Package ‘multiway’

May 19, 2017

Type Package
Title Component Models for Multi-Way Data
Version 1.0-3
Date 2017-05-17
Author Nathaniel E. Helwig <helwig@umn.edu>
Maintainer Nathaniel E. Helwig <helwig@umn.edu>
Depends parallel, quadprog
Description Fits multi-way component models via alternating least squares algorithms with optional constraints: orthogonal, non-negative, unimodal, monotonic, periodic, smooth, or structure. Fit models include Individual Differences Scaling, Parallel Factor Analysis (1 and 2), Simultaneous Component Analysis, and Tucker Factor Analysis.
License GPL (>= 2)
NeedsCompilation no
Repository CRAN
Date/Publication 2017-05-19 06:20:13 UTC

R topics documented:

multiway-package ......................................................... 2
congru ................................................................. 3
const.control .......................................................... 5
corcondia ............................................................... 6
fitted ................................................................. 8
fnls ................................................................. 9
indscal ............................................................... 10
krprod ............................................................. 14
mpinv ............................................................... 15
ncenter ............................................................. 16
nscale ............................................................. 18
parafac ............................................................ 20
parafac2 ............................................................ 24
print ............................................................. 30
Description

Fits multi-way component models via alternating least squares algorithms with optional constraints: orthogonal, non-negative, unimodal, monotonic, periodic, smooth, or structure. Fit models include Individual Differences Scaling, Parallel Factor Analysis (1 and 2), Simultaneous Component Analysis, and Tucker Factor Analysis.

Details

indscal fits the Individual Differences Scaling model. parafac fits the 3-way and 4-way Parallel Factor Analysis-1 model. parafac2 fits the 3-way and 4-way Parallel Factor Analysis-2 model. sca fits the four different Simultaneous Component Analysis models. tucker fits the 3-way and 4-way Tucker Factor Analysis model.

Author(s)

Nathaniel E. Helwig <helwig@umn.edu>
Maintainer: Nathaniel E. Helwig <helwig@umn.edu>

References


**Examples**

```r
congruHxL y 
```

`# See examples for indscal, parafac, parafac2, sca, and tucker`

**Description**

Calculates Tucker’s congruence coefficient (uncentered correlation) between x and y if these are vectors. If x and y are matrices then the congruence between the columns of x and y are computed.

**Usage**

```r
congru(x, y = NULL)```

Arguments

x  Numeric vector, matrix or data frame.
y  NULL (default) or a vector, matrix or data frame with compatible dimensions to x. The default is equivalent to y = x (but more efficient).

Details

Tucker's congruence coefficient is defined as

$$r = \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2 \sum_{i=1}^{n} y_i^2}}$$

where $x_i$ and $y_i$ denote the $i$-th elements of $x$ and $y$.

Value

Returns a scalar or matrix with congruence coefficient(s).

Note

If $x$ is a vector, you must also enter $y$.

Author(s)

Nathaniel E. Helwig <helwig@umn.edu>

References


Examples

```
set.seed(1)
A <- rnorm(100)
B <- rnorm(100)
C <- A + 5
D <- A * (-0.5)
congru(A, B)
congru(A, C)
congru(A, D)
```

```
set.seed(1)
A <- cbind(rnorm(20), rnorm(20))
B <- cbind(A[, 1] * -0.5, rnorm(20))
```
const.control

Description

Auxiliary function for controlling the const argument of the parafac and parafac2 functions. Applicable when using constraints 3 (unimodal), 4 (monotonic), 5 (periodic), and/or 6 (smooth).

Usage

const.control(const, df = NULL, degree = NULL, nonneg = NULL)

Arguments

const Constraints for each mode. Vector of length 3 or 4 with entries: 0 = unconstrained (default), 1 = orthogonal, 2 = non-negative, 3 = unimodal, 4 = monotonic, 5 = periodic, 6 = smooth.
df Integer vector of length 3 or 4 giving the degrees of freedom to use for the spline basis in each mode. Can also input a single number giving the common degrees of freedom to use for each mode. Defaults to 7 degrees of freedom for each applicable mode.
degree Integer vector of length 3 or 4 giving the polynomial degree to use for the spline basis in each mode. Can also input a single number giving the common polynomial degree to use for each mode. Defaults to degree 3 (cubic) polynomials for each applicable mode.
nonneg Logical vector of length 3 or 4 indicating whether the weights in each mode should be constrained to be non-negative. Can also input a single logical giving the common non-negativity constraints to use for each mode. Defaults to FALSE for each applicable mode.

Details

The parafac and parafac2 functions pass the input control to this function to determine the fitting options when using constraints 3-6.

Value

Returns a list with elements: const, df, degree, and nonneg.

Author(s)

Nathaniel E. Helwig <helwig@umn.edu>
### Examples

```
# create random data array with Parafac structure  
set.seed(4)  
mydim <- c(30,10,8,10)  
nf <- 4  
aseq <- seq(-3,3,length=mydim[1])  
Amat <- cbind(dnorm(aseq), dchisq(aseq+3.1, df=3),  
              dt(aseq-2, df=4), dgamma(aseq+3.1, shape=3, rate=1))  
Bmat <- svd(matrix(runif(mydim[2]*nf),mydim[2],nf))$u  
Cmat <- matrix(runif(mydim[3]*nf),mydim[3],nf)  
Dmat <- matrix(runif(mydim[4]*nf),mydim[4],nf)  
Xmat <- array(tcrossprod(Amat,krprod(Dmat,krprod(Cmat,Bmat))),dim=mydim)  
Emat <- array(rnorm(prod(mydim)),dim=mydim)  
X <- Xmat + Emat  

# fit Parafac model (unimodal A, orthogonal B, non-negative C, non-negative D)  
pfac <- parafac(X,nfac=nf,nstart=1,const=c(3,1,2,2))  
pfac  

# same as before, but add some options to the unimodality contraints...  
# fewer knots (df=5), quadratic splines (degree=2), and enforce non-negativity  
cc <- const.control(c(3,1,2,2), df=5, degree=2, nonneg=TRUE)  
pfac <- parafac(X,nfac=nf,nstart=1,const=c(3,1,2,2),control=cc)  
pfac
```

---

**corcondia**

### Core Consistency Diagnostic

**Description**

Calculates Bro and Kiers's core consistency diagnostic (CORCONDIA) for a fit `parafac` or `parafacR` model. For Parafac2, the diagnostic is calculated after transforming the data by `pfac$Phi`, and then power scaling.

**Usage**

```
corcondia(X, object, divisor=c("nfac","core"))
```

**Arguments**

- `X`  
  Three-way data array with `dim=c(I,J,K)` or four-way data array with `dim=c(I,J,K,L)`.  
  Can also input a list of two-way or three-way arrays (for Parafac2).

- `object`  
  Object of class "parafac" (output from `parafac`) or class "parafac2" (output from `parafac2`).

- `divisor`  
  Divide by number of factors (default) or core sum of squares.
Details

The core consistency diagnostic is defined as

\[ \text{CORCONDIA} = 100 \times \left( 1 - \frac{\sum (G-S)^2}{\text{divisor}} \right) \]

where \( G \) is the least squares estimate of the Tucker core array, \( S \) is a super-diagonal core array, and \( \text{divisor} \) is the sum of squares of either \( S \) ("nfac") or \( G \) ("core"). A value of 100 indicates a perfect multilinear structure, and smaller values indicate greater violations of multilinear structure.

Value

Returns CORCONDIA value.

Author(s)

Nathaniel E. Helwig <helwig@umn.edu>

References


Examples

```
# create random data array with Parafac structure
set.seed(3)
mydim <- c(50,20,5)
nf <- 2
Amat <- matrix(rnorm(mydim[1]*nf),mydim[1],nf)
Bmat <- matrix(runif(mydim[2]*nf),mydim[2],nf)
Cmat <- matrix(runif(mydim[3]*nf),mydim[3],nf)
Xmat <- array(tcrossprod(Amat,krprod(Cmat,Bmat)),dim=mydim)
Emat <- array(rnorm(prod(mydim)),dim=mydim)
Emat <- nscale(Emat,0,sumsq(Xmat))  # SNR=1
X <- Xmat + Emat

# fit Parafac model (1-4 factors)
pfac1 <- parafac(X,nfac=1,nstart=1)
pfac2 <- parafac(X,nfac=2,nstart=1)
pfac3 <- parafac(X,nfac=3,nstart=1)
pfac4 <- parafac(X,nfac=4,nstart=1)

# check corcondia
corcondia(X, pfac1)
corcondia(X, pfac2)
corcondia(X, pfac3)
corcondia(X, pfac4)
```
fitted

**Extract Multi-Way Fitted Values**

**Description**

Calculates fitted array (or list of arrays) from a multiway object.

**Usage**

```r
## S3 method for class 'indscal'
fitted(object,...)
## S3 method for class 'parafac'
fitted(object,...)
## S3 method for class 'parafac2'
fitted(object,simplify=TRUE,...)
## S3 method for class 'sca'
fitted(object,...)
## S3 method for class 'tucker'
fitted(object,...)
```

**Arguments**

- `object` Object of class "indscal" (output from `indscal`), class "parafac" (output from `parafac`), class "parafac2" (output from `parafac2`), class "sca" (output from `sca`), or class "tucker" (output from `tucker`).
- `simplify` For "parafac2", setting `simplify=FALSE` will always return a list of fitted arrays. Default of `simplify=TRUE` returns a fitted array if all levels of the nesting mode have the same number of observations (and a list of fitted arrays otherwise).
- `...` Ignored.

**Details**

See `indscal`, `parafac`, `parafac2`, `sca`, and `tucker` for more details.

**Value**

- "indscal" objects: 3-way array.
- "parafac" objects: 3-way or 4-way array.
- "parafac2" objects: 3-way or 4-way array (if possible and `simplify=TRUE`); otherwise list of 2-way or 3-way arrays.
- "sca" objects: list of 2-way arrays.
- "tucker" objects: 3-way or 4-way array.

**Author(s)**

Nathaniel E. Helwig <helwig@umn.edu>
**Examples**

```r
### see examples for indscal, parafac, parafac2, sca, and tucker
```

---

### fnnls

**Fast Non-Negative Least Squares**

**Description**

Finds the vector $b$ minimizing

$$\text{sum}(y - X\cdot b)^2$$

subject to $b[j] \geq 0$ for all $j$.

**Usage**

```r
fnnls(XtX,Xty,ntol=NULL)
```

**Arguments**

- **xtX**: Crossproduct matrix \texttt{crossprod(X)} of dimension p-by-p.
- **xty**: Crossproduct vector \texttt{crossprod(X,y)} of length p-by-1.
- **ntol**: Tolerance for non-negativity.

**Value**

The vector $b$ such that $b[j] \geq 0$ for all $j$.

**Note**

Default non-negativity tolerance: \texttt{ntol=10^*(.Machine$double.eps)*max(colSums(abs(XtX)))*p}.

**Author(s)**

Nathaniel E. Helwig <helwig@umn.edu>

**References**

Examples

```
########### EXAMPLE 1 ###########
X <- matrix(1:100, 50, 2)
y <- matrix(101:150, 50, 1)
beta <- solve(crossprod(X)) %*% crossprod(X, y)
beta
beta <- fnnls(crossprod(X), crossprod(X, y))

########### EXAMPLE 2 ###########
X <- cbind(-(1:50), 51:100)
y <- matrix(101:150, 50, 1)
beta <- solve(crossprod(X)) %*% crossprod(X, y)
beta
beta <- fnnls(crossprod(X), crossprod(X, y))

########### EXAMPLE 3 ###########
X <- matrix(rnorm(400), 100, 4)
btrue <- c(1, 2, 0, 7)
y <- X %*% btrue + rnorm(100)
fnnls(crossprod(X), crossprod(X, y))

########### EXAMPLE 4 ###########
X <- matrix(rnorm(2000), 100, 2)
btrue <- runif(20)
y <- X %*% btrue + rnorm(100)
beta <- fnnls(crossprod(X), crossprod(X, y))
crossprod(btrue - beta) / 20
```

---

**Description**

Given a 3-way array \(X = \text{array}(x, \text{dim}=c(J, J, K))\) with \(X[, , k]\) denoting the \(k\)-th subject's dissimilarity matrix rating \(J\) objects, the INDSCAL model can be written as

\[
Z[i, j, k] = \sum B[i, r] \times B[j, r] \times C[k, r] + E[i, j, k]
\]

where \(Z\) is the array of scalar products obtained from \(X\), \(B = \text{matrix}(b, J, R)\) are the object weights, \(C = \text{matrix}(c, K, R)\) are the non-negative subject weights, and \(E = \text{array}(e, \text{dim}=c(J, J, K))\) is the 3-way residual array. The summation is for \(r = \text{seq}(1, R)\).

Weight matrices are estimated using an alternating least squares algorithm with optional constraints.
Usage

```
indscal(X, nfac, nstart=10, const=NULL, maxit=500,
        type=c(“dissimilarity”, “similarity”),
        ctol=1e-4, parallel=FALSE, cl=NULL,
        output=c(“best”, “all”))
```

Arguments

- **X**: Three-way data array with \(dim=(\cdot,J,k)\) where \(X[\cdot,J,k]\) is dissimilarity matrix. Can also input a list of (dis)similarity matrices or objects output by `dist`.
- **nfac**: Number of factors.
- **nstart**: Number of random starts.
- **const**: Constraints for Modes B and C. See Note.
- **maxit**: Maximum number of iterations.
- **type**: Character indicating if \(X\) contains dissimilarity data (default) or similarity data.
- **ctol**: Convergence tolerance.
- **parallel**: Logical indicating if `parLapply` should be used. See Examples.
- **cl**: Cluster created by `makeCluster`. Only used when parallel=TRUE.
- **output**: Output the best solution (default) or output all nstart solutions.

Value

If `output=“best”`, returns an object of class "indscal" with the following elements:

- **B**: Mode B weight matrix.
- **C**: Mode C weight matrix.
- **SSE**: Sum of Squared Errors.
- **Rsq**: R-squared value.
- **GCV**: Generalized Cross-Validation.
- **edf**: Effective degrees of freedom.
- **iter**: Number of iterations.
- **cflag**: Convergence flag.
- **const**: See argument const.

Otherwise returns a list of length nstart where each element is an object of class "indscal".

Warnings

The ALS algorithm can perform poorly if the number of factors nfac is set too large.
Default is unconstrained ALS update, which may produce negative (invalid) Mode C weights. Use `const=c(0,2)` to force non-negativity via `fnnls`. 
Note

Default use is 10 random strats (nstart=10) with 500 maximum iterations of the ALS algorithm for each start (maxit=500) using a convergence tolerance of 1e-4 (ctol=1e-4). The algorithm is determined to have converged once the change in R^2 is less than or equal to ctol.

Input const should be a two element integer vector giving constraints for Modes B and C. There are four possible options:

- const=c(0,0) Unconstrained update for Modes B and C
- const=c(0,2) Unconstrained Mode B with non-negative Mode C
- const=c(1,0) Orthogonal Mode B with unconstrained Mode C
- const=c(1,2) Orthogonal Mode B with non-negative Mode C

Default is unconstrained update for all modes, i.e., const=c(0,0).

Output cflag gives convergence information: cflag=0 if ALS algorithm converged normally, cflag=1 if maximum iteration limit was reached before convergence, and cflag=2 if ALS algorithm terminated abnormally due to problem with non-negativity constraints.

Author(s)

Nathaniel E. Helwig <helwig@umn.edu>

References


Examples

```
# create random data array with INDSCAL structure
set.seed(3)
mydim <- c(50,5,10)
nf <- 2
X <- array(0,dim=c(rep(mydim[2],2),mydim[3]))
for(k in 1:mydim[3]) {
  X[,k] <- as.matrix(dist(t(matrix(rnorm(prod(mydim[1:2])),mydim[1],mydim[2]))))
}

# fit INDSCAL model (unconstrained)
mod <- indscal(X,nfac=nf,nstart=1)
```
```r
indscal

# check solution
Xhat <- fitted(imod)
sum((array(apply(X,3,ed2sp),dim=dim(X))-Xhat)^2)
imod$SSE

# reorder and resign factors
imod$B[1:4,]
imod <- reorder(imod, 2:1)
imod$B[1:4,]
imod <- resign(imod, newsign=c(1,-1))
imod$B[1:4,]
sum((array(apply(X,3,ed2sp),dim=dim(X))-Xhat)^2)
imod$SSE

# rescale factors
colSums(imod$B^2)
colSums(imod$C^2)
imod <- rescale(imod, mode="C")
colSums(imod$B^2)
colSums(imod$C^2)
sum((array(apply(X,3,ed2sp),dim=dim(X))-Xhat)^2)
imod$SSE

############ list example ############

# create random data array with INDSCAL structure
set.seed(4)
mydim <- c(100,8,20)
f <- 3
X <- vector("list",mydim[3])
for(k in 1:mydim[3]) {
  X[[k]] <- dist(t(matrix(rnorm(prod(mydim[1:2])),mydim[1],mydim[2])))
}

# fit INDSCAL model (orthogonal B, non-negative C)
imod <- indscal(X,nf=nf,nstart=1,const=c(1,2))
imod

# check solution
Xhat <- fitted(imod)
sum((array(unlist(lapply(X,ed2sp)),dim=mydim[c(2,2,3)])-Xhat)^2)
imod$SSE
crossprod(imod$B)

## Not run:

############ parallel computation ############

# create random data array with INDSCAL structure
set.seed(3)
mydim <- c(50,5,10)
nf <- 2
X <- array(0,dim=c(rep(mydim[2],2),mydim[3]))
for(k in 1:mydim[3]) {
  X[,k] <- as.matrix(dist(t(matrix(rnorm(prod(mydim[1:2])),mydim[1],mydim[2]))))
}

# fit INDSCAL model (10 random starts -- sequential computation)
set.seed(1)
system.time({imod <- indscal(x,nfac=nf)})
imod

# fit INDSCAL model (10 random starts -- parallel computation)
set.seed(1)
cl <- makeCluster(detectCores())
ce <- clusterEvalQ(cl,library(multiway))
system.time({imod <- indscal(x,nfac=nf,parallel=TRUE,cl=cl)})
imod
stopCluster(cl)

## End(Not run)

---

krprod

**Khatri-Rao Product**

**Description**

Given $X$ (n-by-p) and $Y$ (m-by-p), the Khatri-Rao product $Z = \text{krprod}(X,Y)$ is defined as

$$Z[,j] = \text{kronecker}(X[,j],Y[,j])$$

which is the mn-by-p matrix containing Kronecker products of corresponding columns of $X$ and $Y$.

**Usage**

```
krprod(X,Y)
```

**Arguments**

- **X** Matrix of order n-by-p.
- **Y** Matrix of order m-by-p.

**Value**

The mn-by-p matrix of columnwise Kronecker products.
Note

X and Y must have the same number of columns.

Author(s)

Nathaniel E. Helwig <helwig@umn.edu>

Examples

```
###########  EXAMPLE 1  ###########
X <- matrix(1:4,2)
Y <- matrix(1:4,2,2)
krprod(X,Y)
```

```
###########  EXAMPLE 2  ###########
X <- matrix(1:2,4,2)
Y <- matrix(1:4,2,2)
krprod(X,Y)
```

```
###########  EXAMPLE 3  ###########
X <- matrix(1:2,4,2,byrow=TRUE)
Y <- matrix(1:4,2,2)
krprod(X,Y)
```

---

### mpinv

**Moore-Penrose Pseudoinverse**

Description

Calculates the Moore-Penrose pseudoinverse of the input matrix using a truncated singular value decomposition.

Usage

```
mpinv(X, tol=.Machine$double.eps)
```

Arguments

- **X**: Real-valued matrix.
- **tol**: Stability tolerance for singular values.

Value

Returns pseudoinverse of X.
Note

Basically returns \( Y \cos \% \cos \text{diag}(1/Y\cos d) \cos \% \cos \top(Y\cos u) \) where \( Y = \text{svd}(X) \).

Author(s)

Nathaniel E. Helwig <helwig@umn.edu>

References


Examples

```r
set.seed(1)
X <- matrix(rnorm(2000),100,20)
Xi <- mpinv(X)
sum( (X - X \cos \% \cos X \cos \%)^2 )
sum( (Xi - Xi \cos \% \cos X \cos \%)^2 )
issymmetric(X \cos \% \cos Xi)
issymmetric(Xi \cos \% \cos X)
```

ncenter | Center n-th Dimension of Array

Description

Fiber-center across the levels of the specified mode. Can input 2-way, 3-way, and 4-way arrays, or input a list containing array elements.

With \( X \) a matrix (I-by-J) there are two options:

- **mode=1**: \( x[i,j] - \text{mean}(x[,j]) \)
- **mode=2**: \( x[i,j] - \text{mean}(x[i,]) \)

With \( X \) a 3-way array (I-by-J-by-K) there are three options:

- **mode=1**: \( x[i,j,k] - \text{mean}(x[,j,k]) \)
- **mode=2**: \( x[i,j,k] - \text{mean}(x[i,,k]) \)
- **mode=3**: \( x[i,j,k] - \text{mean}(x[i,j,]) \)
With X a 4-way array (I-by-J-by-K-by-L) there are four options:

- mode=1: \( x[i,j,k,l] - \text{mean}(x[,,k,1]) \)
- mode=2: \( x[i,j,k,l] - \text{mean}(x[i,,k,1]) \)
- mode=3: \( x[i,j,k,l] - \text{mean}(x[i,j,,1]) \)
- mode=4: \( x[i,j,k,l] - \text{mean}(x[i,j,k,]) \)

Usage

\( \text{ncenter}(X, \text{mode}=1) \)

Arguments

- \( X \), Array (2-way, 3-way, or 4-way) or a list containing array elements.
- \( \text{mode} \), Mode to center across.

Value

Returns centered version of \( X \).

Note

When entering a list with array elements, each element must (a) be an array (2-way, 3-way, or 4-way), and (b) have the same dimension (i.e., array size).

Author(s)

Nathaniel E. Helwig <helwig@umn.edu>

Examples

```
# EXAMPLE 1
X <- matrix(rnorm(2000),100,20)
Xc <- ncenter(X)  # center across rows
sum(colSums(Xc))
Xc <- ncenter(Xc,mode=2)  # recenter across columns
sum(colSums(Xc))
sum(rowSums(Xc))

# EXAMPLE 2
X <- array(rnorm(20000),dim=c(100,20,10))
Xc <- ncenter(X,mode=2)  # center across columns
sum(rowSums(Xc))

# EXAMPLE 3
X <- array(rnorm(100000),dim=c(100,20,10,5))
Xc <- ncenter(X,mode=4)  # center across 4-th mode
sum(rowSums(Xc))
```
### EXAMPLE 4

```r
X <- replicate(5, array(rnorm(20000), dim=c(100,20,10)), simplify=FALSE)
Xc <- ncenter(X)
sum(colSums(Xc[[1]]))
```

---

**nscale**

Scale n-th Dimension of Array

---

**Description**

Slab-scale within each level of the specified mode. Can input 2-way, 3-way, and 4-way arrays, or input a list containing array elements (see Note).

With X a matrix (I-by-J) there are two options:

- **mode=1**: \( x[i,j]*\sqrt{\text{ssnew/sumsq}(x[i,])} \)
- **mode=2**: \( x[i,j]*\sqrt{\text{ssnew/sumsq}(x[,j])} \)

With X a 3-way array (I-by-J-by-K) there are three options:

- **mode=1**: \( x[i,j,k]*\sqrt{\text{ssnew/sumsq}(x[i,,])} \)
- **mode=2**: \( x[i,j,k]*\sqrt{\text{ssnew/sumsq}(x[,j,])} \)
- **mode=3**: \( x[i,j,k]*\sqrt{\text{ssnew/sumsq}(x[,,k])} \)

With X a 4-way array (I-by-J-by-K-by-L) there are four options:

- **mode=1**: \( x[i,j,k,l]*\sqrt{\text{ssnew/sumsq}(x[i,,,])} \)
- **mode=2**: \( x[i,j,k,l]*\sqrt{\text{ssnew/sumsq}(x[,j,,])} \)
- **mode=3**: \( x[i,j,k,l]*\sqrt{\text{ssnew/sumsq}(x[,,k,])} \)
- **mode=4**: \( x[i,j,k,l]*\sqrt{\text{ssnew/sumsq}(x[,,,l])} \)

**Usage**

```r
nscale(X, mode=1, ssnew=1)
nrescale(X, mode=1, ssnew=1)
```

**Arguments**

- **X**
  - Array (2-way, 3-way, or 4-way) or a list containing array elements.
- **mode**
  - Mode to scale within (set mode=0 to scale across all modes).
- **ssnew**
  - Desired sum-of-squares for each level of scaled mode.
Value

Returns scaled version of \( X \).

Note

When entering a list with array elements, each element must be a 2-way or 3-way array. The list elements are treated as the 3rd mode (for list of 2-way arrays) or the 4th mode (for list of 3-way arrays) in the formulas provided in the Description.

Author(s)

Nathaniel E. Helwig <helwig@umn.edu>

Examples

```
####### EXAMPLE 1 #######
X <- matrix(rnorm(2000),100,20)
Xr <- nscale(X,mode=2)  # scale columns to ssnew=1
colSums(Xr^2)
Xr <- nscale(X,mode=2,ssnew=2)  # scale columns to ssnew=2
colSums(Xr^2)
```

```
####### EXAMPLE 2 #######
Xold <- X <- matrix(rnorm(4000),20,20)
iter <- 0
chk <- 1
# iterative rescaling of modes 1 and 2
while(iter<500 & chk>=10^-9){
    Xr <- nscale(Xold,mode=1)
    Xr <- nscale(Xr,mode=2)
    chk <- sum((Xold-Xr)^2)
    Xold <- Xr
    iter <- iter + 1
}
iter
rowSums(Xr^2)
colSums(Xr^2)
```

```
####### EXAMPLE 3 #######
X <- array(rnorm(200000),dim=c(100,20,10))
Xc <- nscale(X,mode=2)  # scale within columns
rowSums(aperm(Xc,perm=c(2,1,3))^2)
```

```
####### EXAMPLE 4 #######
X <- array(rnorm(100000),dim=c(100,20,10,5))
Xc <- nscale(X,mode=4)  # scale across 4-th mode
rowSums(aperm(Xc,perm=c(4,1,2,3))^2)
```
##### Example 5

```
X <- replicate(5, array(rnorm(20000), dim=c(100, 20)), simplify=FALSE)
Xc <- nscale(X, ssnew=(20*10*5))  # mean square of 1
rowSums(sapply(Xc, function(x) rowSums(x^2))) / (20*10*5)
```

---

**parafac**

*Parallel Factor Analysis-I*

---

**Description**

Given a 3-way array \( X = \text{array}(x, \text{dim}=c(I, J, K)) \), the 3-way Parafac model can be written as

\[
X[i,j,k] = \sum A[i,r]*B[j,r]*C[k,r] + E[i,j,k]
\]

where \( A = \text{matrix}(a, I, R) \) are the Mode A (first mode) weights, \( B = \text{matrix}(b, J, R) \) are the Mode B (second mode) weights, \( C = \text{matrix}(c, K, R) \) are the Mode C (third mode) weights, and \( E = \text{array}(e, \text{dim}=c(I, J, K)) \) is the 3-way residual array. The summation is for \( r = \text{seq}(1,R) \).

Given a 4-way array \( X = \text{array}(x, \text{dim}=c(I, J, K, L)) \), the 4-way Parafac model can be written as

\[
X[i,j,k,l] = \sum A[i,r]*B[j,r]*C[k,r]*D[l,r] + E[i,j,k,l]
\]

where \( D = \text{matrix}(d, L, R) \) are the Mode D (fourth mode) weights, \( E = \text{array}(e, \text{dim}=c(I, J, K, L)) \) is the 4-way residual array, and the other terms can be interpreted as previously described.

Weight matrices are estimated using an alternating least squares algorithm with optional constraints.

**Usage**

```
parafac(X, nfac, nstart=10, const=NULL, control=NULL,
        Bfixed=NULL, Cfixed=NULL, Dfixed=NULL,
        Bstart=NULL, Cstart=NULL, Dstart=NULL,
        Bstruc=NULL, Cstruc=NULL, Dstruc=NULL,
        maxit=500, ctol=1e-4, parallel=FALSE,
        cl=NULL, output=c("best", "all"))
```

**Arguments**

- **X**: Three-way data array with \( \text{dim}=c(I, J, K) \) or four-way data array with \( \text{dim}=c(I, J, K, L) \).
- **nfac**: Number of factors.
- **nstart**: Number of random starts.
- **const**: Constraints for each mode. Vector of length 3 or 4 with entries: 0 = unconstrained (default), 1 = orthogonal, 2 = non-negative, 3 = unimodal, 4 = monotonic, 5 = periodic, 6 = smooth. Use control argument to adjust options for constraints 3-6.
List of parameters controlling options for constraints 3-6. This is passed to `const.control`, which describes the available options.

- **Bfixed**: Fixed Mode B weights. Only used to fit model with fixed Mode B weights.
- **Cfixed**: Fixed Mode C weights. Only used to fit model with fixed Mode C weights.
- **Dfixed**: Fixed Mode D weights. Only used to fit model with fixed Mode D weights.
- **Bstart**: Starting Mode B weights for ALS algorithm. Default uses random weights.
- **Cstart**: Starting Mode C weights for ALS algorithm. Default uses random weights.
- **Dstart**: Starting Mode D weights for ALS algorithm. Default uses random weights.
- **Bstruc**: Structure constraints for Mode B weights. Default uses unstructured weights.
- **Cstruc**: Structure constraints for Mode C weights. Default uses unstructured weights.
- **Dstruc**: Structure constraints for Mode D weights. Default uses unstructured weights.
- **maxit**: Maximum number of iterations.
- **ctol**: Convergence tolerance (R² change).
- **parallel**: Logical indicating if `parLapply` should be used. See Examples.
- **cl**: Cluster created by `makeCluster`. Only used when `parallel=TRUE`.
- **output**: Output the best solution (default) or output all `nstart` solutions.

**Value**

If `output="best"`, returns an object of class "parafac" with the following elements:

- **A**: Mode A weight matrix.
- **B**: Mode B weight matrix.
- **C**: Mode C weight matrix.
- **D**: Mode D weight matrix.
- **SSE**: Sum of Squared Errors.
- **Rsq**: R-squared value.
- **GCV**: Generalized Cross-Validation.
- **edf**: Effective degrees of freedom.
- **iter**: Number of iterations.
- **cflag**: Convergence flag.
- **const**: See argument const.
- **control**: See argument control.
- **fixed**: Logical vector indicating whether 'fixed' weights were used for each mode.
- **struc**: Logical vector indicating whether 'struc' constraints were used for each mode.

Otherwise returns a list of length `nstart` where each element is an object of class "parafac".
Warnings

The ALS algorithm can perform poorly if the number of factors \( n_{\text{fac}} \) is set too large.
Non-negativity constraints can be sensitive to local optima.
Non-negativity constraints can result in slower performance.
Structure constraints for override constraints in \( \text{const} \) input.

Note

Default use is 10 random starts (\( n_{\text{start}}=10 \)) with 500 maximum iterations of the ALS algorithm for each start (\( \text{maxit}=500 \)) using a convergence tolerance of 1e-4 (\( \text{ctol}=1e-4 \)). The algorithm is determined to have converged once the change in \( R^2 \) is less than or equal to \( \text{ctol} \).
Output \( \text{cflag} \) gives convergence information: \( \text{cflag}=0 \) if ALS algorithm converged normally, \( \text{cflag}=1 \) if maximum iteration limit was reached before convergence, and \( \text{cflag}=2 \) if ALS algorithm terminated abnormally due to a problem with the constraints.
Constraints 3 (unimodality) and 4 (monotonicity) are implemented using I-splines, and constraints 5 (periodicity) and 6 (smoothness) are implemented using M-splines (see Ramsay, 1988).

Author(s)

Nathaniel E. Helwig <helwig@umn.edu>

References


Examples

```
# create random data array with Parafac structure
set.seed(3)
mydim <- c(50,20,5)
nf <- 3
Amat <- matrix(rnorm(mydim[1]*nf),mydim[1],nf)
Bmat <- matrix(runif(mydim[2]*nf),mydim[2],nf)
Cmat <- matrix(runif(mydim[3]*nf),mydim[3],nf)
Xmat <- array(tcrossprod(Amat,krprod(Cmat,Bmat)),dim=mydim)
Emat <- array(rnorm(prod(mydim)),dim=mydim)
Emat <- nscale(Emat,0,sumsq(Xmat))  # SNR=1
```
parafac

X <- Xmat + Emat

# fit Parafac model (unconstrained)
pfac <- parafac(X,nfac=nf,nstart=1)
pfac

# fit Parafac model (non-negativity on Modes B and C)
pfacNN <- parafac(X,nfac=nf,nstart=1, const=c(0,2,2))
pfacNN

# check solution
Xhat <- fitted(pfac)
sum((Xmat-Xhat)^2)/prod(mydim)

# reorder and resign factors
pfacB %in% 1:4,]
pfac <- reorder(pfac, c(3,1,2))
pfacB %in% 1:4,]
pfac <- resign(pfac, mode="B")
pfacB %in% 1:4,]
Xhat <- fitted(pfac)
sum((Xmat-Xhat)^2)/prod(mydim)

# rescale factors
colSums(pfacB^2)
colSums(pfacC^2)
pfac <- rescale(pfac, mode="C", absorb="B")
colSums(pfacB^2)
colSums(pfacC^2)
Xhat <- fitted(pfac)
sum((Xmat-Xhat)^2)/prod(mydim)

############# 4-way example #############

# create random data array with Parafac structure
set.seed(4)
mydim <- c(30,10,8,10)
nf <- 4
aseq <- seq(-3,3,length=mydim[1])
Amat <- cbind(dnorm(aseq), dchisq(aseq+3,1, df=3),
dt(aseq-2, df=4), dgamma(aseq+3.1, shape=3, rate=1))
Bmat <- svd(matrix(runif(mydim[2]*nf),mydim[2],nf))$u
Cmat <- matrix(runif(mydim[3]*nf),mydim[3],nf)
Dmat <- matrix(runif(mydim[4]*nf),mydim[4],nf)
Xmat <- array(tcrossprod(Amat,krprod(Dmat,krprod(Cmat,Bmat))),dim=mydim)
Emat <- array(rnorm(prod(mydim)),dim=mydim)
Emat <- nscale(Emat,0,sumsq(Xmat))  # SNR=1
X <- Xmat + Emat

# fit Parafac model (unimodal A, orthogonal B, non-negative C, non-negative D)
pfac <- parafac(X,nfac=nf,nstart=1, const=c(3,1,2,2))
pfac
# check solution
Xhat <- fitted(pfac)
sum((Xmat-Xhat)^2)/prod(mydim)
congru(Amat, pfac$A)
crossprod(pfac$B)
pfac$C

### Not run:

########## parallel computation ##########

# create random data array with Parafac structure
set.seed(3)
mydim <- c(50,20,5)
nf <- 3
Amat <- matrix(rnorm(mydim[1]*nf),mydim[1],nf)
Bmat <- matrix(runif(mydim[2]*nf),mydim[2],nf)
Cmat <- matrix(runif(mydim[3]*nf),mydim[3],nf)
Xmat <- array(tcrossprod(Amat,krprod(Cmat,Bmat)),dim=mydim)
Emat <- array(rnorm(prod(mydim)),dim=mydim)
Emat <- nscale(Emat,0,sumsq(Xmat))  # SNR=1
X <- Xmat + Emat

# fit Parafac model (10 random starts -- sequential computation)
set.seed(1)
system.time((pfac <- parafac(X,nfac=nf)))
pfac

# fit Parafac model (10 random starts -- parallel computation)
set.seed(1)
cl <- makeCluster(detectCores())
ce <- clusterEvalQ(cl,library(multiway))
system.time((pfac <- parafac(X,nfac=nf,parallel=TRUE,cl=cl)))
pfac
stopCluster(cl)

### End(Not run)

description

Given a list of matrices \( X[[k]] = \text{matrix}(x_k, I[k], J) \) for \( k = \text{seq}(1,K) \), the 3-way Parafac2 model (with Mode A nested in Mode C) can be written as

\[
X[[k]] = \text{tcrossprod}(A[[k]]%*%\text{diag}(C[k],),B) + E[[k]]
\]

subject to crossprod(A[[k]]) = Phi
where \( A[[k]] = \text{matrix}(a_k, I[k], R) \) are the Mode A (first mode) weights for the \( k \)-th level of Mode C (third mode), \( \Phi \) is the common crossproduct matrix shared by all \( K \) levels of Mode C, \( B = \text{matrix}(b, J, R) \) are the Mode B (second mode) weights, \( C = \text{matrix}(c, K, R) \) are the Mode C (third mode) weights, and \( E[[k]] = \text{matrix}(e_k, I[k], J) \) is the residual matrix corresponding to \( k \)-th level of Mode C.

Given a list of arrays \( X[[1]] = \text{array}(x_1, \text{dim}=c(I[1], J, K)) \) for \( l = \text{seq}(1, L) \), the 4-way Parafac2 model (with Mode A nested in Mode D) can be written as

\[
X[[1]][,,k] = \text{tcrossprod}(A[[1]]%*%\text{diag}(D[l,]*C[k]),B) + E[[k]]
\]

subject to \( \text{crossprod}(A[[1]]) = \Phi \)

\( A[[1]] = \text{matrix}(a_1, I[1], R) \) are the Mode A (first mode) weights for the \( l \)-th level of Mode D (fourth mode), \( \Phi \) is the common crossproduct matrix shared by all \( L \) levels of Mode D, \( D = \text{matrix}(d, L, R) \) are the Mode D (fourth mode) weights, and \( E[[1]] = \text{matrix}(e_1, I[1], J, K) \) is the residual array corresponding to \( l \)-th level of Mode D.

Weight matrices are estimated using an alternating least squares algorithm with optional constraints.

**Usage**

```r
parafac2(X, nfac, nstart=10, const=NULL, control=NULL,
          Gfixed=NULL, Bfixed=NULL, Cfixed=NULL, Dfixed=NULL,
          Gstart=NULL, Bstart=NULL, Cstart=NULL, Dstart=NULL,
          Gstruc=NULL, Bstruc=NULL, Cstruc=NULL, Dstruc=NULL,
          maxit=500, ctol=1e-4, parallel=FALSE,
          cl=NULL, output=c("best","all"))
```

**Arguments**

- **X**
  - For 3-way Parafac2: list of length \( K \) where \( k \)-th element is \( I[k]\)-by-\( J \) matrix or three-way data array with `dim=c(I,J,K)`. For 4-way Parafac2: list of length \( L \) where \( l \)-th element is \( I[l]\)-by-\( J \)-by-\( K \) array or four-way data array with `dim=c(I,J,K,L)`.
- **nfac**
  - Number of factors.
- **nstart**
  - Number of random starts.
- **const**
  - Constraints for each mode. Vector of length 3 or 4 with entries: 0 = unconstrained (default), 1 = orthogonal, 2 = non-negative, 3 = unimodal, 4 = monotonic, 5 = periodic, 6 = smooth. Use `control` argument to adjust options for constraints 3-6. Note: constraints 2-4 cannot be applied on Mode A.
- **control**
  - List of parameters controlling options for constraints 3-6. This is passed to `const.control`, which describes the available options.
- **Gfixed**
  - Fixed Mode A crossproducts (`crossprod(Gfixed)=Phi`). Only used to fit model with fixed \( \Phi \) matrix.
- **Bfixed**
  - Fixed Mode B weights. Only used to fit model with fixed Mode B weights.
- **Cfixed**
  - Fixed Mode C weights. Only used to fit model with fixed Mode C weights.
- **Dfixed**
  - Fixed Mode D weights. Only used to fit model with fixed Mode D weights.
parafac2

Gstart
Starting Mode A crossproduct matrix for ALS algorithm (crossprod(Gstart)=Phi).
Default uses random weights.

Bstart
Starting Mode B weights for ALS algorithm. Default uses random weights.

Cstart
Starting Mode C weights for ALS algorithm. Default uses random weights.

Dstart
Starting Mode D weights for ALS algorithm. Default uses random weights.

Gstruc
Structure constraints for Mode A crossproduct matrix (crossprod(Gstruc) = Phi structure).
Default uses unstructured crossproducts.

Bstruc
Structure constraints for Mode B weights. Default uses unstructured weights.

Cstruc
Structure constraints for Mode C weights. Default uses unstructured weights.

Dstruc
Structure constraints for Mode D weights. Default uses unstructured weights.

maxit
Maximum number of iterations.

tol
Convergence tolerance.

parallel
Logical indicating if parLapply should be used. See Examples.

cl
Cluster created by makeCluster. Only used when parallel=TRUE.

output
Output the best solution (default) or output all nstart solutions.

Value
If output="best", returns an object of class "parafac2" with the following elements:

A
List of Mode A weight matrices.

B
Mode B weight matrix.

C
Mode C weight matrix.

D
Mode D weight matrix.

Phi
Mode A crossproduct matrix.

SSE
Sum of Squared Errors.

Rsq
R-squared value.

GCV
Generalized Cross-Validation.

edf
Effective degrees of freedom.

iter
Number of iterations.

cflag
Convergence flag.

const
See argument const.

control
See argument control.

fixed
Logical vector indicating whether 'fixed' weights were used for each mode.

struc
Logical vector indicating whether 'struc' constraints were used for each mode.

Otherwise returns a list of length nstart where each element is an object of class "parafac2".
Warnings

The ALS algorithm can perform poorly if the number of factors nfac is set too large.
Non-negativity constraints can be sensitive to local optima.
Non-negativity constraints can result in slower performance.
Structure constraints for override constraints in const input.

Note

Default use is 10 random starts (nstart=10) with 500 maximum iterations of the ALS algorithm for each start (maxit=500) using a convergence tolerance of 1e-4 (ctol=1e-4). The algorithm is determined to have converged once the change in R^2 is less than or equal to ctol.
Output cflag gives convergence information: cflag=0 if ALS algorithm converged normally, cflag=1 if maximum iteration limit was reached before convergence, and cflag=2 if ALS algorithm terminated abnormally due to a problem with the constraints.
Constraints 3 (unimodality) and 4 (monotonicity) are implemented using I-splines, and constraints 5 (periodicity) and 6 (smoothness) are implemented using M-splines (see Ramsay, 1988).

Author(s)

Nathaniel E. Helwig <helwig@umn.edu>

References


Examples

############# 3-way example #############

# create random data list with Parafac2 structure
set.seed(3)
mydim <- c(NA,10,20)
nf <- 2
nk <- sample(c(50,100,200),mydim[3],replace=TRUE)
Gmat <- matrix(rnorm(nf^2),nf,nf)
Bmat <- matrix(rnorm(mydim[1]*nf),mydim[2],nf)
Cmat <- matrix(rnorm(mydim[3]*nf),mydim[3],nf)
Xmat <- Emat <- Hmat <- vector("list",mydim[3])
for(k in 1:mydim[3]){
  Hmat[[k]] <- svd(matrix(rnorm(nk[k]*nf),nk[k],nf),nv=0)$u
  Xmat[[k]] <- tcrossprod(Hmat[[k]]%*%Gmat%*%diag(Cmat[[k]]),Bmat)
  Emat[[k]] <- matrix(rnorm(nk[k]*mydim[2]),nk[k],mydim[2])
}
Emat <- nscale(Emat,0,sumsq(Xmat))  # SNR=1
X <- sapply("*",Xmat,Emat)

# fit Parafac2 model (unconstrained)
pfac <- parafac2(X,nfac=nf,nstart=1)
pfac

# check solution
Xhat <- fitted(pfac)
sse <- sumsq(sapply("-",Xmat,Xhat))
sse/(sum(nk)*mydim[2])
crossprod(pfac$A[[1]])
crossprod(pfac$A[[2]])
pfac$Phi

# reorder and resign factors
pfac$B[1:4,,]
pfac$B[2:1,,]
pfac <- resign(pfac,mode="B")
pfac$B[1:4,,]
Xhat <- fitted(pfac)
sse <- sumsq(sapply("-",Xmat,Xhat))
sse/(sum(nk)*mydim[2])

# rescale factors
colSums(pfac$B*2)
colSums(pfac$C*2)
pfac <- rescale(pfac,mode="C",absorb="B")
colSums(pfac$B*2)
colSums(pfac$C*2)
Xhat <- fitted(pfac)
sse <- sumsq(sapply("-",Xmat,Xhat))
sse/(sum(nk)*mydim[2])

########### 4-way example ###########

# create random data list with Parafac2 structure
set.seed(4)
mydim <- c(NA,10,20,5)
nf <- 3
nk <- sample(c(50,100,200),mydim[4],replace=TRUE)
Gmat <- matrix(rnorm(nf^2),nf,nf)
Bmat <- scale(matrix(rnorm(mydim[2]*nf),mydim[2],nf), center=FALSE)
cseq <- seq(-3, 3, length=mydim[3])
Cmat <- cbind(pnorm(cseq), pgamma(cseq+3.1, shape=1, rate=1)*(3/4), pt(cseq-2, df=4)*2)
Dmat <- scale(matrix(runif(mydim[4]*nf)*2, mydim[4], nf), center=FALSE)
Xmat <- Emat <- Hmat <- vector("list", mydim[4])
for(k in 1:mydim[4]){  
  aseq <- seq(-3, 3, length=nk)
  Hmat[[k]] <- svd(cbind(sin(aseq), sin(abs(aseq)), exp(~aseq^2)), nv=0)
  Xmat[[k]] <- array(tcrossprod(Hmat[[k]]%*%Gmat%*%diag(Dmat[[k]]),  
                        krprod(Cmat,Bmat)),dim=c(nk,mydim[2],mydim[3]))
  Emat[[k]] <- array(rnorm(nk*mydim[2]*mydim[3]),dim=c(nk,mydim[2],mydim[3]))
}
Emat <- nscale(Emat,0,sumsq(Xmat))  # SNR=1
X <- mapply("+",Xmat,Emat)

# fit Parafac model (smooth A, unconstrained B, monotonic C, non-negative D)
pfac <- parafac2(X,nfac=nf,nstart=1,const=c(6,0,4,2))
pfac

# check solution
Xhat <- fitted(pfac)
sse <- sumsq(mapply("-",Xmat,Xhat))
sse/(sum(nk)*mydim[2]*mydim[3])
crossprod(pfac$A[[1]])
crossprod(pfac$A[[2]])
pfac$Phi

## Not run:

############## parallel computation ##############

# create random data list with Parafac2 structure
set.seed(3)
mydim <- c(NA,10,20)
nf <- 2
nk <- sample(c(50,100,200),mydim[3],replace=TRUE)
Gmat <- matrix(rnorm(nf*2),nf,nf)
Bmat <- matrix(rnorm(mydim[2]*nf),mydim[2],nf)
Cmat <- matrix(rnorm(mydim[3]*nf),mydim[3],nf)
Xmat <- Emat <- Hmat <- vector("list", mydim[3])
for(k in 1:mydim[3]){  
  Hmat[[k]] <- svd(matrix(rnorm(nk*nf),nk,nf),nv=0)
  Xmat[[k]] <- tcrossprod(Hmat[[k]]%*%Gmat%*%diag(Cmat[[k]]),Bmat)
  Emat[[k]] <- matrix(rnorm(nk*mydim[2]),nk,mydim[2])
}
Emat <- nscale(Emat,0,sumsq(Xmat))  # SNR=1
X <- mapply("+",Xmat,Emat)

# fit Parafac2 model (10 random starts -- sequential computation)
set.seed(1)
system.time({pfac <- parafac2(X,nfac=nf)})
pfac
# fit Parafac2 model (10 random starts -- parallel computation)
set.seed(1)
cl <- makeCluster(detectCores())
ce <- clusterEvalQ(cl,library(multiway))
system.time({pfac <- parafac2(X,nfac=nf,parallel=TRUE,cl=cl)}

pfac
stopCluster(cl)

## End(Not run)

---

**print**

*Print Multi-Way Model*

**Description**

Prints constraint, fit, and convergence details for a fit multiway model.

**Usage**

```r
## S3 method for class 'indscal'
print(x,...)
## S3 method for class 'parafac'
print(x,...)
## S3 method for class 'parafac2'
print(x,...)
## S3 method for class 'sca'
print(x,...)
## S3 method for class 'tucker'
print(x,...)
```

**Arguments**

- `x` Object of class "indscal" (output from `indscal`), class "parafac" (output from `parafac`), class "parafac2" (output from `parafac2`), class "sca" (output from `sca`), or class "tucker" (output from `tucker`).
- `...` Ignored.

**Details**

See `indscal`, `parafac`, `parafac2`, `sca`, and `tucker` for examples.

**Author(s)**

Nathaniel E. Helwig <helwig@umn.edu>
Examples

```r
### see examples for indscal, parafac, parafac2, sca, and tucker
```

## Description

Reorders factors from a multiway object.

## Usage

```r
## S3 method for class 'indscal'
reorder(x, neworder, ...)
## S3 method for class 'parafac'
reorder(x, neworder, ...)
## S3 method for class 'parafac2'
reorder(x, neworder, ...)
## S3 method for class 'sca'
reorder(x, neworder, ...)
## S3 method for class 'tucker'
reorder(x, neworder, mode="A", ...)
```

## Arguments

- `x` Object of class "indscal" (output from `indscal`), class "parafac" (output from `parafac`), class "parafac2" (output from `parafac2`), class "sca" (output from `sca`), or class "tucker" (output from `tucker`).
- `neworder` Vector specifying the new factor ordering. Must be a permutation of the integers 1 to nfac.
- `mode` Character indicating which mode to reorder (only for tucker models). For 3-way Tucker options include "A", "B", and "C". For 4-way Tucker, options are "A", "B", "C", and "D".
- `...` Ignored.

## Details

See `indscal`, `parafac`, `parafac2`, `sca`, and `tucker` for more details.

## Value

Same as input.
Author(s)
Nathaniel E. Helwig <helwig@umn.edu>

Examples

### see examples for indscal, parafac, parafac2, sca, and tucker

---

rescale  

Rescales Multi-Way Factors

Description

Rescales factors from a multiway object.

Usage

```r
## S3 method for class 'indscal'
rescale(x, mode="B", newscale=1, ...)
## S3 method for class 'parafac'
rescale(x, mode="A", newscale=1, absorb="C", ...)
## S3 method for class 'parafac2'
rescale(x, mode="A", newscale=1, absorb="C", ...)
## S3 method for class 'sca'
rescale(x, mode="B", newscale=1, ...)
## S3 method for class 'tucker'
rescale(x, mode="A", newscale=1, ...)
```

Arguments

- **x**: Object of class "indscal" (output from `indscal`), class "parafac" (output from `parafac`), class "parafac2" (output from `parafac2`), class "sca" (output from `sca`), or class "tucker" (output from `tucker`).
- **mode**: Character indicating which mode to rescale.
- **newscale**: Desired root mean-square for each column of rescaled mode. Can input a scalar or a vector with length equal to the number of factors for the given mode.
- **absorb**: Character indicating which mode should absorb the inverse of the rescalings applied to mode (cannot be equal to mode).
- **...**: Ignored.

Details

See `indscal`, `parafac`, `parafac2`, `sca`, and `tucker` for more details.
Value
Same as input.

Author(s)
Nathaniel E. Helwig <helwig@umn.edu>

References

Examples

### see examples for indscal, parafac, parafac2, sca, and tucker

## resign

### Resigns Multi-Way Factors

### Description
Resigns factors from a multiway object.

### Usage

```r
# S3 method for class 'indscal'
resign(x, mode="B", newsign=1, ...)
# S3 method for class 'parafac'
resign(x, mode="A", newsign=1, absorb="C", ...)
# S3 method for class 'parafac2'
resign(x, mode="A", newsign=1, absorb="C", method="pearson", ...)
# S3 method for class 'sca'
resign(x, mode="B", newsign=1, ...)
# S3 method for class 'tucker'
resign(x, mode="A", newsign=1, ...)
```

### Arguments

- **x**
  Object of class "indscal" (output from indscal), class "parafac" (output from parafac), class "parafac2" (output from parafac2), class "sca" (output from sca), or class "tucker" (output from tucker).

- **mode**
  Character indicating which mode to resign.
newsign Desired sign for each column of resigned mode. Can input a scalar or a vector with length equal to the number of factors for the given mode. If \( x \) is of class "parafac2" and \( \text{mode} = \text{"A"} \) you can input a list of covariates (see Details).

absorb Character indicating which mode should absorb the inverse of the rescalings applied to mode (cannot be equal to mode).

method Correlation method to use if \( \text{newsign} \) is a list input (see Details).

... Ignored.

Details

If \( x \) is of class "parafac2" and \( \text{mode} = \text{"A"} \), the input \( \text{newsign} \) can be a list where each element contains a covariate vector for resigning Mode A. You need \( \text{length}(\text{newsign}[[k]]) = \text{nrow}(x\$A[[k]]) \) for all \( k \) when \( \text{newsign} \) is a list. In this case, the resigning is implemented according to the sign of \( \text{cor}(\text{newsign}[[k]], x\$A[[k]][1,]) \) \text{method} \). See Helwig (2013) for details.

See \text{indscal}, \text{parafac}, \text{parafac2}, \text{sca}, and \text{tucker} for more details.

Value

Same as input.

Author(s)

Nathaniel E. Helwig <helwig@umn.edu>

References


Examples

### see examples for indscal, parafac, parafac2, sca, and tucker

---

\text{sca} \hspace{1cm} \text{Simultaneous Component Analysis}

Description

Given a list of matrices \( X[[k]] = \text{matrix}(x[k, I[k], J) \) for \( k = \text{seq}(1,K) \), the SCA model is

\[ X[[k]] = \text{tcrossprod}(D[[k]],B) + E[[k]] \]

where \( D[[k]] = \text{matrix}(d[k, I[k], R) \) are the Mode A (first mode) weights for the k-th level
of Mode C (third mode), \( B = \text{matrix}(b,J,R) \) are the Mode B (second mode) weights, and
\( E[[k]] = \text{matrix}(ek,I[k],J) \) is the residual matrix corresponding to k-th level of Mode C.

There are four different versions of the SCA model: SCA with invariant pattern (SCA-P), SCA with Parafac2 constraints (SCA-PF2), SCA with INDSCAL constraints (SCA-IND), and SCA with equal average crossproducts (SCA-ECP). These four models differ with respect to the assumed crossproduct structure of the \( D[[k]] \) weights:

- **SCA-P:** \( \text{crossprod}(D[[k]])/I[k] = \text{Phi}[[k]] \)
- **SCA-PF2:** \( \text{crossprod}(D[[k]])/I[k] = \text{diag}(C[k])*\text{Phi}*\text{diag}(C[k]) \)
- **SCA-IND:** \( \text{crossprod}(D[[k]])/I[k] = \text{diag}(C[k])*C[k] \)
- **SCA-ECP:** \( \text{crossprod}(D[[k]])/I[k] = \text{Phi} \)

where \( \text{Phi}[[k]] \) is specific to the k-th level of Mode C, \( \text{Phi} \) is common to all k levels of Mode C, and \( C = \text{matrix}(c,K,R) \) are the Mode C (third mode) weights. This function estimates the weight matrices \( D[[k]] \) and \( B \) (and \( C \) if applicable) using alternating least squares.

**Usage**

```r
csa(x,nfac,nstart=10,maxit=500,
  type=c("sca-p","sca-pf2","sca-ind","sca-ecp"),
  rotation=c("none","varimax","promax"),
  ctol=1e-4,parallel=FALSE,cl=NULL)
```

**Arguments**

- **x**
  - List of length \( K \) where the k-th element contains the \( I[k]\)-by-\( J \) data matrix \( X[[k]] \). If \( I[k]=I[1] \) for all \( k \), can input 3-way data array with \( \text{dim}=c(I,J,K) \).
- **nfac**
  - Number of factors.
- **nstart**
  - Number of random starts.
- **maxit**
  - Maximum number of iterations.
- **type**
  - Type of SCA model to fit.
- **rotation**
  - Rotation to use for type="sca-p" or type="sca-ecp".
- **ctol**
  - Convergence tolerance.
- **parallel**
  - Logical indicating if `parLapply` should be used. See Examples.
- **cl**
  - Cluster created by `makeCluster`. Only used when `parallel=TRUE`.

**Value**

- **D**
  - List of length \( K \) where k-th element contains \( D[[k]] \).
- **B**
  - Mode B weight matrix.
- **C**
  - Mode C weight matrix.
- **Phi**
  - Mode A common crossproduct matrix (if `type!="sca-p"`).
- **SSE**
  - Sum of Squared Errors.
- **Rsq**
  - R-squared value.
Generalized Cross-Validation.

Effective degrees of freedom.

Number of iterations.

Convergence flag.

Same as input type.

Same as input rotation.

The ALS algorithm can perform poorly if the number of factors \( n_{\text{fac}} \) is set too large.

The least squares SCA-P solution can be obtained from the singular value decomposition of the stacked matrix \( \text{rbind}(X[[1]], \ldots, X[[K]]) \).

The least squares SCA-PF2 solution can be obtained using the unconstrained Parafac2 ALS algorithm (see \texttt{parafac2}).

The least squares SCA-IND solution can be obtained using the Parafac2 ALS algorithm with orthogonality constraints on Mode A.

The least squares SCA-ECP solution can be obtained using the Parafac2 ALS algorithm with orthogonality constraints on Mode A and the Mode C weights fixed at \( C[k,] = \text{rep}(1[k]^{0.5}, R) \).

Default use is 10 random starts (\( n_{\text{start}} = 10 \)) with 500 maximum iterations of the ALS algorithm for each start (\( \text{maxit} = 500 \)) using a convergence tolerance of \( 1e-4 \) (\( \text{ctol} = 1e-4 \)). The algorithm is determined to have converged once the change in \( R^2 \) is less than or equal to \( \text{ctol} \).

Output \( \text{cflag} \) gives convergence information: \( \text{cflag} = 0 \) if ALS algorithm converged normally, \( \text{cflag} = 1 \) if maximum iteration limit was reached before convergence, and \( \text{cflag} = 2 \) if ALS algorithm terminated abnormally due to problem with non-negativity constraints.

Nathaniel E. Helwig \(<\text{helwig@umn.edu}>\)


Examples

```
# create random data list with SCA-P structure
set.seed(3)
mydim <- c(NA,10,20)
f <- 2
nk <- sample(c(50,100,200),mydim[3],replace=TRUE)
Dmat <- matrix(rnorm(sum(nk)*f),sum(nk),f)
Bmat <- matrix(runif(mydim[2]*f),mydim[2],f)
Dmats <- vector("list",mydim[3])
Xmat <- Emat <- vector("list",mydim[3])
dfc <- 0
for(k in 1:mydim[3]){
  dinds <- 1:nk[k] + dfc
  Dmats[[k]] <- Dmat[dinds,]
  dfc <- dfc + nk[k]
  Xmat[[k]] <- tcrossprod(Dmats[[k]],Bmat)
  Emat[[k]] <- matrix(rnorm(nk[k]*mydim[2]),nk[k],mydim[2])
}
rm(Dmat)
Emat <- nscale(Emat,0,sumsq(Xmat))  # SNR=1
X <- mapply("+",Xmat,Emat)

# fit SCA-P model (no rotation)
scamod <- sca(X,nfac=nf,nstart=1)
scamod

# check solution
crossprod(scamod$D[[1]])  #% diag(scamod$C[1,]^2-1) ) / nk[1]
crossprod(scamod$D[[5]])  #% diag(scamod$C[5,]^2-1) ) / nk[5]
Xhat <- fitted(scamod)
sse <- sumsq(mapply("-",Xmat,Xhat))
sse/(sum(nk)*mydim[2])

# reorder and resign factors
scamod$B[1:4,]
scamod <- reorder(scamod, 2:1)
scamod$B[1:4,]
scamod <- resign(scamod, mode="B", newsign=c(1,-1))
scamod$B[1:4,]
Xhat <- fitted(scamod)
sse <- sumsq(mapply("-",Xmat,Xhat))
sse/(sum(nk)*mydim[2])

# rescale factors
colSums(scamod$B^2)
colSums(scamod$C^2)
scamod <- rescale(scamod, mode="C")
colSums(scamod$B^2)
colSums(scamod$C^2)
Xhat <- fitted(scamod)
```
```r
sse <- sumsq(mapply("-", Xmat, xhat))
ssesum(nk)*mydim[2])

############################################################
sca-pf2############################################################

# create random data list with SCA-PF2 (Parafac2) structure
set.seed(3)
mydim <- c(NA, 10, 20)
f <- 2
nk <- sample(c(50, 100, 200), mydim[3], replace=TRUE)
Gmat <- 10*matrix(rnorm(nk^2), nk, nk)
Bmat <- matrix(runif(mydim[2]*nk), mydim[2], nk)
Cmat <- matrix(runif(mydim[3]*nk), mydim[3], nk)
Xmat <- Emat <- Fmat <- vector("list", mydim[3])
for(k in 1:mydim[3]){
  Fmat[[k]] <- svd(matrix(rnorm(nk[k]*nk), nk[k], nk), nv=0)$u
  Xmat[[k]] <- t(crossprod(Fmat[[k]]%*%Gmat%*%diag(Cmat[k,]), Bmat))
  Emat[[k]] <- matrix(rnorm(nk[k]*mydim[2]), nk[k], mydim[2])
}
Emat <- nscale(Emat, 0, sumsq(Xmat))  # SNR=1
X <- mapply("+", Xmat, Emat)

# fit SCA-PF2 model
scamod <- sca(X, nf = nf, nstart = 1, type = "sca-pf2")
scamod

# check solution
scamod$Phi
crossprod(scamod$D[[1]]%*%diag(scamod$C[1,]^(-1)) ) / nk[1]
crossprod(scamod$D[[5]]%*%diag(scamod$C[5,]^(-1)) ) / nk[5]
Xhat <- fitted(scamod)
ssesum(mapply("-", Xmat, xhat))
ssesum(nk)*mydim[2])

# reorder and resign factors
scamod$B[1:4,]
scamod <- reorder(scamod, 2:1)
scamod$B[1:4,]
scamod <- resign(scamod, mode = "B", newsign = c(1, -1))
scamod$B[1:4,]
Xhat <- fitted(scamod)
ssesum(mapply("-", Xmat, xhat))
ssesum(nk)*mydim[2])

# rescale factors
colSums(scamod$B^2)
colSums(scamod$C^2)
scamod <- rescale(scamod, mode = "C")
colSums(scamod$B^2)
colSums(scamod$C^2)
Xhat <- fitted(scamod)
ssesum(mapply("-", Xmat, xhat))
```
sca/sum(nk)*mydim[2])

### sca-ind ###

# create random data list with SCA-IND structure
set.seed(3)
mydim <- c(NA, 10, 20)
nf <- 2
nk <- sample(c(50, 100, 200), mydim[3], replace=TRUE)
Gmat <- diag(nf)  # SCA-IND is Parafac2 with Gmat=identity
Bmat <- matrix(runif(mydim[2]*nf), mydim[2], nf)
Cmat <- 10*matrix(runif(mydim[3]*nf), mydim[3], nf)
Xmat <- Emat <- Fmat <- vector("list", mydim[3])
for(k in 1:mydim[3]){
    Fmat[[k]] <- svd(matrix(rnorm(nk[k]*nf), nk[k], nf), nv=0)$u
    Xmat[[k]] <- tcrossprod(Fmat[[k]]%*%Gmat%*%diag(Cmat[k,]), Bmat)
    Emat[[k]] <- matrix(rnorm(nk[k]*mydim[2]), nk[k], mydim[2])
}
Emat <- nscale(Emat, 0, sumsq(Xmat))  # SNR=1
X <- mapply("-", Xmat, Emat)

# fit SCA-IND model
scamod <- sca(X, nfac=nf, nstart=1, type="sca-ind")
scamod

# check solution
scamod$Phi
crossprod(scamod$D[[1]] %*% diag(scamod$C[[1]]^-1)) / nk[1]
crossprod(scamod$D[[5]] %*% diag(scamod$C[[5]]^-1)) / nk[5]
Xhat <- fitted(scamod)
sse <- sumsq(mapply("-", Xmat, Xhat))
sse/sum(nk)*mydim[2])

# reorder and resign factors
scamodB[1:4,]
scamod <- reorder(scamod, 2:1)
scamodB[1:4,]
scamod <- resign(scamod, mode="B", newsign=c(1,-1))
scamodB[1:4,]
Xhat <- fitted(scamod)
sse <- sumsq(mapply("-", Xmat, Xhat))
sse/sum(nk)*mydim[2])

# rescale factors
cols(scamodB[2,])
cols(scamodC[2,])
scamod <- rescale(scamod, mode="C")
cols(scamodB[2,])
cols(scamodC[2,])
Xhat <- fitted(scamod)
sse <- sumsq(mapply("-", Xmat, Xhat))
sse/sum(nk)*mydim[2])
### sca-ecp

# create random data list with SCA-ECP structure
set.seed(3)
mydim <- c(NA, 10, 20)
f <- 2
nk <- sample(c(50, 100, 200), mydim[3], replace=TRUE)
Gmat <- diag(nf)
Bmat <- matrix(runif(mydim[2]*nf), mydim[2], nf)
Cmat <- matrix(sqrt(nk), mydim[3], nf)
Xmat <- Emat <- Fmat <- vector("list", mydim[3])
for(k in 1:mydim[3]){  
  Fmat[[k]] <- svd(matrix(rnorm(nk[1]*nf), nk[1], nf), nv=0)$u  
  Xmat[[k]] <- tcrossprod(Fmat[[k]]%*%Gmat%*%diag(Cmat[k,]), Bmat)  
  Emat[[k]] <- matrix(rnorm(nk[1]*mydim[2]), nk[1], mydim[2])
}
Emat <- nscale(Emat, 0, sumsq(Xmat))  # SNR=1
X <- mapply("+", Xmat, Emat)

# fit SCA-ECP model
scamod <- sca(X, nfac=nf, nstart=1, type="sca-ecp")
scamod

# check solution
scamod$Phi
crossprod(scamod$D[[1]] %*% diag(scamod$C[1,]^(-1))) / nk[1]
crossprod(scamod$D[[5]] %*% diag(scamod$C[5,]^(-1))) / nk[5]
Xhat <- fitted(scamod)
sse <- sumsq(mapply("-", Xmat, Xhat))
sse/(sum(nk)*mydim[2])

# reorder and resign factors
scamod$B[1:4,]
scamod <- reorder(scamod, 2:1)
scamod$B[1:4,]
scamod <- resign(scamod, mode="B", newsign=c(-1,1))
scamod$B[1:4,]
Xhat <- fitted(scamod)
sse <- sumsq(mapply("-", Xmat, Xhat))
sse/(sum(nk)*mydim[2])

# rescale factors
colSums(scamod$B^2)
colSums(scamod$C^2)
scamod <- rescale(scamod, mode="B")
colSums(scamod$B^2)
colSums(scamod$C^2)
Xhat <- fitted(scamod)
sse <- sumsq(mapply("-", Xmat, Xhat))
sse/(sum(nk)*mydim[2])
## Not run:

```
#### parallel computation ####

# create random data list with SCA-IND structure
set.seed(3)
mydim <- c(NA,10,20)
f <- 2
nk <- sample(c(50,100,200),mydim[3],replace=TRUE)
Gmat <- diag(nf) # SCA-IND is Parafac2 with Gmat=identity
Bmat <- matrix(rnorm(mydim[2]*nf),mydim[2],nf)
Cmat <- 10*matrix(runif(mydim[3]*nf),mydim[3],nf)
Xmat <- Emat <- Fmat <- gmat <- identity
bmat <- matrix(rnorm(nf*mydim[2]),nk[2],mydim[2])
for(k in 1:mydim[3]){  # snr=1
  Fmat[[k]] <- svd(matrix(rnorm(nk[2]*nf),nk[2],nf),nv=0)$u
  Xmat[[k]] <- tcrossprod(Fmat[[k]],Gmat*diag(Cmat[[k]]),Bmat)
  Emat[[k]] <- matrix(rnorm(nf*mydim[2]),nk[2],mydim[2])
}
Emat <- nscale(Emat,0,sumsq(Xmat))
X <- mapply("+",Xmat,Emat)

# fit SCA-PF2 model (10 random starts -- sequential computation)
set.seed(1)
system.time((scamod <- sca(X,nfac=nf,type="sca-pf2")))
scamod

# fit SCA-PF2 model (10 random starts -- parallel computation)
set.seed(1)
cl <- makeCluster(detectCores())
ce <- clusterEvalQ(cl,library(multiway))
system.time((scamod <- sca(X,nfac=nf,type="sca-pf2",parallel=TRUE,cl=cl)))
scamod
stopCluster(cl)

# fit SCA-IND model (10 random starts -- sequential computation)
set.seed(1)
system.time((scamod <- sca(X,nfac=nf,type="sca-ind")))
scamod

# fit SCA-IND model (10 random starts -- parallel computation)
set.seed(1)
cl <- makeCluster(detectCores())
ce <- clusterEvalQ(cl,library(multiway))
system.time((scamod <- sca(X,nfac=nf,type="sca-ind",parallel=TRUE,cl=cl)))
scamod
stopCluster(cl)

# fit SCA-ECP model (10 random starts -- sequential computation)
set.seed(1)
system.time((scamod <- sca(X,nfac=nf,type="sca-ecp")))
scamod
```
# fit SCA-ECP model (10 random starts -- parallel computation)
set.seed(1)
c1 <- makeCluster(detectCores())
ce <- clusterEvalQ(cl=library(multiway))
system.time({scamod <- sca(X,nfac=nf,type="sca-ecp",parallel=TRUE,cl=c1)})
scamod
stopCluster(cl)

## End(Not run)

---

**smpower**

*Symmetric Matrix Power*

**Description**

Raise symmetric matrix to specified power. Default calculates symmetric square root.

**Usage**

```r
smpower(X, power=0.5, tol=Machine$double.eps)
```

**Arguments**

- `X`: Symmetric real-valued matrix.
- `power`: Power to apply to eigenvalues of `X`.
- `tol`: Stability tolerance for eigenvalues.

**Value**

Returns `X` raised to specified power.

**Note**

Basically returns `tcrossprod(Y$vec^%*%diag(Y$val^power),Y$vec)` where `Y = eigen(X,symmetric=TRUE)`.

**Author(s)**

Nathaniel E. Helwig <helwig@umn.edu>

**Examples**

```r
# example
X <- crossprod(matrix(rnorm(2000),100,20))
Xsqrt <- smpower(X)  # square root
Xinv <- smpower(X,-1)  # inverse
Xisqrt <- smpower(X,-0.5)  # inverse square root
```
sumsq

Sum-of-Squares of Given Object

Description
Calculates the sum-of-squares of X.

Usage
sumsq(X)

Arguments
X Numeric scalar, vector, list, matrix, or array.

Value
Sum-of-squares of X.

Author(s)
Nathaniel E. Helwig <helwig@umn.edu>

Examples

```
### EXAMPLE 1
X <- 1:10
sumsq(X)

### EXAMPLE 2
X <- 1:10
sumsq(X)

### EXAMPLE 3
X <- matrix(1:10,5,2)
sumsq(X)

### EXAMPLE 4
X <- array(matrix(1:10,5,2),dim=c(5,2,2))
sumsq(X)

### EXAMPLE 5
X <- vector("list",5)
for(k in 1:5){ X[[k]] <- matrix(1:10,5,2) }
sumsq(X)
```
**Description**

Given a 3-way array $X = \text{array}(x, \text{dim}=c(I,J,K))$, the 3-way Tucker model can be written as

$$X[i,j,k] = \sum \sum \sum A[i,p]B[j,q]C[k,r]G[p,q,r] + E[i,j,k]$$

where $A = \text{matrix}(a, I, P)$ are the Mode A (first mode) weights, $B = \text{matrix}(b, J, Q)$ are the Mode B (second mode) weights, $C = \text{matrix}(c, K, R)$ are the Mode C (third mode) weights, $G = \text{array}(g, \text{dim}=c(P, Q, R))$ is the 3-way core array, and $E = \text{array}(e, \text{dim}=c(I,J,K))$ is the 3-way residual array. The summations are for $p = \text{seq}(1, P)$, $q = \text{seq}(1, Q)$, and $r = \text{seq}(1, R)$.

Given a 4-way array $X = \text{array}(x, \text{dim}=c(I,J,K,L))$, the 4-way Tucker model can be written as

$$X[i,j,k,l] = \sum \sum \sum \sum A[i,p]B[j,q]C[k,r]D[l,s]G[p,q,r,s] + E[i,j,k,l]$$

where $D = \text{matrix}(d, L, S)$ are the Mode D (fourth mode) weights, $G = \text{array}(g, \text{dim}=c(P, Q, R, S))$ is the 4-way residual array, $E = \text{array}(e, \text{dim}=c(I,J,K,L))$ is the 4-way residual array, and the other terms can be interpreted as previously described.

Weight matrices are estimated using an alternating least squares algorithm.

**Usage**

```r
tucker(X, nfac, nstart=10, Afixed=NULL, Bfixed=NULL, Cfixed=NULL, Dfixed=NULL, Bstart=NULL, Cstart=NULL, Dstart=NULL, maxit=500, ctol=1e-4, parallel=FALSE, cl=NULL, output=c("best","all"))
```

**Arguments**

- **X**: Three-way data array with dim=c(I,J,K) or four-way data array with dim=c(I,J,K,L).
- **nfac**: Number of factors in each mode.
- **nstart**: Number of random starts.
- **Afixed**: Fixed Mode A weights. Only used to fit model with fixed weights in Mode A.
- **Bfixed**: Fixed Mode B weights. Only used to fit model with fixed weights in Mode B.
- **Cfixed**: Fixed Mode C weights. Only used to fit model with fixed weights in Mode C.
- **Dfixed**: Fixed Mode D weights. Only used to fit model with fixed weights in Mode D.
- **Bstart**: Starting Mode B weights for ALS algorithm. Default uses random weights.
- **Cstart**: Starting Mode C weights for ALS algorithm. Default uses random weights.
- **Dstart**: Starting Mode D weights for ALS algorithm. Default uses random weights.
- **maxit**: Maximum number of iterations.
tucker

ctol Convergence tolerance.
parallel Logical indicating if parLapply should be used. See Examples.
cl Cluster created by makeCluster. Only used when parallel=TRUE.
output Output the best solution (default) or output all nstart solutions.

Value
If output="best", returns an object of class "tucker" with the following elements:
A Mode A weight matrix.
B Mode B weight matrix.
C Mode C weight matrix.
D Mode D weight matrix.
G Core array.
SSE Sum of Squared Errors.
Rsq R-squared value.
GCV Generalized Cross-Validation.
edf Effective degrees of freedom.
iter Number of iterations.
cflag Convergence flag.

Otherwise returns a list of length nstart where each element is an object of class "tucker".

Warnings
The ALS algorithm can perform poorly if the number of factors nfac is set too large.
Input matrices in Afixed, Bfixed, Cfixed, Dfixed, Bstart, Cstart, and Dstart must be columnwise orthonormal.

Note
Default use is 10 random starts (nstart=10) with 500 maximum iterations of the ALS algorithm for each start (maxit=500) using a convergence tolerance of 1e-4 (ctol=1e-4). The algorithm is determined to have converged once the change in $R^2$ is less than or equal to ctol.
Output cflag gives convergence information: cflag=0 if ALS algorithm converged normally, and cflag=1 if maximum iteration limit was reached before convergence.

Author(s)
Nathaniel E. Helwig <helwig@umn.edu>

References
Examples

!!!!!!!! 3-way example !!!!!!!!

# create random data array with Tucker structure
set.seed(3)
mydim <- c(50, 20, 5)
nf <- c(3, 2, 3)
Amat <- svd(matrix(rnorm(mydim[1] * nf[1]), mydim[1], nf[1]), nu=nf[1])$u
Cmat <- svd(matrix(rnorm(mydim[3] * nf[3]), mydim[3], nf[3]), nu=nf[3])$u
Gmat <- array(rnorm(prod(nf)), dim=nf)
Xmat <- array(tcrossprod(Amat %*% matrix(Gmat, nf[1], nf[2] * nf[3]), kronecker(Cmat, Bmat)), dim=mydim)
Emat <- array(rnorm(prod(mydim)), dim=mydim)
Emat <- nscale(Emat, 0, sumsq(Xmat))  # SNR=1
X <- Xmat + Emat

# fit Tucker model
tuck <- tucker(X, nfac=nf, nstart=1)
tuck

# check solution
Xhat <- fitted(tuck)
sum((Xmat-Xhat)^2)/prod(mydim)

# reorder mode="A"
tuck$A[1:4,]
tuck$G
tuck <- reorder(tuck, neworder=c(3,1,2), mode="A")
tuck$A[1:4,]
tuck$G
Xhat <- fitted(tuck)
sum((Xmat-Xhat)^2)/prod(mydim)

# reorder mode="B"
tuck$B[1:4,]
tuck$G
tuck <- reorder(tuck, neworder=2:1, mode="B")
tuck$B[1:4,]
tuck$G
Xhat <- fitted(tuck)
sum((Xmat-Xhat)^2)/prod(mydim)

# resign mode="C"
tuck$C[1:4,]
tuck <- resign(tuck, mode="C")
tuck$C[1:4,]
Xhat <- fitted(tuck)
sum((Xmat-Xhat)^2)/prod(mydim)

!!!!!!!! 4-way example !!!!!!!!
# create random data array with Tucker structure
set.seed(4)
mydim <- c(30,10,8,10)
f <- c(2,3,4,3)
Amat <- svd(matrix(rnorm(mydim[1]*nf[1]),mydim[1],nf[1]),nu=nf[1])$u
Bmat <- svd(matrix(rnorm(mydim[2]*nf[2]),mydim[2],nf[2]),nu=nf[2])$u
Cmat <- svd(matrix(rnorm(mydim[3]*nf[3]),mydim[3],nf[3]),nu=nf[3])$u
Dmat <- svd(matrix(rnorm(mydim[4]*nf[4]),mydim[4],nf[4]),nu=nf[4])$u
Gmat <- array(rnorm(prod(nf)),dim=nf)
Xmat <- array(tcrossprod(Amat%*%matrix(Gmat,nf[1],prod(nf[2:4])),
                       kronecker(Dmat,kronecker(Cmat,Bmat))),dim=mydim)
Emat <- array(rnorm(prod(mydim)),dim=mydim)
Emat <- nscale(Emat,0,sumsq(Xmat))  # SNR=1
X <- Xmat + Emat

# fit Tucker model
tuck <- tucker(X,nfac=nf,nstart=1)
tuck

# check solution
Xhat <- fitted(tuck)
sum((Xmat-Xhat)^2)/prod(mydim)

### Not run:

########## parallel computation ##########

# create random data array with Tucker structure
set.seed(3)
mydim <- c(50,20,5)
f <- c(3,2,3)
Amat <- svd(matrix(rnorm(mydim[1]*nf[1]),mydim[1],nf[1]),nu=nf[1])$u
Bmat <- svd(matrix(rnorm(mydim[2]*nf[2]),mydim[2],nf[2]),nu=nf[2])$u
Cmat <- svd(matrix(rnorm(mydim[3]*nf[3]),mydim[3],nf[3]),nu=nf[3])$u
Gmat <- array(rnorm(prod(nf)),dim=nf)
Xmat <- array(tcrossprod(Amat%*%matrix(Gmat,nf[1],nf[2]*nf[3]),kronecker(Cmat,Bmat)),dim=mydim)
Emat <- array(rnorm(prod(mydim)),dim=mydim)
Emat <- nscale(Emat,0,sumsq(Xmat))  # SNR=1
X <- Xmat + Emat

# fit Tucker model (10 random starts -- sequential computation)
set.seed(1)
system.time((tuck <- tucker(X,nfac=nf)))
tuck$Rsq

# fit Tucker model (10 random starts -- parallel computation)
set.seed(1)
cl <- makeCluster(detectCores())
ce <- clusterEvalQ(cl,library(multiway))
system.time((tuck <- tucker(X,nfac=nf,parallel=TRUE,cl=cl)))
tuck$Rsq
stopCluster(cl)
Description

This dataset contains yearly (1970-2013) consumption data from the 50 United States and the District of Columbia for three types of alcoholic beverages: spirits, wine, and beer. The data were obtained from the National Institute on Alcohol Abuse and Alcoholism (NIAAA) Surveillance Report #102 (see below link).

Usage

data("USalcohol")

Format

A data frame with 6732 observations on the following 8 variables.

- `year` integer Year (1970-2013)
- `state` factor State Name (51 levels)
- `region` factor Region Name (4 levels)
- `type` factor Beverage Type (3 levels)
- `beverage` numeric Beverage Consumed (thousands of gallons)
- `ethanol` numeric Absolute Alcohol Consumed (thousands of gallons)
- `pop14` numeric Population Age 14 and Older (thousands of people)
- `pop21` numeric Population Age 21 and Older (thousands of people)

Details

In the data source, the population age 21 and older for Mississippi in year 1989 is reported to be 3547.839 thousand, which is incorrect. In this dataset, the miscoded population value has been replaced with the average of the corresponding 1988 population (1709 thousand) and the 1990 population (1701.527 thousand).

Source

https://pubs.niaaa.nih.gov/publications/surveillance102/pcyr19702013.txt
References


Examples

```r
# load data and print first six rows
data(USAlcohol)
head(USAlcohol)

# form tensor (time x variables x state)
Xbev <- with(USAlcohol, tapply(beverage/pop21, list(year, type, state), c))
Xeth <- with(USAlcohol, tapply(ethanol/pop21, list(year, type, state), c))
X <- array(0, dim=c(44, 6, 51))
X[, c(1,3,5),] <- Xbev
X[, c(2,4,6),] <- Xeth
dnames <- dimnames(Xbev)
dnames[[2]] <- c(paste0(dnames[[2]],".bev"), paste0(dnames[[2]], ".eth"))[c(1,4,2,5,3,6)]
dimnames(X) <- dnames

# center each variable across time (within state)
Xc <- ncenter(X, mode=1)

# scale each variable to have mean square of 1 (across time and states)
Xs <- nscale(Xc, mode=2, ssnew=44*51)

# fit parafac model with 3 factors
set.seed(1)
 pfac <- parafac(Xs, nfac=3, nstart=1)

# fit parafac model with functional constraints
set.seed(1)
pfacF <- parafac(Xs, nfac=3, nstart=1, const=c(6,0,0))

# fit parafac model with functional and structural constraints
Bstruc <- matrix(c(rep(c(TRUE,FALSE), c(2,4)), rep(c(FALSE,TRUE,FALSE), c(2,2,2)), rep(c(FALSE,TRUE), c(4,2))), nrow=6, ncol=3)
set.seed(1)
pfacFS <- parafac(Xs, nfac=3, nstart=1, const=c(6,0,0), Bstruc=Bstruc)
```
Index

*Topic datasets
  USalcohol, 48

*Topic package
  multiway-package, 2

congru, 3
const.control, 5, 21, 25
corcondia, 6
dist, 11
fitted, 8
fnnls, 9, 11
indscal, 2, 8, 10, 30–34
krprod, 14
makeCluster, 11, 21, 26, 35, 45
mpinv, 15
multiway (multiway-package), 2
multiway-package, 2
ncenter, 16
nrescale (nscale), 18
nscale, 18
parafac, 2, 5, 6, 8, 20, 30–34
parafac2, 2, 5, 6, 8, 24, 30–34, 36
parLapply, 11, 21, 26, 35, 45
print, 30
reorder, 31
rescale, 32
resign, 33
sca, 2, 8, 30–34, 34
smpower, 42
sumsq, 43
tucker, 2, 8, 30–34, 44
USalcohol, 48