])
            for (s in 1 : ndim) {
                oo <- 1
                for (k in 1:nd) {
```

```
29                      if (k == i) next ()
30                      oo <- drop (outer (oo, vv[[k]][, s
                            ]))
31                  }
32                  for (k in 1 : ni[i]) {
33                      kk <- li
34                      kk[[i]] <- k
35                      f[s, k] <- sum (drop (aplSelect (
                            adat, kk)) * oo)
36                  }
37              }
38              cp <- matrix (1, ndim, ndim)
39              for (k in 1:nd) {
40                  if (k == i) next()
41                  cp <- cp * cc[[k]]
42              }
43              vv[[i]] <- t (solve (cp, f))
44              cc[[i]] <- crossprod (vv[[i]])
45              if (verbose) {
46                  nloss <- mkLoss (ni, adat, vv)
47                  cat("Iteration:    ",formatC(itel,
                        digits=3,width=3),
48                      "After Block : ",formatC(i,digits
                            =3,width=3),
49                       formatC(nloss,digits=6,width=10,
                            format="f"),
50                      "\n")
51              }
52          }
53          if (((oloss - nloss) < eps) || (itel == itmax)
                ) break()
54              oloss <- nloss; itel<-itel+1
55      }
56      return (list(v = vv, itel = itel, loss = nloss))
57  }
58
```

```
mkLoss <- function (ni, adat, vv) {
    nloss <- 0
    nsiz <- prod (ni)
    nd <- length (ni)
    for (i in 1:nsiz) {
        indi <- aplEncode (i, ni)
        vprd <- rep (1, ndim)
        for (s in 1:nd)
            vprd <- vprd * vv[[s]][indi[s],]
        nloss <- nloss + (sum (vprd) - adat[aplDecode(
           indi,ni)])^2
    }
    return (nloss)
}
```

Appendix F. TUCKER using the APL package

```
require("apl")

tucker <- function (a, ind, ndim, itmax = 100, eps = 1
   e-6, intercept = FALSE, verbose = TRUE) {
    nn <- dim (a)
    nd <- length (dim (a))
    ni <- sapply (ind, length)
    li <- lapply (ni, function (i) 1:i)
    vv <- list ()
    for (i in 1:nd) {
        vv[[i]] <- qr.Q (qr (matrix (runif (ni[i] *
           ndim), ni[i], ndim)))
    }
    # mkCore
    itel <- 1
    oloss <- Inf
    adat <- aplSelect (a, ind)
    repeat {
        nloss <- mkLoss (ni, adat, vv)
        if (!verbose) {
            cat("Iteration: ",formatC(itel,digits=3,
               width=3),
                "nloss    : ",formatC(nloss,digits=6,
                   width=10,format="f"),
                "\n")
        }
        for (i in 1:nd) {
            f <- matrix (0, ndim, ni[i])
            for (s in 1 : ndim) {
                oo <- 1
                for (k in 1:nd) {
                    if (k == i) next ()
```

```
                    oo <- drop (outer (oo, vv[[k]][, s
                        ]))
                }
                for (k in 1 : ni[i]) {
                    kk <- li
                    kk[[i]] <- k
                    f[s, k] <- sum (drop (aplSelect (
                        adat, kk)) * oo)
                }
            }
            cp <- matrix (1, ndim, ndim)
            for (k in 1:nd) {
                if (k == i) next()
                cp <- cp * cc[[k]]
            }
            vv[[i]] <- t (solve (cp, f))
            cc[[i]] <- crossprod (vv[[i]])
            if (verbose) {
                nloss <- mkLoss (ni, adat, vv)
                cat("Iteration:    ",formatC(itel,
                    digits=3,width=3),
                    "After Block : ",formatC(i,digits
                        =3,width=3),
                     formatC(nloss,digits=6,width=10,
                         format="f"),
                    "\n")
            }
        }
        if (((oloss - nloss) < eps) || (itel == itmax)
            ) break()
            oloss <- nloss; itel<-itel+1
    }
    return (list(v = vv, itel = itel, loss = nloss))
}

mkLoss <- function (ni, adat, vv) {
```

```
    nloss <- 0
    nsiz <- prod (ni)
    nd <- length (ni)
    for (i in 1:nsiz) {
        indi <- aplEncode (i, ni)
        vprd <- rep (1, ndim)
        for (s in 1:nd)
            vprd <- vprd * vv[[s]][indi[s],]
        nloss <- nloss + (sum (vprd) - adat[aplDecode(
           indi,ni)])^2
    }
    return (nloss)
}
```

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

E-mail address, Irina Kukuyeva: ikukuyeva@gmail.com

URL, Irina Kukuyeva: https://sites.google.com/site/ikukuyeva/