

# Package 'FAiR'

November 3, 2009

**Type** Package

**Title** Factor Analysis in R

**Version** 0.4-5

**Date** 2009-11-02

**Author** Ben Goodrich

**Maintainer** Ben Goodrich <bgokGM@gmail.com>

**Description** This package estimates factor analysis models using a genetic algorithm, which permits a general mechanism for restricted optimization with arbitrary restrictions that are chosen at run time with the help of a GUI. Importantly, inequality restrictions can be imposed on functions of multiple parameters, which provides a new avenues for testing and generating theories with factor analysis models. This package also includes an entirely new estimator of the common factor analysis model called semi-exploratory factor analysis, which is a general alternative to exploratory and confirmatory factor analysis. Finally, this package integrates a lot of other packages that estimate sample covariance matrices and thus provides a lot of alternatives to the traditional sample covariance calculation. Note that you need to have the Gtk run time library installed on your system to use this package; see the URL below for detailed installation instructions. Most users would only need to understand the first twenty-four pages of the PDF manual.

**URL** <http://wiki.r-project.org/rwiki/doku.php?id=packages:cran:fair>

**License** file LICENSE

**Encoding** UTF-8

**Imports** stats4, rrcov

**Enhances** stats4, rrcov

**Depends** R (>= 2.7.0), methods, rgenoud (>= 5.4-7), gWidgetsRGtk2 (>= 0.0-31), stats4, rrcov, Matrix

**Suggests** corpcor, mvnmle, polycor, nFactors, Rgraphviz, mvnormtest, energy, jit, GPArotation

**LazyLoad** yes

**Repository** CRAN

**Date/Publication** 2009-11-03 11:23:27

## R topics documented:

FAiR-package . . . . .	2
create_FAobject . . . . .	4
equality_restriction-class . . . . .	7
Factanal . . . . .	9
GPA2FA . . . . .	14
loadings . . . . .	15
make_manifest . . . . .	17
make_restrictions . . . . .	21
manifest-class . . . . .	28
mapping_rule . . . . .	30
model_comparison . . . . .	33
parameter-class . . . . .	36
read.cefa . . . . .	38
read.triangular . . . . .	40
restrictions-class . . . . .	41
restrictions2draws . . . . .	45
restrictions2Mathomatic . . . . .	47
restrictions2model . . . . .	48
restrictions2RAM . . . . .	50
Rotate . . . . .	52
S3methodsFAiR . . . . .	57
S4GenericsFAiR . . . . .	59
summary.FA-class . . . . .	61
<b>Index</b>	<b>63</b>

---

FAiR-package      *Factor Analysis in R*

---

### Description

This package estimates exploratory, confirmatory, and semi-exploratory factor analysis models via a genetic algorithm, namely `genoud`. This use of a genetic algorithm is tantamount to restricted optimization with virtually unlimited possibilities for restrictions. In particular, semi-exploratory factor analysis, which is new to the literature, minimizes a discrepancy function subject to a restriction on the *number* of exact zeros in each column of the primary pattern matrix but does not require that the *locations* of the zeros be specified in advance (as in confirmatory factor analysis). **FAiR** encourages the use of *inequality* restrictions on functions of multiple parameters to characterize testable hypotheses.

### Details

Package: FAiR  
 Type: Package  
 URL: <http://wiki.r-project.org/rwiki/doku.php?id=packages:cran:fair>  
 Version: 0.4-0

Date: 2008-07-09  
 License: See the LICENSE file for details.

Let the factor analysis model in the population be

$$\Sigma = \Omega(\beta\Phi\beta' + \Theta)\Omega$$

where  $\Sigma$  is the covariance matrix among outcome variables,  $\Omega$  is a diagonal matrix of standard deviations of the manifest variables,  $\beta$  is the primary pattern matrix (calibrated to standardized variables) with one column per factor,  $\Phi$  is the correlation matrix among the primary factors, and  $\Theta$  is the diagonal matrix of uniquenesses, which is fully determined by  $\beta$ ,  $\Phi$ , and the requirement that the matrix within parentheses has ones down its diagonal. Hence,  $\beta\Phi\beta' + \Theta$  is the model's purported correlation matrix among outcome variables as a function of the factors.

Each of the matrices on the right-hand side is a parameter to be estimated, and unlike many structural equation modeling programs, there is no mechanism for “translating” the model from a path diagram or otherwise avoiding the matrix algebra representation of the model. On a technical programming note, each of these matrices is represented by a `parameter-class` in **FAiR**, which includes a slot for the (proposed) estimate but also includes slots for ancillary information.

The usual steps to estimate and interpret a factor analysis model are as follows:

0. Get your data into **R** somehow. It is best to load the raw data in one of the usual fashions (e.g. `read.table`, `read.spss`, etc.) and the `read.cefa` function can also be used if your data are saved in the format used by CEFA 2.0. If you only have a covariance matrix, then `read.triangular` can be used to load it into **R**.
1. Call `make_manifest` to construct the left-hand side of the factor analysis model, namely an S4 object to house the sample estimate of  $\Sigma$  and some other information (e.g. number of observations).
2. Call `make_restrictions` to establish the additional restrictions to be imposed on the right-hand side of factor analysis model, inclusive of whether the model is exploratory, semi-exploratory, or confirmatory and what discrepancy function to use. There is an extensive GUI that pops up when `make_restrictions` is called to guide you through this step. **FAiR** differs fundamentally from other factor analysis software in that it permits you to impose inequality restrictions on functions of  $\beta$  and  $\Phi$ . Hence, the `restrictions-class` is critical to the way **FAiR** is programmed internally and houses S4 objects representing each of the matrices to be estimated.
3. Call `Factanal` to estimate the model and thereby produce estimates of  $\Omega$ ,  $\beta$ ,  $\Phi$ , and  $\Theta$ .
4. (exploratory factor analysis only) Call `Rotate` to choose a transformation matrix (**T**) for the factors. There is an extensive GUI that pops up when `Rotate` is called to guide you through this step. Again, `Rotate` differs fundamentally from other approaches to factor rotation in that it permits you to impose inequality restrictions on functions of parameters when searching for **T**.
5. Call the usual post-estimation methods to interpret the estimates, like `summary`, `pairs`, `FA-method`, etc., and call `model_comparison` to see the test statistics and fit indices.

The vignette has additional information regarding the pop-up menus produced in step 2 by `make_restrictions` and in step 4 by `Rotate`; execute `vignette("FAiR")` to read it. The primary examples are in `Factanal` and `Rotate`.

#### Author(s)

Ben Goodrich

**Examples**

```
## See the examples for Factanal() and Rotate()
```

---

```
create_FAobject      Class "FA" and Its Constructors
```

---

**Description**

It is not necessary to understand this help page if one merely wants to estimate a factor analysis model. This help page is intended for those who want to modify or extend FAiR or otherwise want some idea of how **FAiR** works “behind the scenes”.

The classes that inherit from "FA" encapsulate estimates from factor analysis models. First, the constructor(s) will be discussed.

**Usage**

```
## S4 method for signature 'restrictions, manifest.basic':
create_FAobject(restrictions, manifest, opt, call, scores, lower,
                analytic)
```

**Arguments**

restrictions	object of <a href="#">restrictions-class</a>
manifest	object of <a href="#">manifest-class</a>
opt	The list produced by <a href="#">genoud</a>
call	the call to <a href="#">Factanal</a>
scores	A character string indicating what kind of factor scores to calculate; see the same argument to <a href="#">Factanal</a>
lower	A small numeric scalar indicating the lower bound for positive definiteness or minimum uniqueness; see the corresponding argument to <a href="#">Factanal</a> .
analytic	A logical indicating whether analytic gradients were used.

**Details**

The methods for `create_FAobject` are called internally right at the end of [Factanal](#). They take the result of the optimization and produce an object that inherits from class "FA", which is conceptually simple, although the implementation is somewhat complicated and relies on a bunch of helper functions that are not exported.

**Value**

The methods for `create_FAobject` produce an object of class "FA" or that inherits from class "FA" as appropriate.

## Objects from the Class

Objects can be created by calls of the form `new("FA", ...)`. However, this use of `new("FA", ...)` is not recommended because both `Factanal` and `Rotate` provide constructors for users with the help of the formal methods defined for `create_FAobject`.

## Slots

The "FA" class is not virtual but does serve as the basis for some inherited classes. Its slots are:

**loadings** A numeric array with as many rows as there are outcome variables, as many columns as there are (first-order) factors, and five shelves. Each shelf is thus a matrix and contains the estimated primary pattern (PP), primary structure (PS), reference pattern (RP), reference structure (RS), and factor contribution (FC) matrices respectively with the dimname indicated in parentheses.

**correlations** A numeric array with as many rows and columns as there are (first-order) factors and three shelves. Each shelf is a correlation matrix and contains the estimated correlations among the primary factors (PF), among reference factors (RF), and between each primary factor and its corresponding reference factor (PR) respectively with the dimname indicated in parentheses.

**uniquenesses** A numeric vector with as many elements as there are outcome variables and contains the estimated unique variances from the factor analysis model.

**scale** A numeric vector with as many elements as there are outcome variables and contains the estimated standard deviations of the outcome variables in the factor analysis model.

**restrictions** Object that inherits from `restrictions-class`

**Jacobian** A numeric matrix that contains the derivatives of the lower triangle of the reproduced covariance matrix with respect to each free parameter (by column) at their estimates.

**vcov** A square, numeric matrix that is the estimated variance-covariance matrix among the estimated parameters

**scores** A numeric matrix, possibly with zero rows. If, in the call to `Factanal` the user specifies that factor scores should be calculated, this matrix contains the factor scores and will have as many rows as there are observations and as many columns as there are (first-order) factors.

**manifest** Object that inherits from `manifest-class`

**optimization** A list that contains what is returned by the underlying optimization algorithm when called internally by `Factanal` and `Rotate`.

**call** This slot contains the call to `Factanal`.

**seeds** A numeric matrix with two columns and either one or two rows. The first row contains the `unif.seed` used by `genoud` and the `int.seed` in the call to `Factanal`. If `Rotate` is used to transform the factors after preliminary factors have been extracted as part of exploratory factor analysis, this matrix has a second row containing `unif.seed` and `int.seed` used in the call to `Rotate`.

An object of "FA.EFA" inherits from the "FA" class and has the following additional slots:

**rotated** Logical indicating whether the loadings have been rotated

**Lambda** A numeric matrix with the preliminary factor loadings

**trans\_mats** A numeric array with as many rows and columns as there are factors and three shelves. Its first shelf is called "primary" and contains the transformation matrix that postmultiplies `Lambda` to yield the rotated primary pattern matrix. Its second shelf is called "reference" and contains the transformation matrix that postmultiplies `Lambda` to yield the rotated reference structure matrix. Its third shelf is called "T" and contains the matrix whose `crossprod` is the correlation matrix among primary factors. If `rotated = FALSE`, then all of these transformation matrices are identity matrices.

An object of "FA.general" inherits from the "FA" class and its `restrictions` slot has an object of `restrictions.general-class`. It also has the following additional slots:

**loadings\_2nd** A numeric array that has the same form as that in the `loadings` slot. However, since there is only one second-order factor, there is no distinction among the various pattern and structure matrices. The factor contribution matrix in the fifth shelf is simply the square of these loadings

**uniquenesses\_2nd** A numeric vector giving the uniquenesses at level two

An object of "FA.2ndorder" inherits from the "FA.general" class and its `restrictions` slot has an object of `restrictions.2ndorder-class`. It also has an additional slot,

**correlations\_2nd:** a numeric array whose form is similar to that in the `correlations` slot but obviously pertains to the correlations among second-order factors rather than first-order factors.

## Methods

The `create_FAobject` methods *construct* an object that inherits from class "FA" but their signatures hinge on `restrictions-class` and (in the future) `manifest-class`. The following methods are defined for the various classes that inherit from "FA":

```
BIC signature(object = "FA")
coef signature(object = "FA")
confint signature(object = "FA")
logLik signature(object = "FA")
plot signature(x = "FA", y = "ANY")
profile signature(fitted = "FA")
show signature(object = "FA")
summary signature(object = "FA")
vcov signature(object = "FA")
```

In addition, the following methods are defined for classes that inherit from class "FA" that are documented in `loadings` or `S3methodsFAiR` but are also listed here:

**loadings** extracts various "loadings" matrices, use instead of `coef`

**cormat** extracts various correlation matrices

**uniquenesses** extracts uniquenesses

**pairs** Thurstone-style `pairs` plot among reference structure correlations taken two reference factors at a time

**fitted** the covariance or correlation matrix as reproduced by the model  
**residuals** covariance or correlation residuals  
**rstandard** covariance residuals normalized by manifest standard deviations  
**weights** (approximate) weight matrices for each correlation  
**influence** a weighted residuals matrix  
**df.residual** extract the degrees of freedom  
**deviance** the value of the discrepancy function (scaled by one less the number of observations)  
**model.matrix** extract that sample covariance or correlation matrix  
**simulate** simulated draws from the sampling distribution of the covariance or correlation matrix as reproduced by the model

In addition, the following functions are *not* S4 generics but nevertheless expect an object that inherits from "FA" class and will throw an error otherwise:

**model\_comparison** the [model\\_comparison](#) function produces test statistics and fit indices  
**paired\_comparison** the [paired\\_comparison](#) function tests one model against another model in which it is nested  
**FA2draws** the [FA2draws](#) function is essentially a wrapper around the [restrictions2draws](#) generic function but is more convenient  
**FA2RAM** the [FA2RAM](#) function is essentially a wrapper around the [restrictions2RAM](#) generic function but is more convenient  
**Rotate** the [Rotate](#) function finds an optimal transformation of preliminary factors in exploratory factor analysis  
**GPA2FA** the [GPA2FA](#) function requires an object of "FA.EFA" class

### Author(s)

Ben Goodrich

### See Also

[Factanal](#)

### Examples

```
showClass("FA")
showClass("FA.EFA")
showClass("FA.general")
showClass("FA.2ndorder")
```

---

equality\_restriction-class

*Class "equality\_restriction"*

---

## Description

It is not necessary to understand this help page if one merely wants to estimate a factor analysis model. This help page is intended for those who want to modify or extend FAiR or otherwise want some idea of how FAiR works “behind the scenes”.

This class defines an equality restriction for use in a semi-exploratory or confirmatory factor analysis model.

## Objects from the Class

Objects can be created by calls of the form `new("equality_restriction", ...)`. However, rarely if ever, would a user need to construct an object this way. The `make_restrictions` methods do so internally.

## Slots

**free:** an integer indicating which cell of a primary pattern matrix is considered “free”

**fixed:** an integer vector indicating which cells of a primary pattern matrix are constrained to be equal to the “free” cell

**dims:** an integer vector of length two indicating the dimensions of the primary pattern matrix under consideration

**rownames:** a character vector of rownames for the primary pattern matrix under consideration

**level:** either 1L or 2L indicating the level of the factor analysis model under consideration

## Methods

`show signature(object = "equality_restriction")`: Prints the equality restriction to the screen

## Author(s)

Ben Goodrich

## See Also

[restrictions.lstorder-class](#), [make\\_restrictions](#)

**Examples**

```

showClass("equality_restriction")

man <- make_manifest(covmat = ability.cov)

## Here is how to set up an equality restriction in a SEFA model the hard way
beta <- matrix(NA_real_, nrow = nrow(cormat(man)), ncol = 2)
rownames(beta) <- rownames(cormat(man))
beta[6,2] <- Inf # "fix" beta[6,2] to Inf provisionally

## Note 11L corresponds to beta[5,2] and 12L corresponds to beta[6,2]
ER <- new("equality_restriction", free = 11L, fixed = 12L, dims = dim(beta),
         rownames = rownames(cormat(man)), level = 1L)
free <- is.na(beta) # i.e. all but beta[6,2]
beta <- new("parameter.coef.SEFA", x = beta, free = free, num_free = sum(free),
          equalities = list(ER))

Phi <- diag(2)
free <- lower.tri(Phi)
Phi <- new("parameter.cormat", x = Phi, free = free, num_free = sum(free))

## Now set up a restriction to prohibit the equal coefficients from being zero
blockers <- matrix(FALSE, nrow = nrow(cormat(man)), ncol = 2)
blockers[5:6,2] <- TRUE # these two cells are hence not allowed to both be zero

res <- make_restrictions(manifest = man, beta = beta, Phi = Phi,
                        discrepancy = "MLE", criteria = list("block_1st"),
                        methodArgs = list(blockers = blockers))

## Not run:
## Here is the easy way to do the same thing, in the following pop-up menus
## select the options to impose equality restrictions and inequality
## restrictions (block those coefficients from being zero in a SEFA model)
res <- make_restrictions(manifest = man, factors = 2, model = "SEFA")
## End(Not run)
show(res)

```

**Description**

This function is intended for users and estimates a factor analysis model that has been set up previously with a call to [make\\_manifest](#) and a call to [make\\_restrictions](#).

**Usage**

```

Factanal(manifest, restrictions, scores = "none", seeds = 12345,
lower = sqrt(.Machine$double.eps), analytic = TRUE, reject = TRUE,
NelderMead = TRUE, impatient = FALSE, ...)

```

**Arguments**

<code>manifest</code>	An object that inherits from <code>manifest-class</code> and is typically produced by <code>make_manifest</code> .
<code>restrictions</code>	An object that inherits from <code>restrictions-class</code> and is typically produced by <code>make_restrictions</code> .
<code>scores</code>	Type of factor scores to produce, if any. The default is "none". Other valid choices (which can be partially matched) are "regression", "Bartlett", "Thurstone", "Ledermann", "Anderson-Rubin", "McDonald", "Krinjen", "Takeuchi", and "Harman". See Beauducel (2007) for formulae for these factor scores as well as proofs that all but "regression" and "Harman" produce the same correlation matrix.
<code>seeds</code>	A vector of length one or two to be used as the random number generator seeds corresponding to the <code>unif.seed</code> and <code>int.seed</code> arguments to <code>genoud</code> respectively. If <code>seeds</code> is a single number, this seed is used for both <code>unif.seed</code> and <code>int.seed</code> . These seeds override the defaults for <code>genoud</code> and make it easier to replicate an analysis exactly. If NULL, the default arguments for <code>unif.seed</code> and <code>int.seed</code> as specified in <code>genoud</code> are used. NULL should be used in simulations or else they will be horribly wrong.
<code>lower</code>	A lower bound. In exploratory factor analysis, <code>lower</code> is the minimum uniqueness and corresponds to the 'lower' element of the list specified for <code>control</code> in <code>factanal</code> . Otherwise, <code>lower</code> is the lower bound used for singular values when checking for positive-definiteness and ranks of matrices. If the unlikely event that you get errors referencing positive definiteness, try increasing the value of <code>lower</code> slightly.
<code>analytic</code>	A logical (default to TRUE) indicating whether analytic gradients should be used as much as possible. If FALSE, then numeric gradients will be calculated, which are slower and slightly less accurate but are necessary in some situations and useful for debugging analytic gradients.
<code>reject</code>	Logical indicating whether to reject starting values that fail the constraints required by the model; see <code>create_start</code>
<code>NelderMead</code>	Logical indicating whether to call <code>optim</code> with <code>method = "Nelder-Mead"</code> when the genetic algorithm has finished to further polish the solution. This option is not relevant or necessary for exploratory factor analysis models.
<code>impatient</code>	Logical that defaults to FALSE. If <code>restrictions</code> is of <code>restrictions.factanal-class</code> , setting it to TRUE will cause <code>factanal</code> to be used for optimization instead of <code>genoud</code> . In all other situations, setting it to TRUE will use <code>factanal</code> to generate initial communality estimates instead of the slower default mechanism.
<code>...</code>	Further arguments that are passed to <code>genoud</code> . The following arguments to <code>genoud</code> are hard-coded and cannot be changed because they are logically required by the factor analysis estimator:

<b>argument</b>	<b>value</b>	<b>why?</b>
<code>nvars</code>	<code>restrictions@nvars</code>	
<code>max</code>	FALSE	minimizing the objective

<code>hessian</code>	FALSE	we roll our own
<code>lexical</code>	TRUE (usually)	for restricted optimization
<code>Domains</code>	<code>restrictions@Domains</code>	
<code>data.type.int</code>	FALSE	parameters are doubles
<code>fn</code>	wrapper around <code>fitS4</code>	
<code>BFGSfn</code>	wrapper around <code>bfgs_fitS4</code>	
<code>BFGShelp</code>	wrapper around <code>bfgs_helpS4</code>	
<code>gr</code>	various	it is complicated
<code>unif.seed</code>	taken from <code>seeds</code>	replicability
<code>int.seed</code>	taken from <code>seeds</code>	replicability

The following arguments to `genoud` default to values that differ from those documented at `genoud` but can be overridden by specifying them explicitly in the ...:

argument	value	why?
<code>boundary.enforcement</code>	1 usually	2 can cause problems
<code>MemoryMatrix</code>	FALSE	runs faster
<code>print.level</code>	1	output is not that helpful for $\geq 2$
<code>P9mix</code>	1	to always accept the BFGS result
<code>BFGSburnin</code>	-1	to delay the gradient check
<code>max.generations</code>	1000	big number is often necessary
<code>project.path</code>	contains <code>"Factanal.txt"</code>	
<code>starting.values</code>	see the Details section	

Other arguments to `genoud` will take the documented default values unless explicitly specified. In particular, you may want to change `wait.generations` and `solution.tolerance`. Also, if informative bounds were placed on any of the parameters in the call to `make_restrictions` it is usually preferable to specify that `boundary.enforcement = 2` to use constrained optimization in the internal calls to `optim`. However, the "L-BFGS-B" optimizer is less robust than the default "BFGS" optimizer and occasionally causes fatal errors, largely due to misfortune.

## Details

The call to `Factanal` is somewhat of a formality in the sense that most of the difficult decisions were already made in the call to `make_restrictions` and the call to `make_manifest`. The most important remaining detail is the specification of the values for the starting population in the genetic algorithm.

It is not necessary to provide starting values, since there are methods for this purpose; see `create_start`. Also, if `starting.values = NA`, then a population of starting values will be created using the typical mechanism in `genoud`, namely random uniform draws from the domain of the parameter.

Otherwise, if `reject = TRUE`, starting values that fail one or more constraints are rejected and new vectors of starting values are generated until the population is filled with admissible starting values. In some cases, the constraints are quite difficult to satisfy by chance, and it may be more practical to specify `reject = FALSE` or to supply starting values explicitly. If starting values are supplied, it is helpful if at least one member of the genetic population satisfies all the constraints

imposed on the model. Note the rownames of `restrictions@Domains`, which indicate the proper order of the free parameters.

A matrix (or vector) of starting values can be passed as `starting.values`. (Also, it is possible to pass an object of `FA-class` to `starting.values`, in which case the estimates from the previous call to `Factanal` are used as the starting values.) If a matrix, it should have columns equal to the number of rows in `restrictions@Domains` in the specified order and one or more rows up to the number of genetic individuals in the population.

If `starting.values` is a vector, its length can be equal to the number of rows in `restrictions@Domains` in which case it is treated as a one-row matrix, or its length can be equal to the number of manifest variables, in which case it is passed to the `start` argument of `create_start` as a vector of initial communality estimates, thus avoiding the sometimes time-consuming process of generating good initial communality estimates. This process can also be accelerated by specifying `impatient = TRUE`.

### Value

An object of that inherits from `FA-class`.

### Note

The underlying genetic algorithm can print a variety of output as it progresses. On Windows, you either have to move the scrollbar periodically to flush the output to the screen or disable buffering by either going to the Misc menu or by clicking Control+W. The output will, by default, look something like this

Generation number	First constraint	Second constraint	...	Last constraint	Discrepancy function
0	-1.0	-1.0	...	-1.0	double
1	-1.0	-1.0	...	-1.0	double
...	...	...	...	...	...
42	-1.0	-1.0	...	-1.0	double

The integer on the far left indicates the generation number. If it appears to skip one or more generations, that signifies that the best individual in the “missing” generation was no better than the best individual in the previous generation. The sequence of `-1.0` indicates that various constraints are being satisfied by the best individual in the generation. Some of these constraints are hard-coded, some are added by the choices the user makes in the call to `make_restrictions`. The curious are referred to the source code, but for the most part users need not worry about them provided they are `-1.0`. If any but the last are not `-1.0` after the first few generations, there is a **major** problem because no individual is satisfying all the constraints. The last number is a double-precision number indicating the value of the discrepancy function. This number will decrease, sometimes painfully slowly, sometimes intermittently, over the generations since the discrepancy function is being minimized, subject to the aforementioned constraints.

### Author(s)

Ben Goodrich

## References

- Bartholomew, D. J. and Knott, M. (1990) *Latent Variable Analysis and Factor Analysis*. Second Edition, Arnold.
- Beauducel, A. (2007) In spite of indeterminacy, many common factor score estimates yield an identical reproduced covariance matrix. *Psychometrika*, **72**, 437–441.
- Smith, G. A. and Stanley G. (1983) Clocking *g*: relating intelligence and measures of timed performance. *Intelligence*, **7**, 353–368.
- Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*. Fourth edition. Springer.

## See Also

[make\\_manifest](#), [make\\_restrictions](#), and [Rotate](#)

## Examples

```
## Example from Venables and Ripley (2002, p. 323)
## Previously from Bartholomew and Knott (1999, p. 68--72)
## Originally from Smith and Stanley (1983)
## Replicated from example(ability.cov)

man <- make_manifest(covmat = ability.cov)

## Not run:
## Here is the easy way to set up a SEFA model, which uses pop-up menus
res <- make_restrictions(manifest = man, factors = 2, model = "SEFA")
## End(Not run)

## This is the hard way to set up a restrictions object without pop-up menus
beta <- matrix(NA_real_, nrow = nrow(cormat(man)), ncol = 2)
rownames(beta) <- rownames(cormat(man))
free <- is.na(beta)
beta <- new("parameter.coef.SEFA", x = beta, free = free, num_free = sum(free))

Phi <- diag(2)
free <- lower.tri(Phi)
Phi <- new("parameter.cormat", x = Phi, free = free, num_free = sum(free))
res <- make_restrictions(manifest = man, beta = beta, Phi = Phi)

# This is how to make starting values where Phi is the correlation matrix
# among factors, beta is the matrix of coefficients, and the scales are
# the logarithm of the sample standard deviations. It is also the MLE.
starts <- c( 4.46294498156615e-01, # Phi_{21}
            4.67036349420035e-01, # beta_{11}
            6.42220238211291e-01, # beta_{21}
            8.88564379236454e-01, # beta_{31}
            4.77779639176941e-01, # beta_{41}
            -7.13405536379741e-02, # beta_{51}
            -9.47782525342137e-08, # beta_{61}
            4.04993872375487e-01, # beta_{12}
            -1.04604290549591e-08, # beta_{22}
```

```

-9.44950629176182e-03, # beta_{32}
2.63078925240678e-04, # beta_{42}
9.38038168787216e-01, # beta_{52}
8.43618801925473e-01, # beta_{62}
log(man@sds)          # log manifest standard deviations

sefa <- Factanal(manifest = man, restrictions = res,
  # NOTE: Do NOT specify any of the following tiny values in a
  # real research situation; it is done here solely for speed
  starting.values = starts, pop.size = 2, max.generations = 6,
  wait.generations = 1)
nsim <- 101 # number of simulations, also too small for real work
show(sefa)
summary(sefa, nsim = nsim)
model_comparison(sefa, nsim = nsim)

stuff <- list() # output list for various methods
stuff$model.matrix <- model.matrix(sefa) # sample correlation matrix
stuff$fitted <- fitted(sefa, reduced = TRUE) # reduced covariance matrix
stuff$residuals <- residuals(sefa) # difference between model.matrix and fitted
stuff$rstandard <- rstandard(sefa) # normalized residual matrix
stuff$weights <- weights(sefa) # (scaled) approximate weights for residuals
stuff$influence <- influence(sefa) # weights * residuals
stuff$cormat <- cormat(sefa, matrix = "RF") # reference factor correlations
stuff$uniquenesses <- uniquenesses(sefa, standardized = FALSE) # uniquenesses
stuff$FC <- loadings(sefa, matrix = "FC") # factor contribution matrix
stuff$draws <- FA2draws(sefa, nsim = nsim) # draws from sampling distribution

if(require(nFactors)) screepLOT(sefa) # Enhanced scree plot
profile(sefa) # profile plots of non-free parameters
pairs(sefa) # Thurstone-style plot
if(require(Rgraphviz)) plot(sefa) # DAG

```

---

GPA2FA

*Covert Result of Gradient Projection Algorithm*


---

## Description

This utility function can be used in an exploratory factor analysis when the factors are extracted via [Factanal](#) but transformed using the [GPFoblq](#) or [GPFoblq](#) functions in the suggested **GPArotation** package. This function simply synthesizes the results of each to produce an object of [FAclass](#) that will be recognized by the various post-estimation methods and other functions.

## Usage

```
GPA2FA(GPAobject, FAobject)
```

**Arguments**

GPAobject	the result of a call to one of the transformation functions in <b>GPArotation</b> (see <a href="#">GPA</a> ) with <code>A = loadings(FAobject)</code>
FAobject	an object of <a href="#">FA.EFA-class</a> produced by <a href="#">Factanal</a>

**Details**

In some cases, it may be preferable to use the gradient projection algorithm in the **GPArotation** package rather than the genetic algorithm used by [Rotate](#). The gradient projection algorithm is faster, simpler, and allows for the possibility of orthogonal rotation. If the gradient projection algorithm is used, then this function permits a seamless transition from an object of S3 class "GPArotation" back to a S4 object of [FA.EFA-class](#) so that the rest of the post-estimation methods and functions defined in **FAiR** can be used on the result.

[Rotate](#) does not do orthogonal rotation for computational and philosophical reasons but the oblique criteria in **GPArotation** have now been copied into **FAiR** so that they can be utilized by [Rotate](#). The genetic algorithm permits constrained optimization, which makes it possible to avoid factor collapse and more generally to optimize with respect to the intersection of a set of criteria. Also, [Rotate](#) allows one to use the reference structure or factor contribution matrix instead of the primary pattern matrix when optimizing with respect to one of the criteria in **GPArotation**.

**Value**

An object of [FA.EFA-class](#) with transformed loadings.

**Author(s)**

Ben Goodrich

**See Also**

[GPA](#)

**Examples**

```
# step 1: extract factors with Factanal(), rather than factanal()
man <- make_manifest(covmat = ability.cov)
res <- make_restrictions(man, factors = 2, model = "EFA")
efa <- Factanal(manifest = man, restrictions = res, impatient = TRUE)

# steps 2 and 3: transform with quartimin() and then synthesize objects
if( require(GPArotation) ) {
  efa.GPA <- quartimin(loadings(efa))
  efa.Rotated <- GPA2FA(efa.GPA, efa)
}
if(!require(GPArotation)) { # steps 2 and 3 are equivalent to this
  efa.Rotated <- Rotate(efa, criteria = list("quartimin"), methodArgs =
    list(nfc_threshold = 0, matrix = "PP"))
}

# step 4: interpret with various commands, such as
```

```
summary(efa.Rotated)
```

---

```
loadings
```

```
Extractor ("get") functions
```

---

## Description

These functions extract various elements of formal S4 objects that are important in factor analysis models, namely the loadings, the correlations among factors, and the unique variances. Occasionally, it may be useful to call these generic functions directly.

## Usage

```
## S4 method for signature 'FA':
coef(object)
## S4 method for signature 'restrictions':
coef(object)
## S4 method for signature 'FA':
loadings(x, matrix = "PP", standardized = TRUE)
## S4 method for signature 'FA.general':
loadings(x, matrix = "PP", standardized = TRUE, level = 1)
## S4 method for signature 'restrictions.general':
loadings(x, standardized = TRUE, level = 1)
## S4 method for signature 'FA':
cormat(object, matrix = "PF")
## S4 method for signature 'FA.2ndorder':
cormat(object, matrix = "PF", level = 1)
## S4 method for signature 'restrictions':
cormat(object)
## S4 method for signature 'restrictions.2ndorder':
cormat(object, level = 1)
## S4 method for signature 'FA':
uniquenesses(object, standardized = TRUE)
## S4 method for signature 'FA.general':
uniquenesses(object, standardized = TRUE, level = 1)
## S4 method for signature 'restrictions':
uniquenesses(object, standardized = TRUE)
```

## Arguments

object	an object that inherits from <code>FA-class</code> or <code>restrictions-class</code>
x	an object that inherits from <code>FA-class</code> or <code>restrictions-class</code>
matrix	a character string with exactly two letters indicating which matrix to extract; see the <code>Details</code> section
standardized	a logical indicating whether to standardize the result so that it is calibrated for a correlation matrix among manifest variables, rather than their covariance matrix
level	either 1 or 2 to indicate from which level of the factor analysis model is pertinent when the model has two levels

## Details

Let the factor analysis model be

$$\Sigma = \Omega(\beta\Phi\beta' + \Theta)\Omega$$

By default, the `loadings` methods extract the estimate of  $\beta$ , the `cormat` methods extract the estimate of  $\Phi$ , and the `uniquenesses` methods extract the diagonal of  $\Theta$ . In addition, the `coef` methods and the `loadings` methods that are defined for objects `restrictions-class` extract the primary pattern matrix (at level 1).

At the moment there is no special function to get the diagonal of  $\Omega$ , which is a diagonal matrix of *estimated* standard deviations of the manifest variables. However, they can be extracted from the appropriate slot using the `@` operator. Also, if `standardized = FALSE` in the call to `loadings` or `uniquenesses`, then the `loadings` or `uniquenesses` are scaled by these estimated standard deviations to produce estimates on the covariance metric.

Additionally, for the `loadings` and `cormat` methods that are defined on objects of `FA-class`, the `matrix` argument can be specified to extract a different set of estimated coefficients or correlations. By default, `matrix = "PP"` for these `loadings` methods, indicating that the primary pattern matrix should be extracted. Other possible choices are `"PS"` to extract the primary structure matrix (defined as  $\beta\Phi$ ), `"RS"` to extract the reference structure matrix (which is column-wise proportional to  $\beta$ ), `"RP"` to extract the reference pattern matrix (which is column-wise proportional to  $\beta\Phi$ ), and `"FC"` to extract the factor contribution matrix (which is defined as  $\beta * (\beta\Phi)$ , where the `*` indicates element-by-element multiplication of two matrices with the same dimensions).

By default, `matrix = "PF"` for these `cormat` methods, indicating that the correlation matrix among primary factors should be extracted. Other possible choices are `"RF"` to extract the correlation matrix among reference factors and `"PR"` to extract the (diagonal) correlation matrix between primary and reference factors.

In the case of a two-level model, the `level` argument can be specified to extract such matrices from the second level of the model (including the methods for the `uniquenesses` generic).

## Value

`loadings` outputs a matrix of S3 class `"loadings"`, which has a special print method (see `print.loadings`). `coef` returns the primary pattern matrix at level one and is not of class `"loadings"`. The `cormat` methods output a (symmetric) matrix, and the `uniquenesses` methods output a non-negative numeric vector.

## Methods

There are methods for every flavor of `FA-class` and virtually all flavors of `restrictions-class`. Also, in the code of `cormat`, there is a method for objects that inherit from `manifest-class`.

## Author(s)

Ben Goodrich

## Examples

```
## See the example for Factanal()
```

---

make\_manifest      *Make an object that inherits from class "manifest"*

---

## Description

This function is intended for users and sets up the left-hand side of the factor analysis model and is a prerequisite for calling [make\\_restrictions](#) and [Factanal](#).

Although it is possible to simply estimate and use the unbiased sample covariance matrix, there are many other ways to estimate a covariance that can be superior, particularly when the traditional maximum likelihood discrepancy function is not chosen in the call to [make\\_restrictions](#).

In technical terms, `make_manifest` is the constructor for objects of `manifest-class`, which houses the sample covariance estimate and some ancillary information in its slots. The three arguments in the signature of the S4 generic function are:

`x`, `data`, and `covmat`

## Usage

```
## S4 method for signature 'missing, missing, list':
make_manifest(covmat, n.obs = NA_integer_, shrink = FALSE)
## S4 method for signature 'missing, missing, hetcor':
make_manifest(covmat, shrink = FALSE)
## S4 method for signature 'missing, missing, matrix':
make_manifest(covmat, n.obs = NA_integer_, shrink = FALSE, sds = NULL)
## S4 method for signature 'missing, missing, CovMcd':
make_manifest(covmat)

# Use the methods above when only the covariance matrix is available
# Use the methods below when the raw data are available (preferable)

## S4 method for signature 'data.frame, missing, missing':
make_manifest(x, subset, shrink = FALSE,
bootstrap = 0, how = "default", seed = 12345, wt = NULL, ...)

## S4 method for signature 'missing, data.frame, missing':
make_manifest(data, subset, shrink = FALSE,
bootstrap = 0, how = "default", seed = 12345, wt = NULL, ...)

## S4 method for signature 'missing, matrix, missing':
make_manifest(data, subset, shrink = FALSE,
bootstrap = 0, how = "default", seed = 12345, wt = NULL, ...)

## S4 method for signature 'matrix, missing, missing':
make_manifest(x, subset, shrink = FALSE,
bootstrap = 0, how = "default", seed = 12345, wt = NULL, ...)

## S4 method for signature 'formula, data.frame, missing':
```

```
make_manifest(x, data, subset, shrink = FALSE, na.action = "na.pass",
             bootstrap = 0, how = "default", seed = 12345, wt = NULL, ...)
```

### Arguments

<code>x</code>	a formula, data.frame, nonsquare matrix of observations by variables, or missing. If a formula, then <code>data</code> must be a data.frame and the formula should not have a response. If a data.frame or a matrix of data, then all its columns are used.
<code>data</code>	a data.frame, nonsquare matrix of observations by variables, or missing. If a data.frame and <code>formula</code> is not specified, then all its columns are used and similarly if it is a matrix of data.
<code>covmat</code>	A covariance matrix, a list, an object of <code>CovMcd-class</code> , an object of S4 class <code>"hetcor"</code> from the <b>polycor</b> package, or missing. If a list, it must contain an element named <code>"cov"</code> and may contain the following named elements: <b>n.obs</b> the number of observations used in calculating the <code>"cov"</code> element <b>W</b> a positive definite matrix to be used as a weight matrix in the ADF discrepancy function. However, the <code>make_restrictions-methods</code> can calculate various weight matrices if the raw data are passed to <code>make_manifest</code> , so this mechanism should only be used if those options are inadequate <b>sds</b> a numeric vector of standard deviations to be used if <code>"cov"</code> is really a correlation matrix
<code>n.obs</code>	The number of observations, which is used if <code>covmat</code> is a covariance matrix or if <code>covmat</code> is a list with no element named <code>n.obs</code> . It is possible to obtain maximum likelihood estimates without knowing the number of observations but nothing else
<code>shrink</code>	A logical indicating whether to use a “shrinkage” estimator of the covariance matrix. If <code>TRUE</code> , then the “minimax shrinkage” estimator discussed in theorem 3.1 of Dey and Srinivasan (1985) is invoked on the sample covariance matrix as calculated according to the other arguments. In some circumstances, <code>shrink</code> is inappropriate and ignored with a warning
<code>sds</code>	Either <code>NULL</code> or a numeric vector that contains the standard deviations of the manifest variables, which is used when <code>covmat</code> is a correlation matrix
<code>subset</code>	A specification of the cases to be used
<code>bootstrap</code>	A nonnegative integer (defaulting to zero) indicating how many bootstraps to do when estimating the uncertainty of the sample covariance estimates.
<code>how</code>	A character string indicating how the covariance matrix should be estimated; see the Details section
<code>seed</code>	A vector of length at most one to be used as the random number generator seed if <code>how = "mcd"</code> or <code>bootstrap &gt; 0</code> . If <code>NULL</code> , then the current seed is used. This argument defaults to 12345.
<code>wt</code>	An <b>optional</b> numeric vector of weights that is the same length as the number of observations that indicates the weight for each observation when <code>x</code> is specified. By default, the observations are weighted equally. The <code>wt</code> argument can be used in two ways. First, it is passed to the the corresponding argument of <code>cov.wt</code> if

appropriate (see below). Second, it is passed to the `prob` argument of `sample` when `bootstrap > 0`.

`na.action` The `na.action` to be used if `x` is a formula.

`...` Further arguments that are passed to downstream functions when `covmat` is unspecified, implying that the raw data are being used to estimate the sample covariance.

## Details

The rules governing the calculation of the sample covariance matrix are as follows and primarily depend on whether any of the manifest variables are ordered factors. First, consider the case where all manifest variables are numeric. If any of these manifest variables contain missing values, then the covariance matrix is estimated via maximum likelihood under multivariate normality assumptions but requires the suggested `mvnml` package. Otherwise, the `how` argument dictates how the covariance matrix is estimated. There is much to be said in favor the Minimum Covariance Determinant (`CovMcd`) estimator (see Pison et. al. 2003) and it is used as the default when there are no missing data, although it can subtly affect the sampling distributions of estimates that subsequently derived from it. The same could probably be said for the shrinkage estimators (either via `how = "lambda"` or `shrink = TRUE`). The Dey and Srinivasan (1985) shrinkage estimator preserves the eigenvectors of the preliminarily-calculated covariance matrix but deterministically compresses the eigenvalues. The `cov.shrink` estimator in the `corpcor` package is based on the idea that the amount shrinkage should be proportional to the variance of the covariance estimates. Use `how = "mle"` or `how = "unbiased"` to obtain either the maximum likelihood or unbiased sample covariance estimator, the latter of which is the one used in virtually all factor applications whether appropriate or not.

Next, consider the case where at least one manifest variable is an ordered factor. If `how = "ranks"`, Spearman correlations are estimated from the integer codes underlying the ordered factors. This mechanism is recommended only if there are at least five levels of each ordered factor and no missing data. In that case, one would presumably want to specify `method = "ADF"` in the subsequent call to `make_restrictions`). If `how != "ranks"` all pairwise correlations are estimated under bivariate normality assumptions via `hetcor` in the suggested `polycor` package, which will allow pairwise-deletion when there are missing data. If `how != "ranks"` and `bootstrap > 0` (recommended), then there must not be any missing data because the bootstrapping utilizes fast Spearman correlations and then tries to correct the bias by rescaling the bootstrapped means to equal to point estimates calculated with the call to `hetcor`.

In general, bootstrapping is good for estimating the uncertainty of the estimated sample covariances and this uncertainty estimate is needed for the ADF discrepancy function and its special cases. In some cases, bootstrapping is the only way to obtain such an uncertainty estimate.

## Value

An object that inherits from `manifest-class`.

## Author(s)

Ben Goodrich

## References

Dey, D. K. and Srinivasan K. (1985) Estimation of a covariance matrix under Stein's loss. *The Annals of Statistics*, **13**, 1581–1591.

Pison, G., Rousseeuw, P.J., Filzmoser, P. and Croux, C. (2003) Robust factor analysis. *Journal of Multivariate Analysis*, **84**, 145–172.

## See Also

[Factanal](#), [make\\_restrictions](#), [manifest-class](#), [covMcd](#), [cov.wt](#), [hetcor](#), [mlest](#), [cov.shrink](#), and [cov](#).

## Examples

```
man <- make_manifest(covmat = Harman23.cor)
show(man)          # some basic info
if(require(nFactors)) screepLOT(man) # advanced Scree plot
cormat(man)        # sample correlation matrix
```

---

make\_restrictions *Make an object of class "restrictions"*

---

## Description

This function is intended for users and sets up the restrictions to be imposed on the factor analysis model, which includes specifying whether an exploratory, confirmatory, or semi-exploratory model is to be estimated and what discrepancy function to use. It is a prerequisite for calling [Factanal](#) and assumes you have already called [make\\_manifest](#).

However, it is not necessary to understand the function arguments in great detail because most of the functionality is implemented via pop-up menus. The vignette provides the “substantive” documentation for this function and provides screenshots of the pop-up menus; execute `vignette("FAiR")` to view it. It is possible (though not recommended in normal usage) to avoid the pop-up menus entirely, in which case it is necessary to thoroughly understand the documentation here and in the vignette as well as the [restrictions-class](#) definition.

In technical terms, this S4 generic function is a constructor for objects of [restrictions-class](#). The arguments in the signature of the S4 generic function are:

`manifest`, `Omega`, `beta`, `Phi`, `Delta`, and `Xi`

The first S4 method defined immediately below is the one intended for the vast majority of usage. It *requires* `manifest` to be specified, *forbids* these Greek letters from being specified, and has a few optional arguments that can be specified to modify the default behavior.

## Usage

```
## S4 method for signature 'manifest.basic, missing,
##   missing, missing, missing, missing':
make_restrictions(
manifest, factors = NULL, model = c("SEFA", "EFA", "CFA"),
```

```
discrepancy = "default", nl_1 = NULL, nl_2 = NULL)

# USE THE ABOVE METHOD IN MOST CASES!!!
# USE ONE OF THE FOLLOWING METHODS IN SIMULATIONS, ETC.

## S4 method for signature 'manifest.basic,
##   parameter.scale, missing, missing, missing, missing':
make_restrictions(
manifest, Omega, discrepancy = "default")

# Above method implies zero common factors

## S4 method for signature 'manifest.basic, missing,
##   parameter.coef, missing, missing, missing':
make_restrictions(
manifest, beta, discrepancy = "default")

# Above method implies EFA with orthogonal factors

## S4 method for signature 'manifest.basic, missing,
##   parameter.coef, parameter.cormat, missing, missing':
make_restrictions(
manifest, beta, Phi, discrepancy = "default",
criteria = list(), methodArgs = list())

# Adjacent methods imply correlated factors

## S4 method for signature 'manifest.basic,
##   parameter.scale, parameter.coef, parameter.cormat,
##   missing, missing':
make_restrictions(
manifest, Omega, beta, Phi, discrepancy = "default",
criteria = list(), methodArgs = list())

## S4 method for signature 'manifest.basic, missing,
##   parameter.coef, missing, parameter.coef, missing':
make_restrictions(
manifest, beta, Delta, discrepancy = "default",
criteria = list(), methodArgs = list())

# Adjacent methods imply one second-order factor

## S4 method for signature 'manifest.basic,
##   parameter.scale, parameter.coef, missing,
##   parameter.coef, missing':
make_restrictions(
manifest, Omega, beta, Delta, discrepancy = "default",
criteria = list(), methodArgs = list())
```

```
## S4 method for signature 'manifest.basic, missing,
##   parameter.coef, missing, parameter.coef,
##   parameter.cormat':
make_restrictions(
manifest, beta, Delta, Xi, discrepancy = "default",
criteria = list(), methodArgs = list())

# Adjacent methods imply multiple second-order factors

## S4 method for signature 'manifest.basic,
##   parameter.scale, parameter.coef, missing,
##   parameter.coef, parameter.cormat':
make_restrictions(
manifest, Omega, beta, Delta, Xi, discrepancy = "default",
criteria = list(), methodArgs = list())
```

## Arguments

manifest	a <i>required</i> object that inherits from <code>manifest.basic-class</code>
Omega	usually missing or else an object of <code>parameter.scale-class</code>
beta	usually missing or else an object that inherits from <code>parameter.coef-class</code>
Phi	usually missing or else an object of <code>parameter.cormat-class</code>
Delta	usually missing or else an object that inherits from <code>parameter.coef-class</code>
Xi	usually missing or else an object of <code>parameter.cormat-class</code>
factors	either NULL or a vector of length one or two indicating the number of factors at level one and level two
model	a character string among "SEFA" (default), "CFA", and "EFA" indicating whether to estimate a semi-exploratory, confirmatory, or exploratory model
discrepancy	a character string among "default", "MLE", "ADF", "ELLIPTICAL", "HK", "SHK", and "YWLS" indicating which discrepancy function to use. The default behavior is "ADF" if possible, otherwise "MLE". See the Note section for details.
nl_1	either NULL or a function with an argument called $x$ that imposes nonlinear exact restrictions on some cells of the primary pattern matrix at level one. See the Details section.
nl_2	either NULL or a function with an argument called $x$ that imposes nonlinear exact restrictions on some cells of the primary pattern matrix at level two. See the Details section.
criteria	either NULL or a list (possibly of length zero) of functions or character strings naming functions to be used as criteria in the lexical optimization process. See the Note section.
methodArgs	a list (possibly of length zero) of required arguments to the functions listed in the <code>criteria</code> argument. See the Note section.

## Details

Let the factor analysis model in the population be

$$\Sigma = \Omega(\beta\Phi\beta' + \Theta)\Omega$$

where  $\Sigma$  is the covariance matrix among outcome variables,  $\Omega$  is a diagonal matrix of standard deviations of the manifest variables,  $\beta$  is the primary pattern matrix (calibrated to standardized variables) with one column per factor,  $\Phi$  is the correlation matrix among the primary factors, and  $\Theta$  is the diagonal matrix of uniquenesses, which is fully determined by  $\beta$ ,  $\Phi$ , and the requirement that the matrix within parentheses has ones down its diagonal. Hence,  $\beta\Phi\beta' + \Theta$  is the model's correlation matrix among outcome variables as a function of the factors. This parameterization is often attributed to Cudeck (1989, equation 21) or to Krane and McDonald (1975) and will essentially yield the correct results even if the model is fit to a correlation matrix.

The `make_restrictions` methods set up the right-hand side of the factor analysis model, including the restrictions that are placed on the parameters. **FAiR** differs fundamentally from other factor analysis software in that you can place inequality restrictions on functions of multiple parameters, although this mechanism is not permitted during the factor extraction stage of an exploratory factor analysis. The classes that inherit from `restrictions-class` typically have slots that are Greek letters, which are objects that inherit from `parameter-class`. But the `restrictions-class` has additional slots that contain other information about the model.

Exploratory factor analysis (EFA) requires a minimal set of restrictions and permits no additional restrictions in the factor extraction stage. EFA preliminarily assumes the factors are orthogonal, i.e.  $\Phi = I$ , and either assumes that  $\beta'\Theta^{-1}\beta$  is diagonal or that  $\beta$  has all zeros in its upper triangle. However, after a transformation of the factors has been chosen (see `Rotate`), EFA takes no strong position on how many or which cells of  $\beta$  are (near) zero, unless target rotation or the simplimax criterion is used.

Confirmatory factor analysis (CFA) allows the user to choose which cells of  $\beta$  are exactly zero according to substantive theory and also permits other kinds of restrictions. Semi-exploratory factor analysis (SEFA) allows the user to choose how many cells in each column of  $\beta$  are zero but does not require the user to specify which cells are zero. SEFA also permits other kinds of restrictions, including constraining specific cells in  $\beta$  to be zero, as in a CFA.

For CFA and SEFA, it is optionally possible to estimate a “two-level” model where the correlation matrix among first-order factors is a function of one or more second-order factors. Hence, let the second-order factor analysis model in the population be

$$\Phi = \Delta\Xi\Delta' + \Gamma$$

where  $\Phi$  is the correlation matrix among first-order primary factors,  $\Delta$  is the second-order primary pattern matrix with one column per factor,  $\Xi$  is the correlation matrix among the second-order primary factors, and  $\Gamma$  is the diagonal matrix of second-order uniquenesses, which is fully determined by  $\Delta$ ,  $\Xi$ , and the requirement that  $\Phi$  has ones down its diagonal. Hence, in a two-level model,  $\Phi$  is restricted to be an exact function of  $\Delta$  and  $\Xi$  and the parameters for the two levels are estimated simultaneously.

**If you are at all unsure about what to do, use the first method listed in the Usage section, where "manifest" is the only required argument.** This method covers all the functionality of the other methods and will walk you through all the necessary steps using pop-up menus. However, if you would like to impose any nonlinear exact restrictions on the cells of  $\beta$  or  $\Delta$ , then you need to

define such functions (whose first argument should be called "x") in the global environment and specify them as the `nl_1` and / or `nl_2`. See also `parameter.coef.nl-class` for details.

The other methods will ask fewer questions via pop-up menus, and perhaps none at all (see the Note section below). Hence, they place more faith in the user to specify the additional arguments correctly. See the files in the FAiR/tests directory for many examples of setting up models the hard way without resorting to the pop-up menu system. You can stop reading this help page now if you are content with the pop-up menus.

If only "Omega" is specified, an object of `restrictions.independent-class` will emerge and has no common factors.  $\Omega$  is the only free parameter to estimate, which is only useful in constructing a null model to calculate some fit indices for another model (and all of this would normally be handled automatically by `model_comparison` anyway). If "Omega" is specified along with other arguments, then this object of `parameter.scale-class` is merely passed along and will become a slot of the resulting object that inherits from `restrictions-class`.

If "beta" is specified, but not "Phi" or "Delta", an object of `restrictions.orthonormal-class` will emerge, which is appropriate for EFA where the upper triangle of  $\beta$  contains all zeros. If one prefers the maximum-likelihood discrepancy function, there is a faster EFA algorithm that makes the assumption that  $\beta'\Theta^{-1}\beta$  is diagonal, which can be brought about by specifying `model = "EFA"` and `discrepancy = "MLE"` in the call to `make_restrictions` using the first method in the Usage section above. Doing so will produce an object of `restrictions.factanal-class`, whose other methods will reproduce the behavior of `factanal`.

If both "beta" and "Phi" are specified, but not "Delta", an object of `restrictions.1storder-class` will emerge, which is appropriate for CFA or SEFA when there are no second-order factors, implying that  $\Phi$  has no structure, other than that required of a correlation matrix. Whether the model is a SEFA depends on whether "beta" inherits from `parameter.coef.SEFA-class`.

If both "beta" and "Delta" are specified, but not "Xi", an object of `restrictions.general-class` will emerge, which is appropriate for CFA or SEFA with exactly one second-order factor. The off-diagonals of  $\Phi$  are  $\Delta\Delta'$ . Whether the model is SEFA (at level one) depends on whether "beta" inherits from `parameter.coef.SEFA-class` and "Delta" cannot inherit from `parameter.coef.SEFA-class`.

If "beta", "Delta", and "Xi" are specified, an object of `restrictions.2ndorder-class` will emerge, which is appropriate for CFA or SEFA with multiple second-order factors. Whether this two-level model is SEFA depends on whether "beta" and / or "Delta" inherit from `parameter.coef.SEFA-class`.

### Value

Returns an object that inherits from `restrictions-class`. This object would then be passed to the `restrictions` argument of `Factanal`.

### Note

In order to enforce inequality restrictions, it is necessary to use a genetic algorithm for lexical optimization (see `genoud`). Lexical optimization requires a sequence of piecewise functions, each of which evaluates whether the restriction is satisfied, and an ultimate continuous function, which in this case is a discrepancy function (see Browne (1984)).

In normal usage, the constraints would be specified in response to pop-up menus. In abnormal usage, the pop-up menus can be circumvented by specifying the constraints via the `criteria`

argument, whose elements are user-defined functions or character strings naming constraint functions, which are discussed in more detail below. The primary situation where the pop-up menus must be avoided is when you cannot rely on user-intervention, such as (some types of) Monte Carlo simulations, replication scripts, examples in these documentation pages, the files in FAiR/tests directory of the source code, etc. Conversely, there are templates to do virtually anything without resorting to the pop-up menus in the FAiR/tests directory and anyone who is interested should look there. Note also that when doing simulations, you should almost certainly be setting the `seeds` argument for various functions to `NULL` at every opportunity or your simulations will be invalid.

The discrepancy function is named by the `"discrepancy"` argument and is automatically appended to serve as the ultimate lexical criterion and should not be explicitly specified in `criteria` under any circumstances. The default behavior of the `"discrepancy"` argument depends on the exact class of the `manifest` argument. If `manifest` inherits from `"manifest.data"` or is of class `"manifest.basic.userW"`, the default discrepancy function is Browne's (1984) asymptotically distribution-free ("ADF") discrepancy function (in certain cases the weight matrix for the ADF discrepancy function is diagonal, which is often referred to as "diagonally weighted least squares"). Otherwise, this argument will default to `"MLE"`, which assumes the manifest variables are distributed multivariate normal.

Both `"ELLIPTICAL"` and `"HK"` are discrepancy functions that put further structure on the (inverse) weight matrix used by the `"ADF"` discrepancy function, where the former assumes there is a single kurtosis parameter and the latter assumes a different kurtosis for each manifest variable. The `"SHK"` discrepancy function differs from the `"HK"` discrepancy function in that the sample kurtosis estimates are shrunk towards the median kurtosis estimate using the same idea as in the `cov.shrink` shrinkage estimator of variance in the suggested `corpcor` package.

If `method = "YWLS"`, Yates' (1987 p.229) weighted least squares "discrepancy" function is used, which does not satisfy Browne's (1984) definition of a discrepancy function. This function has never received much scrutiny and is included in FAiR so that it can be fully evaluated. However, it does not lend itself to calculating standard errors or test statistics, and in limited testing seems prone to finding a solution that is geared more toward minimizing the weights than minimizing the squared residuals. Note that it also differs from the way in which "weighted least squares" is typically used in the factor analysis literature and in particular the LISREL program. "Weighted least squares" in LISREL essentially corresponds to `"ADF"` in FAiR.

User-defined functions can be passed as elements of `criteria` and will be called in the `environment` of the important but not exported function `FAiR:::FAiR_lexical_driver`, which is defined in `FAiR/R/Utils.R`. If `criteria` is a list that includes character strings, they should be names of one or more of the "canned" functions in the following table. Some of these functions have additional arguments that can be specified as elements of `methodArgs` to avoid seeing a pop-up menu asking for them. See the vignette for formal definitions of these restrictions and `methodArgs`.

<b>name</b>	<b>methodArgs</b>	<b>reminder of what function does</b>
<code>"ranks_rows_XXX"</code>	<code>row_ranks</code>	row-wise ordering constraints
<code>"ranks_cols_XXX"</code>	<code>col_ranks</code>	column-wise ordering constraints
<code>"indicators_XXX"</code>	<code>indicators</code>	designate which is the best indicator of a factor
<code>"evRF_XXX"</code>	<code>none</code>	restrict effective variance of reference factors
<code>"evPF_XXX"</code>	<code>none</code>	restrict effective variance of primary factors
<code>"h2_over_FC_XXX"</code>	<code>none</code>	communalities $\geq$ factor contributions
<code>"no_neg_suppressors_XXX"</code>	<code>FC_threshold</code>	no negative suppressors
<code>"gv_XXX"</code>	<code>none</code>	generalized variance of primary $\leq$ reference factors
<code>"distinguishability_XXX"</code>	<code>none</code>	best indicators have no negative suppressors

"cohyperplanarity_XXX"	none	hard to explain in this table
"dist_cols_XXX"	cutpoint	minimum binary distance between columns
"volume_1st"	none	$\det(R) \geq \det(S)$
"block_XXX"	blockers	force coefficients to be non-zero in SEFA

"XXX" can be either "1st" or "2nd" to indicate that the restriction should be applied at the first or second level of a model if there are two levels. However, `volume_1st` is defined only at the first level of the model. If, for example, one wanted to place row-wise restrictions on the order of the factor contributions at level one and level two of a model, then append "\_1st" and "\_2nd" to the corresponding elements of `methodArgs`.

### Author(s)

Ben Goodrich

### References

Browne, M.W. (1984), Asymptotically distribution-free methods for the analysis of covariance structures, *British Journal of Mathematical and Statistical Psychology*, **37**, 62–83.

Cudeck, R. (1989), Analysis of correlation matrices using covariance structure models, *Psychological Bulletin*, **105** 317–327.

Krane, W. R., and McDonald, R. P. (1978). Scale invariance and the factor analysis of covariance matrices. *British Journal of Mathematical and Statistical Psychology*, **31**, 218—228.

Yates, A. (1987) *Multivariate Exploratory Data Analysis: A Perspective on Exploratory Factor Analysis*. State University of New York Press.

### See Also

[Factanal](#) and [restrictions-class](#)

### Examples

```
man <- make_manifest(covmat = Harman74.cor)

## Not run:
## Here is the easy way to set up a two-level SEFA model using pop-up menus
res <- make_restrictions(manifest = man, factors = c(5,2), model = "SEFA")
## End(Not run)

## Here is the hard way to set up a two-level SEFA model,
## which eschews the pop-up menus; do NOT do this without good reason.
## There is an EFA example in ?Rotate
## There is a CFA example in ?restrictions2RAM
## There is an example with equality restrictions in ?equality_restriction-class

factors <- c(5,2)
beta <- matrix(NA_real_, nrow = nrow(cormat(man)), ncol = factors[1])
rownames(beta) <- rownames(cormat(man))
free <- is.na(beta)
beta <- new("parameter.coef.SEFA", x = beta, free = free, num_free = sum(free))
```

```

Delta <- matrix(NA_real_, nrow = factors[1], ncol = factors[2])
rownames(Delta) <- paste("F", 1:factors[1], sep = "")
free <- is.na(Delta)
Delta <- new("parameter.coef.SEFA", x = Delta, free = free, num_free = sum(free))

Xi <- diag(2)
free <- lower.tri(Xi)
## For fun, require the second-order factors to be positively correlated
uppers <- lowers <- Xi
lowers[2,1] <- 0
uppers[2,1] <- 1
Domains <- array(cbind(lowers, uppers), c(dim(Xi), 2))
Xi <- new("parameter.cormat", x = Xi, free = free, num_free = sum(free),
        Domains = Domains)

res <- make_restrictions(manifest = man, beta = beta, Delta = Delta, Xi = Xi)
show(res)

```

---

manifest-class      *Class "manifest"*

---

## Description

It is not necessary to understand this help page if one merely wants to estimate a factor analysis model. This help page is intended for those who want to modify or extend FAiR or otherwise want some idea of how **FAiR** works “behind the scenes”.

This class contains information in its slots about the left-hand side of a factor analysis model.

## Details

In the future, the advantages of formal S4 classes will be exploited more fully. For example, to include a new model in **FAiR**, the first step might be to define a new class that extends “manifest” if necessary. Then, appropriate methods for this class would need to be written. Here is a table of the class hierarchy:

Class	Parent Class	Comment
"manifest"	none	none (virtual class)
"manifest.basic"	"manifest"	Can be used for any model
"manifest.basic.userW"	"manifest.basic"	ADF with user-specified weight matrix
"manifest.data"	"manifest.basic"	Can be used for any model
"manifest.data.ordinal"	"manifest.data"	Used when some variables are ordinal
"manifest.data.ranks"	"manifest.data"	Used when variables are converted to their ranks
"manifest.data.mcd"	"manifest.data"	Can be used for any model

## Objects from the Class

Objects can be created by calls of the form `new("manifest", ...)`. However, it is not recommended to do so in normal usage since `make_manifest` provides the constructor for users.

## Slots

The "manifest" is virtual and has the following slots:

**n.obs** An integer indicating the number of observations.

**how** A character string indicating how the covariance matrix among the manifest variables was calculated.

**call** The call to `make_manifest`.

An object of class "manifest.basic" inherits from the "manifest" class and has five additional slots:

**cov** A sample covariance matrix among manifest variables.

**cor** A sample correlation matrix among manifest variables.

**sds** A numeric vector containing the square root of the diagonal of the sample covariance matrix.

**center** A numeric vector containing the mean of the manifest variables.

**diag** A logical indicating whether the diagonal elements of the sample "covariance" matrix are all 1.0.

An object of class "manifest.data" inherits from the "manifest.basic" class and has four additional slots:

**X** A numeric matrix of manifest variables (unstandardized).

**wt** A numeric vector whose length is the same as the number of rows in X indicating the weight of each observation in calculating the sample covariance matrix.

**acov** An object of `dsyMatrix-class` from the **Matrix** package representing an asymptotic estimate of the fourth-order central moments of the data

An object of class "manifest.data.ordinal" inherits from the "manifest.data" but has no additional slots. It is used when some of the manifest variables are ordered factors. Likewise for the "manifest.data.ranks" class which has no additional slots and is used when Spearman correlations are calculated. An object of class "manifest.data.mcd" inherits from the "manifest.data" class and has one additional slot, `CovMcd`, which is an object of `CovMcd-class` that is defined in the **rrcov** package.

## Methods

Technically, "manifest" is in the signature of all the model-fitting generic functions documented in `S4GenericsFAiR` and the same is true for `make_restrictions`. However, at the moment those methods are only defined for the "manifest.basic" class. The following methods are defined for the "manifest.basic" class, but some of them are also defined for inherited classes if their behavior needs to differ in small ways.

**cormat** signature(object = "manifest.basic")

**model.matrix** signature(object = "manifest.basic"), which also has a standardized = TRUE argument that controls whether the sample correlation matrix or sample covariance matrix is extracted

**show** signature(object = "manifest.basic")

**plot** signature(x = "manifest.basic", y = "ANY")

**screplot** signature(x = "manifest.basic")

### Author(s)

Ben Goodrich

### See Also

[make\\_manifest](#) and [S4GenericsFAiR](#)

### Examples

```
showClass("manifest")
showClass("manifest.basic")
showClass("manifest.data")
showClass("manifest.data.mcd")
man <- make_manifest(covmat = Harman74.cor)
show(man)
plot(man)
str(man)
```

---

mapping\_rule

*Default Mapping Rule*

---

### Description

It is not generally necessary to understand this help page if one merely wants to estimate a factor analysis model and this function should never be called directly. However, it is used as the default function to squash some coefficients to zero during semi-exploratory factor analysis and thus needs to be documented. Also, if one were to specify a different mapping rule function in the call to [make\\_restrictions](#), then that function would need to have some of the same arguments as this one.

### Usage

```
mapping_rule(coefs, cormat, zeros = rep(ncol(coefs), ncol(coefs)),
row_complexity = NA_integer_, quasi_Yates = FALSE, weak_Thurstone = FALSE,
Butler = FALSE, viral = FALSE, mve = FALSE, communality = FALSE)
```

**Arguments**

<code>coefs</code>	A primary pattern matrix with all cells filled in
<code>cormat</code>	A correlation matrix among primary factors with all cells filled in
<code>zeros</code>	An integer vector whose length is the same as the number of factors that indicates the requisite number of zeros in the corresponding column of the primary pattern matrix
<code>row_complexity</code>	An integer vector, either of length one or the same length as the number of rows in <code>coefs</code> , which indicates the requisite number of <i>non-zero</i> coefficients in that row of the primary pattern matrix. If a scalar, this complexity is used for all rows of the primary pattern matrix. If <code>NA_integer_</code> (the default) this row-wise mapping rule is not enforced
<code>quasi_Yates</code>	A logical indicating whether to enforce a mapping rule that is inspired by Yates (1987) to produce cohyperplanarity; see Details
<code>weak_Thurstone</code>	A logical indicating whether to enforce a mapping rule that is based on a minimal version of Thurstone's (1947) second and third rules of thumb for identifying the structure; see Details
<code>Butler</code>	A logical indicating whether to enforce a mapping a mapping rule that is inspired by Butler (1969) to produce a unit complexity basis for the common factor space; see Details
<code>viral</code>	A logical indicating whether to enforce a mapping rule that puts some outcomes (namely double the number of factors) into two hyperplanes in common factor space; see Details
<code>mve</code>	A logical that must be <code>FALSE</code> because this mapping rule is (perhaps permanently) disabled
<code>communality</code>	A logical indicating whether to enforce a mapping rule that tries to put one zero per factor within a row of the primary pattern matrix that has a relatively large communality; see Details

**Details**

The “mapping rule” is a deterministic function that takes a matrix with an insufficient number of exact zeros (often no zeros at all) and transforms it rather minimally to a matrix with the requisite number of exact zeros in the specified columns and / or rows. Hence, when the transformed matrix is returned, it should have enough zeros in the proper locations to potentially satisfy some theorem on model identification, typically Reiersøl's (1950). It is not possible to verify that all aspects of Reiersøl's (1950) theorem are satisfied, but another internal function verifies as much as it can. During the (genetic) optimization, the discrepancy function is called *after* the mapping rule is enforced and after the primary pattern matrix is checked for compliance with Reiersøl's (1950) theorem. Hence, optimization is intended to find the optimal estimates, subject to the restriction that the model is identified (with the help of the mapping rule).

By default, this function is copied into the `mapping_rule` slot in the call to `make_restrictions` and its `formals` are adjusted to correspond to the specified mapping rule. You may wish to do so manually in rare circumstances. If a different function is used to enforce a mapping rule, it should also have `coefs`, `cormat`, and `zeros` arguments as documented above.

The vignette has more details on Reiersøl's (1950) theorem and defines these mapping rules in symbols. To describe them briefly, the default "mapping rule" simply loops through the `zeros` argument and squashes the smallest `zeros[p]` cells (by magnitude) in the  $p$ th column of `coefs` to zero and returns the resulting matrix. This is the behavior depicted in the first example and can be brought about by leaving all the arguments that have default values at their defaults.

Only one mapping rule should be requested. The exception is that the default mapping rule is often called after a non-default mapping rule has been applied to make sure that that the  $p$ th column has `zeros[p]` zeros in it.

The `weak_Thurstone` mapping rule is similar but has an additional provision that no row of `coefs` may contain more than one (exact) zero. This mapping rule can be seen as minimally satisfying Thurstone's (1947) second and third rules for identification.

The `row_complexity` mapping rule is simply applied to the rows of the primary pattern matrix rather than the columns. Note that the "complexity" of an outcome is its number of *non-zero* coefficients. This mapping rule could loop over `row_complexity` (it actually utilizes `apply`) and squashes all but the largest (by magnitude) `row_complexity[j]` coefficients in the  $j$ th row of the reference structure matrix to zero and then rescales the result back to the primary pattern matrix. Then the default mapping rule is called, which may do nothing if there are now enough zeros in each column of `coefs` and the result is returned.

The `Butler` mapping rule concentrates the zeros within rows so that each factor corresponds to an outcome of complexity one. Then, the default mapping rule is called to obtain more zeros for the  $p$ th factor.

The `viral` mapping rule is less drastic but concentrates two zeros into each of  $2 * factors$  rows of `coefs` and is possibly useful if the number of factors is large. Then, the default mapping rule is called to obtain more zeros for the the  $p$ th factor.

The other mapping rules are more advanced and rely on the factor contribution matrix which is the element-by-element product of the the pattern and structure matrices. The `rowSums` of the factor contribution matrix is a vector of communalities. The `communality` mapping rule places one zero per factor in a row of `coefs` corresponding to a row of the factor contribution matrix with a high ratio of its arithmetic mean to its geometric mean. Then, the default mapping rule is called to obtain a sufficient number of zeros for the  $p$ th factor.

The `quasi_Yates` mapping rule places zeros in rows of `coefs` corresponding to rows of the factor contribution matrix with large differences between columns. The idea is to place zeros in rows where one factor is weak and another factor is strong in terms of explaining the variance in the corresponding outcome variable. This mapping rule is intended to achieve "cohyperplanarity" as discussed in Yates (1987), albeit not in reference to SEFA.

## Value

A matrix, specifically the primary pattern matrix with the requisite number of zeros in its columns and / or rows.

## Author(s)

Ben Goodrich

## References

Reiersøl, O. (1950) On the Identifiability of Parameters in Thurstone's Multiple Factor Analysis. *Psychometrika*, **15**, 121–149.

## See Also

[make\\_restrictions, parameter.coef.SEFA-class](#)

## Examples

```
## This is just a demo; you should NOT call this function directly

cormat <- diag(2)           # factor intercorrelation matrix
coefs  <- matrix(rnorm(20), nrow = 10, ncol = 2) # primary pattern matrix
zeros  <- c(2, 2)          # we require two zeros per factor

any(coefs == 0)            # FALSE

## Default mapping rule
coefs_default <- mapping_rule(coefs, cormat, zeros)
colSums(coefs_default == 0) # c(2, 2)
# Now the 2 smallest coefficients in each column are squashed to zero
print(cbind(coefs, NA, coefs_default), digits = 3)

## row_complexity mapping rule
coefs_row <- mapping_rule(coefs, cormat, zeros, row_complexity = 1)
colSums(coefs_row == 0)    # at least two per factor
# Now the smaller coefficient in each row is squashed to zero
print(cbind(coefs, NA, coefs_row), digits = 3)

## The other mapping rules are sort of useless in the two factor case
```

---

model\_comparison    *Compare Factor Analysis Models*

---

## Description

These functions produce the usual model comparison statistics for factor analysis models.

## Usage

```
model_comparison(..., correction = c("swain", "bartlett", "none"),
                 conf.level = .9, nsim = 1001)
paired_comparison(M_0, M_1)
```

## Arguments

<code>...</code>	objects of <code>FA-class</code> produced by <code>Factanal</code>
<code>correction</code>	character string indicating what correction to use
<code>conf.level</code>	confidence interval for the RMSEA statistic
<code>nsim</code>	number of simulations for the nonparametric tests, see Details
<code>M_0</code>	object of <code>FA-class</code> produced by <code>Factanal</code> that nests <code>M_1</code>
<code>M_1</code>	object of <code>FA-class</code> produced by <code>Factanal</code> that is nested within <code>M_0</code>

## Details

For exactly two nested models, `paired_comparison` performs the simple version of the test recommended in Satorra and Bentler (2000); however, it is up to the user to verify that `M_1` is nested within `M_0`.

Any number of objects of `FA-class` that are produced by `Factanal` can be passed to `model_comparison` and a wide variety of statistic tests and fit indices will be calculated. The exact behavior heavily depends on how the model was estimated and in the case of traditional maximum likelihood estimation also depends on the `correction` argument.

If `correction = "swain"` (the default), the maximum likelihood test statistic is scaled by one of the correction factors in Swain (1975) that has been recommended in Herzog, Boomsma, and Reinecke (2007) and in Herzog and Boomsma (forthcoming) and is based on <http://www.ppsw.rug.nl/~boomsma/swain.R>. Users should refer to these works for details, simulation results, and in publications making use of this Swain correction. If `correction = "bartlett"`, the correction factor recommended in Bartlett (1950), which is only strictly appropriate for exploratory factor analysis and has been implemented in `factanal` for a long time. If `correction = "none"`, then no correction factor is utilized, which is also the behavior for models that do not use the traditional maximum likelihood discrepancy function. If the ADF discrepancy function is used (or one of its special cases), the primary test statistic is that advocated in Yuan and Bentler (1998) but the test in equation 2.20b of Browne (1984) is also calculated.

The (primary) test statistic is then used in the root mean squared error of approximation (RMSEA) (see Steiger and Lind 1980) to conduct a test of “close fit”, namely that the true RMSEA is less than 0.05. Confidence intervals are also reported and depend on the value of `conf.level`. The RMSEA is in turn used to calculate Steiger’s (1989)  $\gamma$  index. In the maximum likelihood case, both of these are affected by the `correction` factor.

If the traditional maximum likelihood discrepancy function is used, then the `BIC` and `SIC` (Stochastic Information Criterion, see Preacher 2006 and Preacher, Cai, and MacCallum 2007) are calculated. These information criteria can be used to compare nonnested models and in both cases, smaller is better.

Finally, several model comparison statistics are calculated, largely based on the `summary.sem` function in the `sem` package. Most of these statistics are discussed in Bollen (1989). These are

List element	Reference
McDonald	McDonald’s (1989) Centrality Index
GFI	Jöreskog’s and Sorböm’s (1981) Goodness of Fit Index
AGFI	Jöreskog’s and Sorböm’s (1981) Adjusted Goodness of Fit Index
SRMR	Bentler’s (1995) Standardized Root Mean-squared Residual

TLI	Tucker and Lewis (1973) Index
CFI	Bentler's (1995) Comparative Fit Index
NFI	Bentler and Bonett's (1980) Normalized Fit Index
NNFI	Bentler and Bonett's (1980) Nonnormalized Fit Index

### Value

paired\_comparison produces an object of S3 class "htest"; model\_comparison produces a list with the following elements:

restrictions	the restrictions object for the model
exact_fit	a list of one or more objects, usually of S3 class "htest", indicating the results of the associated test(s) of exact fit
close_fit	a list of one or more objects, usually of S3 class "htest", indicating the results of the associated test or measure of "close" fit
fit_indices	a list of numeric fit indices
infocriteria	in the case of maximum likelihood estimation, a list of the information criteria that were calculated

### Warning

If Yates' weighted least squares discrepancy function is used, the test statistic is not strictly valid.

### Author(s)

Ben Goodrich with major contributions by Anne Boomsma, John Fox, and Walter Herzog

### References

- Bentler, P.M. (1995), *EQS structural equations program manual*. Encino, CA: Multivariate Software.
- Bentler, P.M., & Bonett, D.G. (1980), "Significance tests and goodness of fit in the analysis of covariance structures". *Psychological Bulletin*, **88**, 588–606.
- Browne, M.W. (1984), "Asymptotically distribution-free methods for the analysis of covariance structures", *British Journal of Mathematical and Statistical Psychology*, **37**, 62–83.
- Bollen, K. A. (1989) *Structural Equations With Latent Variables*. Wiley.
- Herzog, W., and Boomsma, A. (forthcoming), "Small-Sample Robust Estimators of Noncentrality-Based and Incremental Model Fit" *Structural Equation Modeling*.
- Herzog, W., Boomsma, A., and Reinecke, S. (2007), "The model-size effect on traditional and modified tests of covariance structures". *Structural Equation Modeling*, **14**, 361–390.
- Hotelling, H. (1931), "The generalization of Student's ratio", *Annals of Mathematical Statistics*, **2**, 360–378.
- Jöreskog, K. G., and Sorböm, D. (1981). *LISREL V: Analysis of linear structural relations by the method of maximum likelihood*. Chicago: International Educational Services.
- McDonald, R.P. (1989), "An index of goodness-of-fit based on noncentrality", *Journal of Classification*, **6**, 97–103.

Preacher, K.J. (2006), “Quantifying Parsimony in Structural Equation Modeling”, *Multivariate Behavioral Research* **41**, 227–259.

Preacher, K.J., Cai, L., and MacCallum, R.C. (2007), “Alternatives to traditional model comparison strategies for covariance structure models.” in *Modeling Contextual Effects in Longitudinal Studies*, eds. Little, T.D., Bovaird, J.A., and Card, N.A. Psychology Press.

Satorra, A and Bentler, P.M. (2001), “A scaled difference chi-square test statistic for moment structure analysis,” *Psychometrika*, **66**, 507–514.

Steiger, J.H. and Lind, J.C. (1980), “Statistically based tests for the number of common factors” Paper presented at the annual meeting of the Psychometric Society, Iowa City, IA.

Steiger, J.H. (1989), EzPATH: A supplementary module for SYSTAT and SYGRAPH. Evanston, IL: SYSTAT.

Swain, A.J. (1975). *Analysis of parametric structures for variance matrices*. Unpublished doctoral dissertation, Department of Statistics, University of Adelaide, Australia.

Tucker, L. R, and Lewis, C. (1973), “A reliability coefficient for maximum likelihood factor analysis”. *Psychometrika*, **38**, 1–10.

## See Also

[Factanal](#)

## Examples

```
## See example in Factanal()
```

---

parameter-class      *Class "parameter"*

---

## Description

It is not necessary to understand this help page if one merely wants to estimate a factor analysis model. This help page is intended for those who want to modify or extend FAiR or otherwise want some idea of how **FAiR** works “behind the scenes”.

This class is used internally to hold information about the parameters in a factor analysis model.

## Objects from the Class

Objects can be created by calls of the form `new("parameter", ...)`. However, rarely, if ever, would one want to do so directly because `make_restrictions` instantiates them and puts them into the slots of an object of `restrictions-class`.

## Slots

The "parameter" class is virtual and has the following slots:

- x:** Object of class "ANY" but is a numeric vector or numeric matrix in all known inherited classes. Its elements can be free parameters or parameters that are fixed and not estimated. In the case of the former, the corresponding element of  $x$  should be NA. In the case of the latter, the corresponding element of  $x$  should be the number that the parameter is fixed to or Inf if the parameter is "fixed" to be a function of other parameters.
- free:** Object of class "ANY" but for all known inherited class is a logical vector or logical matrix whose length is the same as the length of  $x$  that indicates which elements of  $x$  are considered "free" parameters.
- num\_free:** Integer indicating the number of free parameters in  $x$ .
- select:** A logical vector indicating which elements of the grand parameter vector correspond to the free elements of  $x$ .
- invalid:** A numeric scalar that is 0.0 if the parameters are *not* invalid and some number other than 0.0 if the parameters are invalid.
- Domains:** an array that is either empty or stacks exactly two matrices that are the same dimension as  $x$ . The lower matrix should indicate the lower bound for the parameter and the upper matrix should indicate the upper bound. It is often unnecessary to specify Domains because they are completed by the `make_restrictions-methods`.

The "parameter.cormat" class extends "parameter" and is used for correlation matrices among primary factors. Hence,  $x$  is of class "matrix" and the only free parameters are in the lower triangle; however, not all elements of the lower triangle need to be free parameters. The "parameter.scale" class extends "parameter" and is used for (diagonal) matrices of standard deviations of the outcome variables that pre- and post-multiply the correlation matrix in the "embedded correlation" parameterized currently used for all models in FAiR. Hence,  $x$  is of class "numeric" and is the diagonal of such a scale matrix. Typically, all diagonal elements are free but not always, as in the case where some observed variable is designated as a factor, in which case its standard deviation is estimated from the data rather than by the model.

The "parameter.coef" class extends "parameter" and is used for primary pattern matrices. Hence, " $x$ " is of class (numeric) "matrix" and its cells may be free (designated by NA) or fixed. It has one additional slot,

- equalities:** a list (possibly with zero length) of objects of `equality_restriction-class` to indicate equality restrictions among some cells of  $x$ .

The "parameter.coef.nl" class extends "parameter.coef" and has one additional slot,

- nonlinearities:** a function with an argument called "pattern". After  $x$  is filled with free parameters and any equality restrictions are resolved,  $x$  is passed to this function whose body can enforce other exact restrictions on the cells. For example, one "'fixed'" cell could be the product of other cells. Then, this function must return this primary pattern matrix whose cells are all finite numbers, which will be reinserted into the  $x$  slot for later use.

The "parameter.coef.SEFA" class extends "parameter.coef" and is used in semi-exploratory factor analysis. It has two additional slots,

**rankcheck:** either "reiersol" or "howe" to indicate which theorem is to be used in checking the rank of submatrices of the primary pattern matrix with exact zeros in a column. See the Details section.

**mapping\_rule:** a function that defaults to the one documented in [mapping\\_rule](#) and is used to make some cells of `x` exactly zero. It is possible to define a different function for this slot but see [mapping\\_rule](#) because the underlying code is hard-coded to assume some of the same arguments in some places. This function is called after the cells of `x` have been filled and any equality restrictions have been resolved and should output a matrix that will be reinserted into the `x` slot.

**squashed:** a logical matrix indicating which cells of the coefficient matrix were squashed to zero by the mapping rule. This slot is typically empty until the model has been estimated, at which point the [create\\_FAobject-methods](#) must fill it.

The `"parameter.coef.SEFA.nl"` class extends `"parameter.coef.SEFA"` and has one additional slot,

**nonlinearities:** A function, the same as above for the `"parameter.coef.nl"` class. The function enforcing nonlinear restrictions is called before the mapping rule function.

## Methods

A [cormat](#) method extracts the `x` slot of an object of `"parameter.cormat"`. A [coef](#) method extracts the `x` slot of an object that inherits from `"parameter.coef"`. There is no special extractor function for objects of `"parameter.scale"`. There are also [show](#) methods.

The most important methods are those defined for the `make_parameter` S4 generic. This function takes two arguments, `"par"`, which is a numeric vector but not part of the signature, and `"object"`, which is an object that inherits from the `"parameter"` class. The `make_parameter` methods are defined for each inherited class and basically do two things. First, they do something like this

```
object@x[object@free] <- par[object@select]
```

to fill the free elements of `x` with corresponding values from `"par"`. Then, these methods often check whether the parameters are collectively admissible under the assumptions of the factor analysis model. If not, the `make_parameter` method must set the `invalid` slot to some number that is greater than  $-1.0$  and not  $0.0$ . It is preferable if larger values somehow indicate more flagrant inadmissability, since this number is used as a return value in the lexical optimization process (see [genoud](#)). Finally, the `make_parameter` should return `object`.

## Author(s)

Ben Goodrich

## References

Howe, W.G. (1955) *Some Contributions to Factor Analysis*. Dissertation published as ORNL-1919 by Oak Ridge National Laboratory in Tennessee.

Reiersøl, O. (1950) On the Identifiability of Parameters in Thurstone's Multiple Factor Analysis. *Psychometrika*, **15**, 121–149.

**Examples**

```

showClass("parameter")
showClass("parameter.cormat")
showClass("parameter.scale")
showClass("parameter.coef")
showClass("parameter.coef.nl")
showClass("parameter.coef.SEFA")
showClass("parameter.coef.SEFA.nl")
showMethods("make_parameter")

```

---

read.cefa

*Read and Write Files Produced by or for CEFA*


---

**Description**

Compatibility functions between R and Comprehensive Exploratory Factor Analysis (CEFA) Version 2.0 by Michael W. Browne, Robert Cudeck, Krishna Tateneni, and Gerald Mels.

**Usage**

```

read.cefa(file)
read.CEFA(file)
write.cefa(file, FAobject)
write.CEFA(file, FAobject)

```

**Arguments**

file	Character string giving the path to the file to be imported, possibly "" to print the file to the screen.
FAobject	object of <i>FA-class</i> produced by <i>Factanal</i> under EFA

**Details**

read.cefa does not support importing a matrix of factor loadings, (Datatype 3 in CEFA) because the file does not include enough information to be useful to **FAiR**. Instead, import the covariance matrix (Datatype 1 in CEFA) or better the raw dataset (Datatype 2 or 4 in CEFA) and use *Factanal* to reestimate the model.

In contrast, the only export method that is currently supported is that for preliminary factor loadings in a EFA. read.CEFA is just an alias for read.cefa and similarly for write.CEFA.

These functions have not been tested very much.

**Value**

If a raw dataset is imported, `read.cefa` returns a dataframe with columns as variables. Such a dataframe can be passed to the `data` or `x` argument of `make_manifest`. If a covariance matrix is imported, `read.cefa` returns a two-element list with the following items:

<code>cov</code>	The covariance matrix
<code>n.obs</code>	Number of observations

This list can be passed to the `covmat` argument of `make_manifest`. `write.cefa` does not produce anything but writes a file to the specified location.

**Author(s)**

Ben Goodrich

**References**

CEFA is available for Windows (but also runs under Wine in Linux) from <http://faculty.psy.ohio-state.edu/browne/software.php>

**See Also**

`read.triangular` imports a triangular covariance matrix, the **foreign** library has functions to import files created by SPSS, SAS, etc., `read.table` imports delimited text files and `read.fwf` imports fixed-width text files.

**Examples**

```
## Not run:
  OrgComm <- read.cefa(file = file.path("Program Files", "CEFAtool"
                                       "OrgComm.inp"))

## End(Not run)
```

---

`read.triangular`      *Input a Covariance or Correlation Matrix*

---

**Description**

This help file and function are essentially copied from `read.moments`, which is written by John Fox. This function makes it simpler to input covariance (preferable) or correlation matrix to be analyzed. The matrix is input in **lower-triangular** form on as many lines as is convenient, omitting the above-diagonal elements. The elements on the diagonal may also optionally be omitted, in which case they are taken to be 1. In terms of features that **FAiR** supports, it is preferable to input the raw data if available, followed by inputting a covariance matrix, and inputting a correlation matrix is a last resort.

**Usage**

```
read.triangular(file = "", diag = TRUE,
               names = paste("X", 1:n, sep = ""), ...)
```

**Arguments**

file	The (quoted) file from which to read the model specification, including the path to the file if it is not in the current directory. If "" (the default), then the specification is read from the standard input stream, and is terminated by a blank line.
diag	If TRUE (the default), then the input matrix includes diagonal elements, which is preferable in <b>FAiR</b> .
names	a character vector containing the names of the variables, to label the rows and columns of the moment matrix.
...	further arguments passed to <code>scan</code>

**Value**

Returns a square, symmetric covariance or correlation matrix, suitable for input to the `covmat` argument of `make_manifest`.

**Author(s)**

John Fox, with some small changes by Ben Goodrich

**See Also**

`make_manifest`

**Examples**

```
R.DHP <- read.triangular(diag=FALSE, names=c("ROccAsp", "REdAsp", "FOccAsp",
      "FEdAsp", "RParAsp", "RIQ", "RSES", "FSES", "FIQ", "FParAsp"),
      nlines = 9)

      .6247
      .3269 .3669
      .4216 .3275 .6404
      .2137 .2742 .1124 .0839
      .4105 .4043 .2903 .2598 .1839
      .3240 .4047 .3054 .2786 .0489 .2220
      .2930 .2407 .4105 .3607 .0186 .1861 .2707
      .2995 .2863 .5191 .5007 .0782 .3355 .2302 .2950
      .0760 .0702 .2784 .1988 .1147 .1021 .0931 -.0438 .2087

R.DHP
```

---

restrictions-class *Class "restrictions"*

---

## Description

It is not necessary to understand this help page if one merely wants to estimate a factor analysis model. This help page is intended for those who want to modify or extend FAiR or otherwise want some idea of how **FAiR** works “behind the scenes”.

This class contains information in its slots about what restrictions are placed on a factor analysis model, which defines the model to be estimated.

## Details

These classes are fundamental in **FAiR** and fulfill two important roles. First, they serve as a vessel that contains the parameter matrices and other necessary information to estimate different factor analysis models via `Factanal`. During the genetic optimization process, the `restrictions2model` method repeatedly calls `make_parameter-methods` to “fill” the “empty”  $\times$  slots of the matrices within the "restrictions" object that inherit from `parameter-class`. Eventually, the optimal parameters satisfying the specified constraints are found by the genetic algorithm and the `create_FAobject` creates an object that inherits from class `FA-class` on the basis of this “filled” "restrictions" object.

The second purpose of a "restrictions" object is to remind the user of exactly what restrictions were imposed during the estimation process, which is why a tailored `show` method is essential.

Here is a table of the class hierarchy:

Class	Parent Class	Corresponding Model
"restrictions"	none	none (virtual class)
"restrictions.independent"	"restrictions"	zero factor null model
"restrictions.factanal"	"restrictions"	EFA via factanal algorithm
"restrictions.orthonormal"	"restrictions"	EFA via a different algorithm
"restrictions.1storder"	"restrictions"	Models with correlated factors
"restrictions.1storder.EFA"	"restrictions.1storder"	EFA after factor transformation
"restrictions.general"	"restrictions.1storder"	SEFA and CFA
"restrictions.2ndorder"	"restrictions.general"	SEFA and CFA

## Objects from the Class

Objects can be created by calls of the form `new("restrictions", ...)`. However, it is not recommended to do so in normal usage because `make_restrictions` provides the constructor for users.

## Slots

The "restrictions" class is virtual has the following slots:

**factors:** An integer vector of length two indicating the number of factors at level one and level two of the model, must be nonnegative

**nvars:** An integer indicating the number of free parameters in the model, which corresponds to the `nvars` argument to `genoud`.

**dof:** An integer indicating the number of degrees of freedom for the factor analysis model

**Domains:** A numeric matrix with `nvars` rows and exactly two columns indicating the lower and upper bounds for each free parameter, which corresponds to the `Domains` argument to `genoud`.

**model:** A character string indicating whether a "SEFA", "EFA", or "CFA" model is to be estimated.

**discrepancy:** A character string indicating which discrepancy function is used to estimate the model, see the options for the same argument to `make_manifest`

**free:** A logical vector that indicates which of the parameters in the various matrices are free and which are considered fixed, excluding parameters that are "obviously" fixed.

An object of class `"restrictions.independent"` inherits from `"restrictions"` and is used to define a "null" model where there are no factors (hence `factors = c(0L, 0L)`) and the only parameters to estimate are the manifest standard deviations. It has two additional slots:

**scale:** an object of `parameter.scale-class`

**criteria:** A list with one function, the discrepancy function that will be minimized to estimate the model and is that named by the `discrepancy` slot

An object of class `"restrictions.factanal"` inherits from `"restrictions"` but has the same slots. This class is used for exploratory factor analysis via the same restrictions that are used in `factanal`. An object of class `"restrictions.orthonormal"` inherits from `"restrictions"` and also implies exploratory factor analysis but instead imposes the restrictions that the upper triangle of the coefficient matrix has all zeros. It has the following additional slots:

**beta:** an object of `parameter.coef-class` where the `x` slot is a primary pattern matrix with 0.0 in its upper triangle and NA elsewhere

**scale:** an object of `parameter.scale-class`

**criteria:** A list with one function, the discrepancy function that will be minimized to estimate the model and is that named by the `discrepancy` slot

An object of class `"restrictions.lstorder"` inherits from `"restrictions"` and is used for factor analysis models with multiple correlated factors. It has the following additional slots:

**Phi:** an object of `parameter.cormat-class`

**beta:** an object that inherits from `parameter.coef-class`. If `model = "CFA"` it should be of `parameter.coef-class` or `parameter.coef.nl-class`, depending on whether there are any nonlinear exact restrictions to impose. If `model = "SEFA"`, it should be of `parameter.coef.SEFA-class` or `parameter.coef.SEFA.nl-class`, again depending on whether there are any nonlinear exact restrictions to impose. Unlike the case for `"restrictions.orthonormal"`, its `x` slot can have any combination of free and fixed cells.

**coef:** an object of `parameter.scale-class`

**criteria** A list of functions to be evaluated as lexical criteria during the lexical optimization process. Hence, this list may contain more than one function but the function named in the `discrepancy` slot must be the last element. Preceding list elements should be functions that return  $-1.0$  if the constraint is satisfied and some number greater than  $-1.0$  if the constraint is not satisfied

An object of class `"restrictions.1storder.EFA"` inherits from `"restrictions.1storder"` and is used for exploratory factor analysis after the preliminary factors have been transformed. It has the following additional slots:

**Lambda:** A numeric matrix with zeros in its upper triangle containing the preliminary factor loadings, in other words the `x` slot of the `beta` slot of an object of class `"restrictions.orthonormal"`

**orthogonal** A logical indicating whether the transformation was orthogonal or oblique. `Rotate` only supports oblique transformations, but it is possible to “import” an orthogonal transformation from the **GPARotation** package using the `GPA2FA` function

**Tmat** A square numeric matrix whose order is equal to the number of factors that was found during the factor transformation process. Note that this matrix should have unit-length *columns*, whereas many textbooks instead specify unit-length *rows*

**Tcriteria** A list, usually containing functions that were used as lexical criteria during the lexical optimization process (see `Rotate`) If the **GPARotation** package was used to find the transformation, then this list must be a single character string naming the analytic criterion used

An object of class `"restrictions.general"` inherits from `"restrictions.1storder"` and is used for SEFA and CFA with a single second-order factor that explains the systematic correlation among the first-order factors. It has one additional slot,

**Delta:** an object of `parameter.coef-class` or `parameter.coef.nl-class`, depending on whether there are any nonlinear exact restrictions to impose. Its `x` slot has only one column but it can have any combination of free and fixed cells

An object of class `"restrictions.2ndorder"` inherits from `"restrictions.general"` and is used for SEFA and CFA models with multiple second-order factors to explain the systematic correlation among the first-order factors. Its `Delta` slot now must only inherit from `parameter.coef-class` and hence can also be `parameter.coef.nl-class` if `model = "CFA"` or, if `model = "SEFA"` can be of `parameter.coef.SEFA-class` or `parameter.coef.SEFA.nl-class`, again depending on whether there are any nonlinear exact restrictions to impose at level two. The `"restrictions.2ndorder"` class has one additional slot,

**Xi:** an object of `parameter.cormat-class`

## Methods

**show** `signature(object = "restrictions")`, prints the object in a relatively easy-to-digest format and there are tailored methods for almost all of the classes that inherit from the `"restrictions"` class

The following methods are listed here but documented elsewhere

**restrictions2model** the `restrictions2model` methods take a grand parameter vector and “fill” in the free cells of the appropriate matrices with a `"restrictions"` object

- fitS4** the `fitS4` methods evaluate all the lexical criteria given a “filled” object that inherits from class "restrictions"
- bfgs\_fitS4** the `bfgs_fitS4` methods return the discrepancy function given a “filled” object that inherits from class "restrictions"
- gr\_fitS4** the `gr_fitS4` methods return the gradient of the discrepancy function given a “filled” object that inherits from class "restrictions"
- bfgs\_fitS4** the `bfgs_fitS4` methods return a list that is then passed to the `bfgs_fitS4` and `gr_fitS4` methods; see the `BFGShelp` argument to `genoud`
- create\_start** the `create_start` methods return a matrix of starting values for the genetic algorithm; see also `genoud`
- create\_FAobject** the `create_FAobject` methods return an object that inherits from `FA-class` once the optimal estimates have been found
- restrictions2draws** the `restrictions2draws` methods return a list of arrays of simulated draws from the sampling distribution of the estimated free parameters
- restrictions2RAM** the `restrictions2RAM` methods convert an object that inherits from class "restrictions" from that parameterization used in the **FAiR** package to the reticular action model (RAM) parameterization used in the **sem** package
- loadings** the `loadings` methods extract the factor loadings
- coef** the `coef, restrictions-methods` also extract the factor loadings
- cormat** the `cormat` methods extract the correlation matrices among factors
- uniquenesses** the `uniquenesses` methods extract the uniquenesses
- fitted** the `fitted, restrictions-methods` produce a correlation or covariance matrix as a function of the model parameters
- df.residual** the `df.residual, restrictions-methods` extract the degrees of freedom for the factor analysis model

**Author(s)**

Ben Goodrich

**See Also**

`make_restrictions`

**Examples**

```
showClass("restrictions")
showClass("restrictions.independent")
showClass("restrictions.orthonormal")
showClass("restrictions.1storder")
showClass("restrictions.1storder.EFA")
showClass("restrictions.general")
showClass("restrictions.2ndorder")
```

---

restrictions2draws *Draw from sampling distribution of estimates*

---

## Description

It is not necessary to understand this help page if one merely wants to estimate a factor analysis model. This help page is intended for those who want to modify or extend FAiR or otherwise want some idea of how **FAiR** works “behind the scenes”.

These functions facilitate drawing from a multivariate normal distribution defined by the estimated free parameters and their variance-covariance matrix. The draw is accepted if it satisfies all constraints and rejected otherwise. Appropriate transformations are then applied to acceptable draws. This mechanism makes it possible to calculate the uncertainty of estimates in a general fashion. It is unlikely that a user would ever need to call `restrictions2draws` directly; it is more common to use `FA2draws`, which calls `restrictions2draws` internally.

## Usage

```
FA2draws(object, nsim = 1001, seed = NULL,
          covariances = FALSE, standardized = TRUE, ...)
```

## Arguments

<code>object</code>	object of <code>FA-class</code>
<code>nsim</code>	number of non-rejected simulations
<code>seed</code>	either <code>NULL</code> or an integer that will be used in a call to <code>set.seed</code> before simulating the free parameters. The default, <code>NULL</code> will not change the random generator state
<code>covariances</code>	logical indicating whether return simulations of the reproduced covariances, which are a function of the free parameters, rather than simulations of the free parameters themselves. The default, <code>TRUE</code> , returns the simulations of the free parameters, which can then be manipulated into simulations of the reproduced covariances
<code>standardized</code>	logical indicating whether the simulations should be rescaled so that they are calibrated to a correlation matrix
<code>...</code>	other arguments passed to downstream functions; not currently used

## Details

Traditional methods of estimating the uncertainty of parameter estimates are not necessarily appropriate in **FAiR** because of the extensive possibilities for imposing inequality restrictions on the parameters during the factor extraction stage and / or the factor transformation stage (in the case of EFA only). Thus, traditional asymptotic approximations to the sampling distribution of the parameters could place some mass on a region of the parameter space that is inconsistent with the inequality restrictions. To work around this problem, these functions draw simulations of the free

parameters from an unconstrained multivariate normal distribution, following King (1998). If the draw is consistent with the inequality restrictions, it is accepted and transformed as necessary; otherwise it is rejected. Once `nsim` acceptable simulations have been obtained, they are returned in a list of arrays for further analysis.

### Value

A named list of arrays where in each case the third dimension has extent `nsim`.

### Methods

Note `FA2draws` is not a S4 generic function, but it primarily exists to call the S4 generic function, `restrictions2draws`. Methods are currently only defined for objects of class `"manifest.basic"`, which are inherited by objects of class `"manifest.data"` and `"manifest.data.mcd"`. There are methods for each flavor of `restrictions-class`, except for `"restrictions.factanal"`.

There are also at least four arguments that are **not** part of the signature of `restrictions2draws`. The first is `vcov`, which is the variance-covariance matrix for the free parameters and is simply extracted from the slot of the object of `FA-class`. The second through fourth arguments are `nsim`, `covariances`, and `standardized` as documented in the arguments section.

For the `restrictions2draws` method defined for class `"restrictions.orthonormal"`, there are two additional arguments. The first is `Tmat`, which can be `NULL` but can also be an optimal transformation matrix (found by `Rotate`). The second is `criteria`, which is a list of criteria utilized when finding the optimal transformation matrix. If both of these arguments are specified, the method will produce draws of the transformed parameters, after generating them from a multivariate normal distribution. They are extracted from the object of `FA-class` as necessary.

### Author(s)

Ben Goodrich

### References

King, G. (1998) *Unifying Political Methodology: The Likelihood Theory of Statistical Inference*. University of Michigan Press.

### See Also

`confint`, `FA-method`, `simulate`, and `summary.FA-class`

### Examples

```
## See example for Factanal()
```

---

```
restrictions2Mathomatic
    Format Equations for Mathomatic
```

---

### Description

This function exports a system of equations that can be read by the program Mathomatic by George J. Gesslein II.

### Usage

```
restrictions2Mathomatic(object, file = "")
restrictions2mathomatic(object, file = "")
```

### Arguments

<code>object</code>	An object of <code>restrictions-class</code> or <code>FA-class</code> .
<code>file</code>	Character string giving the path to the export file. This argument is passed to <code>cat</code> , so by default the file is printed to the screen (and may be wrapped incorrectly).

### Details

`restrictions2mathomatic` is just an alias for `restrictions2Mathomatic`, which “symbolically” multiplies the correlation matrix among primary factors from the left and right by the primary pattern matrix and equates the lower triangle (exclusive of the diagonal) of this reduced covariance matrix to the lower triangle of the population covariance matrix. The resulting system of equations is exported in a format suitable for reading into Mathomatic, which is a lightweight program that can solve systems of nonlinear equations symbolically. Thus, one can use Mathomatic to verify that a factor analysis model is identified in the sense that (if the model holds exactly) the free parameters can be expressed solely in terms of the population covariances. See, for example, Steiger (2002) illustrations, although `Factanal` should automatically avoid the specific problems that Steiger (2002) is most concerned with.

You have to manually add equations to the system that reflect the constraints imposed on the model; doing so is trivial at the Mathomatic prompt. However, for a variety of reasons, actually *solving* the system of equations in Mathomatic may involve more work and frustration than one might expect. For example, if you have more than  $n = 14$  manifest variables, then you have to recompile Mathomatic changing `#define N_EQUATIONS` in `Mathomatic/am.h` to some number bigger than 100 to hold all  $0.5 * n * (n - 1)$  equations implied by the factor analysis model, plus equations for the constraints.

### Value

Nothing is returned. A file is written to `file`.

**Author(s)**

Ben Goodrich

**References**

Mathomatic is licensed under LGPL 2.1 and is available in source and binary forms from <http://mathomatic.orgserve.de/math/>.

Steiger, J. H. (2002) When constraints interact: A caution about reference variables identification constraints, and scale dependencies in structural equation modeling. *Psychological Methods*, 7, 210–227.

**Examples**

```
man <- make_manifest(covmat = ability.cov)
beta <- matrix(NA_real_, nrow = nrow(covmat(man)), ncol = 2)
rownames(beta) <- rownames(covmat(man))
free <- is.na(beta)
beta <- new("parameter.coef.SEFA", x = beta, free = free, num_free = sum(free))

Phi <- diag(2)
free <- lower.tri(Phi)
Phi <- new("parameter.covmat", x = Phi, free = free, num_free = sum(free))
res <- make_restrictions(manifest = man, beta = beta, Phi = Phi,
                        discrepancy = "MLE")

restrictions2Mathomatic(res, file = "") # file printed to screen
```

---

restrictions2model *Map free parameters to matrices*

---

**Description**

It is not necessary to understand this help page if one merely wants to estimate a factor analysis model. This help page is intended for those who want to modify or extend FAiR or otherwise want some idea of how FAiR works “behind the scenes”.

This S4 generic function takes a numeric vector of free parameters and manipulates it into the matrices that are typical when estimating a factor analysis model. There is little need to call it directly. Various methods are defined for the `restrictions` object, corresponding to different kinds of factor analysis models; see [FAiR-package](#).

**Usage**

```
## S4 method for signature 'restrictions, manifest':
restrictions2model(par, restrictions, manifest, lower, mapping_rule)
```

## Arguments

<code>par</code>	A numeric vector containing values for all the free parameters to be estimated, which corresponds to the <code>par</code> argument for <code>genoud</code> and for <code>optim</code> .
<code>restrictions</code>	An object of <code>restrictions-class</code> .
<code>manifest</code>	An object of <code>manifest-class</code> .
<code>lower</code>	A small numeric scalar indicating the lower bound for positive definiteness or minimum uniqueness; see the corresponding argument to <code>Factanal</code> .
<code>mapping_rule</code>	a logical that controls whether the mapping rule is invoked in semi-exploratory factor analysis. This argument is used to circumvent the mapping rule in some circumstances, like when the function is called indirectly by <code>optim</code>

## Details

This function is called internally by `Factanal` thousands of times during the course of the optimization. Let the factor analysis model in the population be

$$\Sigma = \Omega(\beta\Phi\beta' + \Theta)\Omega$$

and let  $\theta$  be a vector of all the free parameters in the factor analysis model. The `restrictions2model` methods are essentially a mapping from  $\theta$  to the free elements of  $\beta$ ,  $\Phi$ , and  $\Omega$ . The methods are currently defined on the `restrictions` argument, which is fairly skeletal at the outset, is filled in by the body of the `restrictions2model` method using a bunch of calls to the `make_parameter-methods`.

The `restrictions2model` method can (and does) also do some checking of whether the parameters in `par` are admissible in the context of a factor analysis model. For each admissability check, if `par` is admissible, it should receive the value of  $-1.0$  in a numeric vector whose length is equal to the number of admissability checks. If `par` is inadmissible with respect to some check, it should receive a value greater than  $-1.0$ . This numeric vector should also be returned as an element of a list called `"criteria"`.

## Value

A list with exactly two elements named

<code>criteria</code>	a numeric vector
<code>restrictions</code>	a filled-in version of the <code>restrictions</code> object

## Methods

The arguments that are part of the signature are `restrictions` and `manifest`. Methods are currently only defined for objects of class `"manifest.basic"`, which are inherited by objects of class `"manifest.data"` and `"manifest.data.mcd"`. There are methods for each of the classes that inherit from `restrictions-class`, except for `"restrictions.factanal"`, which does not utilize the `restrictions2model` mechanism.

There are also two arguments that are **not** part of the signature. The first is `par`, which is a numeric vector of free parameters and is conceptually similar to the `par` argument to `optim`. The second is `lower`, which is a small positive number that is used as a threshold for positive-definiteness of various matrices.

**Author(s)**

Ben Goodrich

**See Also**

[make\\_parameter](#), [restrictions-class](#)

**Examples**

```
showMethods("restrictions2model")
```

---

`restrictions2RAM`    *Convert to reticular action model*

---

**Description**

These functions convert to the reticular action model (RAM) format utilized by the **sem** package.

**Usage**

```
FA2RAM(FAobject)
```

**Arguments**

FAobject      object of [FA-class](#)

**Value**

A character matrix with three columns of S3 class "mod", which is suitable for passing to [sem](#).

**Methods**

Note `FA2RAM` is not a S4 generic function, but it primarily exists to call the S4 generic function, `restrictions2RAM`. There are methods for each flavor of [restrictions-class](#), except for "restrictions.factanal".

The `restrictions2RAM` S4 generic function takes one argument, `restrictions`, which is an object that inherits from [restrictions-class](#).

**Warning**

It is not possible to map exactly to the RAM specification used in the **sem** package for models with two levels. `FA2RAM` will produce a warning in such cases indicating that the normalizations of the latent variables differ from the normalizations used in **FAiR**. The reproduced covariance matrix should be the same (up to some numerical error) but the parameter values at the optimum will differ.

**Author(s)**

Ben Goodrich

**See Also**

See [specify.model](#) and [sem](#) in the **sem** package for details on the RAM specification.

**Examples**

```
man <- make_manifest(covmat = ability.cov)

## Here is also an example of how to set up a CFA model the hard way
beta <- matrix(NA_real_, nrow = nrow(cormat(man)), ncol = 2)
rownames(beta) <- rownames(cormat(man))
beta[2:3,1] <- beta[5:6,2] <- 0
free <- is.na(beta)
beta <- new("parameter.coef", x = beta, free = free, num_free = sum(free))

Phi <- diag(2)
free <- lower.tri(Phi)
Phi <- new("parameter.cormat", x = Phi, free = free, num_free = sum(free))

res <- make_restrictions(manifest = man, beta = beta, Phi = Phi,
                        discrepancy = "MLE")

show(res)
RAM <- restrictions2RAM(res)
if(require(sem)) {
  RAM[,3] <- c(0.5, 0, 0, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0, 0,
             1, 0, 1, 0.5, 0.5, 0.5, 0.5, 0.5)
  cfa <- sem(RAM, model.matrix(man, standardize = FALSE), man@n.obs)
  summary(cfa)
}
```

---

 Rotate

---

*Choose a Transformation in Exploratory Factor Analysis*


---

**Description**

This function is intended for users and finds an optimal rotation of preliminary factor loadings extracted via exploratory factor analysis. Unlike other software, `Rotate` finds a rotation that is optimal with respect to an **intersection** of criteria, one of which is a “analytic” criterion and the rest are “constraints”. Similar to [make\\_restrictions](#), much of the functionality of `Rotate` is implemented via pop-up menus, which is the **strongly** recommended way to proceed. Also, the vignette details what all these options *mean* in substantive terms; execute `vignette("FAiR")` to read it.

**Usage**

```
Rotate(FAobject, criteria = list(), methodArgs = list(),
       normalize = rep(1, nrow(loadings(FAobject))), seeds = 12345,
       NelderMead = TRUE, ...)
```

**Arguments**

FAobject	An object of <code>FA.EFA-class</code> produced by <code>Factanal</code> .
criteria	An <b>optional</b> list whose elements are functions or character strings naming functions to be used in the lexical optimization process. If unspecified, <code>Rotate</code> will prompt you with pop-up menus, which is the recommended way to proceed. See the Details section otherwise.
methodArgs	an option list with named elements for additional arguments needed by some criterion functions; see the Details section
normalize	a vector with the same length as the number of manifest variables, or a function that takes the preliminary primary pattern matrix as its first argument and produces such a vector, or a character string that is either "kaiser" or "cureton-mulaik", which are discussed more in the Details section. The rows of the preliminary primary pattern matrix are <b>divided</b> by this vector before the optimal transformation is found. Note that row-normalization may not be sensible, especially if the objective function optimizes the reference structure matrix or the factor contribution matrix.
seeds	A vector of length one or two to be used as the random number generator seeds corresponding to the <code>unif.seed</code> and <code>int.seed</code> arguments to <code>genoud</code> respectively. If <code>seeds</code> is a single number, this seed is used for both <code>unif.seed</code> and <code>int.seed</code> . These seeds override the defaults for <code>genoud</code> and make it easier to replicate an analysis exactly. However, if <code>seeds = NULL</code> , then the default seeds are used, which is absolutely necessary during simulations.
NelderMead	Logical indicating whether to call <code>optim</code> with <code>method = "Nelder-Mead"</code> when the genetic algorithm has finished to further polish the solution.
...	Further arguments that are passed to <code>genoud</code> . Note that several of the default arguments to <code>genoud</code> are silently overridden by <code>Factanal</code> out of logical necessity:

argument	value	why?
<code>nvars</code>	<code>FAobject@restrictions@factors[1]^2</code>	
<code>max</code>	<code>FALSE</code>	minimizing the objective not meaningful
<code>hessian</code>	<code>FALSE</code>	restricted optimization
<code>lexical</code>	<code>TRUE</code>	
<code>Domains</code>	<code>NULL</code>	
<code>default.domains</code>	<code>1</code>	parameters are cosines
<code>data.type.int</code>	<code>FALSE</code>	parameters are doubles
<code>fn</code>	wrapper around an internal function	
<code>BFGSfn</code>	wrapper around an internal function	
<code>gn</code>	<code>NULL</code>	analytic gradients are unknown
<code>BFGShelp</code>	wrapper around an internal function	
<code>unif.seed</code>	taken from <code>seeds</code>	replicability
<code>int.seed</code>	taken from <code>seeds</code>	replicability

The following arguments to `genoud` default to values that differ from those documented at `genoud` but can be overridden by specifying them explicitly in the ...:

argument	value	why?
<code>boundary.enforcement</code>	1 usually	2 can cause problems
<code>MemoryMatrix</code>	FALSE	runs faster
<code>print.level</code>	1	output is not that helpful for $\geq 2$
<code>P9mix</code>	1	to always accept the BFGS result
<code>BFGSburnin</code>	5	to have a little burnin
<code>max.generations</code>	1000	big number is often necessary
<code>project.path</code>	contains "Rotate.txt"	

The arguments to `genoud` that remain at their defaults but you may want to consider tweaking are `pop.size`, `wait.generations`, and `solution.tolerance`.

## Details

This help page should really only be used as a reminder for what the various choices are, which are normally indicated by leaving `criteria` and `methodArgs` unspecified and responding to pop-up menus. The vignette provides a step-by-step guide to the pop-up menus and formally defines the criteria; execute `vignette("FAiR")` to read it.

The basic problem is to choose a transformation of the factors that is optimal with respect to some intersection of criteria. Since the objective function is vector valued, lexical optimization is performed via a genetic algorithm, which is tantamount to constrained optimization; see `genoud`.

The following functions can be named as constraints but must not be the last element of `criteria`:

name	methodArgs	reminder of what function does
"no_factor_collapse"	<code>nfc_threshold</code>	to prevent factor collapse
"limit_correlations"	lower and upper	limits factor intercorrelations
"positive_manifold"	<code>pm_threshold</code>	forces "positive" manifold
"ranks_rows_1st"	<code>row_ranks</code>	row-wise ordering constraints
"ranks_cols_1st"	<code>col_ranks</code>	column-wise ordering constraints
"indicators_1st"	<code>indicators</code>	designate which is the best indicator of a factor
"evRF_1st"	none	restrict effective variance of reference factors
"evPF_1st"	none	restrict effective variance of primary factors
"h2_over_FC_1st"	none	communalities $\geq$ factor contributions
"no_neg_suppressors_1st"	<code>FC_threshold</code>	no negative suppressors
"gv_1st"	none	generalized variance of primary $\leq$ reference factors
"distinguishability_1st"	none	best indicators have no negative suppressors

In fact, "no\_factor\_collapse" is always included and is listed above only to emphasize that one must specify `methodArgs$nfc_threshold` to avoid seeing the associated pop-up menu. This restriction to avoid factor collapse makes it possible to utilize one of the following "analytic" criteria that would otherwise result in factor collapse much of the time. One of the following can be named as the **last** element of `criteria`:

name	methodArgs
"phi"	<code>c</code>
"varphi"	<code>weights</code>
"LS"	<code>eps, scale, and E</code>
"minimaximin"	

```

"geomin"          delta
"quartimin"      delta
"target"         Target
"pst"            Target
"oblimax"        delta
"simplimax"      k
"bentler"        delta
"cf"             kappa
"infomax"        delta
"mccammon"       delta
"oblimin"        gam

```

The first four are defined only on the reference structure matrix. The remainder are copied from the **GPArotation** package and have the same arguments, with the exception of "pst" which uses NA in the target matrix for untargeted cells rather than also specifying a weight matrix (which is called W in the **GPArotation** package). In addition, one can specify `methodArgs$matrix` as one of "PP", "RS", or "FC" to use the primary pattern, reference structure, or factor contribution matrix in conjunction with the criteria from the **GPArotation** package, although these criteria are technically defined with respect to the primary pattern matrix.

Row-standardization should not be necessary. Row-standardization was originally intended to counteract some tendencies in the transformation process that can now be accomplished directly through lexical (i.e. constrained) optimization. Nevertheless, Kaiser normalization divides each row of the preliminary primary pattern matrix by its length and Cureton-Mulaik normalization favors rows that are thought to have only one large loading after transformation. Both schemes are thoroughly discussed in Browne (2001), which also discusses most of the continuous analytic criteria available in **FAiR** with the exceptions of those in Thurstone (1935) and Lorenzo-Seva (2003).

It is not necessary to provide starting values for the parameters. But a matrix of starting values can be passed to through the `dots` to `genoud`. This matrix should have rows equal to the `pop.size` argument in `genoud` and columns equal the number of factors squared. The columns correspond to the cells of the transformation matrix in column-major order. In contrast to some texts, the transformation matrix in `Rotate` has unit-length columns, rather than unit-length rows.

### Value

An object of `FA.EFA-class`.

### Note

The underlying genetic algorithm will print a variety of output as it progresses. On Windows, you have to move the scrollbar periodically to flush the output to the screen. The output will look something like this

Generation number	First constraint	Second constraint	...	Last constraint	Analytic criterion
0	-1.0	-1.0	...	-1.0	double
1	-1.0	-1.0	...	-1.0	double
...	...	...	...	...	...
42	-1.0	-1.0	...	-1.0	double

The integer on the very left indicates the generation number. If it appears to skip one or more generations, that signifies that the best individual in the “missing” generation was no better than the best individual in the previous generation. The sequence of negative ones indicates that various constraints specified by the user are being satisfied by the best individual in the generation. The curious are referred to the source code and / or the, vignette, but for the most part users need not worry about them provided they are  $-1.0$ . If any but the last are not  $-1.0$  after the first few generations, there is a problem because no individual is satisfying all the constraints. The last number is a double-precision number and is typically the (logarithm of the) analytic criterion specified by the user. This number will, decrease, sometimes painfully slowly, sometimes intermittently, over the generations since the criterion is being minimized subject to the aforementioned constraints. Finally, do not be particularly concerned if there are messages indicating a gradient check has failed because there is no strong reason to expect the gradient of the (last) criterion with respect to all the cells of the transformation matrix will be particularly small

### Author(s)

Ben Goodrich

### References

- Browne, M.W. (2001) An overview of analytic rotation in exploratory factor analysis. *Multivariate Behavioral Research*, **36**, 111–150.
- Lorenzo-Seva, U. (2003) A factor simplicity index. *Psychometrika*, **68**, 49–60.
- Smith, G. A. and Stanley G. (1983) Clocking *g*: relating intelligence and measures of timed performance. *Intelligence*, **7**, 353–368.
- Thurstone, L. L. (1935) *The Vectors of Mind*. Cambridge University Press.
- Venables, W. N. and Ripley, B. D. (2002) *Modern Applied Statistics with S*. Fourth edition. Springer.

### See Also

[Factanal](#)

### Examples

```
## Example from Venables and Ripley (2002, p. 323)
## Previously from Bartholomew and Knott (1999, p. 68--72)
## Originally from Smith and Stanley (1983)
## Replicated from example(ability.cov)

man <- make_manifest(covmat = ability.cov)
res <- make_restrictions(man, factors = 2, model = "EFA")
efa <- Factanal(manifest = man, restrictions = res, impatient = TRUE)

show(efa); summary(efa)

# 'criteria' and 'methodArgs' would typically be left unspecified
# and equivalent choices would be made from the pop-up menus
efa.rotated <- Rotate(efa, criteria = list("phi"),
                     methodArgs = list(nfc_threshold = 0.25, c = 1.0))
```

```
summary(efa.rotated)
pairs(efa.rotated)
## See the example for Factanal() for more post-estimation commands
```

---

S3methodsFAiR            *S3 methods for "FA" objects*

---

## Description

These methods are technically S4 but are the result of making well-known S3 methods into S4 generic functions and defining methods for objects of [FA-class](#) and / or [restrictions-class](#). In any event, they provide somewhat standard post-estimation functions for factor analysis models.

## Usage

```
## S4 method for signature 'FA':
deviance(object)
## S4 method for signature 'FA':
df.residual(object)
## S4 method for signature 'restrictions':
df.residual(object)
## S4 method for signature 'FA':
fitted(object, reduced = TRUE, standardized = TRUE)
## S4 method for signature 'restrictions':
fitted(object, reduced = TRUE, standardized = TRUE)
## S4 method for signature 'FA':
influence(model)
## S4 method for signature 'FA':
model.matrix(object, standardized = TRUE)
## S4 method for signature 'FA':
pairs(x, ...)
## S4 method for signature 'FA':
residuals(object, standardized = TRUE)
## S4 method for signature 'FA':
rstandard(model)
## S4 method for signature 'FA':
simulate(object, nsim = 1, seed = NULL, standardized = TRUE, ...)
## S4 method for signature 'FA':
weights(object)
```

## Arguments

object	An object of <a href="#">FA-class</a> or <a href="#">restrictions-class</a> , as appropriate
model	An object of <a href="#">FA-class</a>
x	An object of <a href="#">FA-class</a>

<code>reduced</code>	Logical indicating whether communalities should be on the diagonal of the fitted matrix.
<code>standardized</code>	Logical indicating whether the matrix should be calibrated for standardized variables.
<code>nsim</code>	number of simulations
<code>seed</code>	seed to use for random number generation; if <code>NULL</code> the current seed is used
<code>...</code>	additional argument(s) for methods

### Details

The code for each of these methods is quite short. There are some other method definitions for objects that inherit from `restrictions-class` and `FA-class` but they differ only in implementation and not in their nature or their options.

### Value

<code>deviance</code>	returns the value of the discrepancy function
<code>df.residual</code>	returns the degrees of freedom
<code>fitted</code>	returns the model's estimate of the covariance or correlation matrix among manifest variables in common factor space
<code>influence</code>	returns a square matrix that is equal to <code>residuals() * weights()</code>
<code>model.matrix</code>	returns the sample covariance or correlation matrix among outcomes
<code>pairs</code>	returns nothing but plots the estimated reference structure correlations in a nice form
<code>residuals</code>	returns a square matrix that contains the difference between <code>model.matrix()</code> and <code>fitted()</code> and has uniquenesses along the diagonal (based on correlations by default)
<code>rstandard</code>	covariance residuals standardized by the standard deviations of the manifest variables
<code>weights</code>	returns a square matrix with the weights used in the discrepancy function. For Yates' weighted least squares estimator these weights are as defined in equation 188. For maximum likelihood estimation, these weights are proportional to the reciprocal of the crossproduct of the uniquenesses and are only approximately equal to the implied weights that would be used if minimizing the weighted sum of squared residuals. For ease of interpretation they are rescaled so that the mean weight is 1.0.

### Author(s)

Ben Goodrich

### References

Yates, A. (1987) *Multivariate Exploratory Data Analysis: A Perspective on Exploratory Factor Analysis*. State University of New York Press.

**See Also**

[loadings](#), [cormat](#), and [uniquenesses](#)

**Examples**

```
## See the example for Factanal()
```

---

S4GenericsFAiR

*S4 Generics & Methods for Package 'FAiR'*

---

**Description**

It is not necessary to understand this help page if one merely wants to estimate a factor analysis model. This help page is intended for those who want to modify or extend FAiR or otherwise want some idea of how **FAiR** works “behind the scenes”.

[Factanal](#) is just a wrapper around a call to [genoud](#) in the **rgenoud** package. These S4 generic functions are passed to various arguments of [genoud](#) as part of the optimization process. With the exception of the methods for `create_start`, it would be very unusual for any of these methods to be called directly by the user.

**Usage**

```
## S4 method for signature 'restrictions, manifest.basic':
fitS4(par, restrictions, manifest, lower, mapping_rule)
## S4 method for signature 'restrictions, manifest.basic':
bfgs_fitS4(par, restrictions, manifest, helper, lower)
## S4 method for signature 'restrictions, manifest.basic':
gr_fitS4(par, restrictions, manifest, helper, lower)
## S4 method for signature 'restrictions, manifest.basic':
bfgs_helpS4(initial, restrictions, manifest, done, lower)
## S4 method for signature 'restrictions, manifest.basic':
create_start(number, start, restrictions, manifest, reject)
```

**Arguments**

<code>par</code>	A numeric vector containing values for all the free parameters to be estimated, which corresponds to the <code>par</code> argument for <a href="#">genoud</a> and for <a href="#">optim</a> .
<code>initial</code>	Same as <code>par</code> .
<code>restrictions</code>	An object of <a href="#">restrictions-class</a> .
<code>manifest</code>	An object of <a href="#">manifest-class</a> .
<code>lower</code>	A small numeric scalar indicating the lower bound for positive definiteness or minimum uniqueness; see the corresponding argument to <a href="#">Factanal</a> .
<code>mapping_rule</code>	a logical indicating whether to invoke the mapping rule in semi-exploratory models. It takes the value of <code>FALSE</code> when called by <a href="#">optim</a> .

<code>helper</code>	A list that contains necessary information for the <code>bfgs_fitS4</code> and <code>gr_fitS4</code> methods that is passed to the <code>BFGShelp</code> argument for <code>genoud</code> .
<code>done</code>	A logical indicating whether the optimization has terminated or could terminate immediately if a stopping condition is met; see the documentation of the <code>BFGShelp</code> argument to <code>genoud</code> .
<code>number</code>	An integer representing the number of starting values to create
<code>start</code>	A numeric vector or matrix containing initial communality estimates
<code>reject</code>	Logical indicating whether to reject starting values that fail the constraints required by the model
<code>...</code>	Further arguments to be passed to downstream functions; not currently used.

## Details

The `fitS4` method is responsible for producing a complete numeric *vector* of fit criteria and is passed to the `fn` argument of `genoud`. The default method simply calls `restrictions2model` and then calls an internal function that evaluates all the functions in the `criteria` slot of the object of `restrictions-class`.

The `bfgs_fitS4` method produces a *scalar* fit criterion, and this method is passed to the `BFGSfn` argument of `genoud` and is in turned passed to the `fn` argument of `optim`. Usually, this criterion is the last criterion produced by the `fitS4` method, which is the discrepancy function. The `gr_fitS4` method produces the gradient of the function defined by the `bfgs_fitS4` method under traditional maximum likelihood estimation; otherwise it is an internal function. Either way, this function is passed to the `gr` argument of `genoud` and `optim`. Both `bfgs_fitS4` and `gr_fitS4` take an argument called `helper`, which is produced by the `bfgs_helpS4` method and corresponds to the `BFGShelp` argument of `genoud`. The `BFGShelp` method produces a list that contains information about the initial value of the genetic individual when `optim` is called. The `bfgs_fitS4` methods and `gr_fitS4` methods can behave differently depending on the contents of `helper`.

The `create_start` method creates a matrix of starting values that is then passed to the `starting.values` argument of `genoud`. If you think the starting values are inadequate in a particular situation, it is much easier to create a matrix of starting values in the global environment and pass it through the `...` of `Factanal` to the `starting.values` argument of `genoud` yourself. See `Factanal` for details on doing so.

## Value

<code>fitS4</code>	produces a numeric vector of fits
<code>bfgs_fitS4</code>	produces a numeric scalar as a fit
<code>gr_fitS4</code>	produces a numeric vector that is the gradient at the scalar fit
<code>bfgs_helpS4</code>	produces a list of ancillary material for <code>bfgs_fitS4</code> and <code>gr_fitS4</code>
<code>create_start</code>	produces a numeric matrix that constitutes the starting population

**Methods**

There are methods for every flavor of [restrictions-class](#), although in many cases the methods are simply inherited. Currently, there are only methods for objects of class "manifest.basic", which are inherited by objects of class "manifest.data" and "manifest.data.mcd". However, it is possible to tailor methods for different flavors of [manifest-class](#) or to create new classes that inherit from [restrictions-class](#) and write (some) methods for them.

**Author(s)**

Ben Goodrich

**See Also**

[Factanal](#), [manifest-class](#), [restrictions-class](#) and [genoud](#)

**Examples**

```
showMethods("fitS4")
showMethods("bfgs_fitS4")
showMethods("gr_fitS4")
showMethods("bfgs_helpS4")
showMethods("create_start")
```

---

summary.FA-class    *Class "summary.FA"*

---

**Description**

It is not necessary to understand this help page if one merely wants to estimate a factor analysis model. This help page is intended for those who want to modify or extend FAiR or otherwise want some idea of how FAiR works “behind the scenes”.

This class holds the output when `summary` is called on an object of [FA-class](#)

**Objects from the Class**

Objects can be created by calls of the form `new("summary.FA", ...)`. However, rarely if ever, would a user need to construct an object this way. The `summary` method does so internally.

**Slots**

**restrictions:** Object of [restrictions-class](#)

**draws:** List (possibly of empty) of arrays produced by a possible call to [FA2draws](#)

**order:** An integer vector of the same length as the number of first-order factors that can be used to reorder the factors when printed to the screen

**orthogonal:** A logical indicating whether the factors are orthogonal

**polarity:** An integer vector of the same length as the number of first-order factors whose elements are either 1 or  $-1$  that can be used to change the direction of one or more factors (after they have been reordered)

**conf.level:** A number between zero and one exclusive that governs the bounds of the confidence intervals

**standardized:** A logical that indicates whether the estimates are standardized to the correlation metric or unstandardized on the covariance metric

**call:** The call to `Factanal`.

### Methods

**show** signature(object = "summary.FA"): Prints the summary on the screen

**plot** signature(x = "summary.FA", y = "ANY"): Produces a heatmap

### Author(s)

Ben Goodrich

### See Also

[FA-class](#)

### Examples

```
## See also examples for Factanal()
showClass("summary.FA")
```

# Index

## \*Topic **classes**

- create\_FAobject, 3
- equality\_restriction-class, 7
- manifest-class, 27
- parameter-class, 36
- restrictions-class, 42
- summary.FA-class, 61

## \*Topic **manip**

- mapping\_rule, 30
- read.cefa, 39
- read.triangular, 40
- restrictions2Mathomatic, 48
- restrictions2RAM, 51

## \*Topic **methods**

- loadings, 15
- make\_manifest, 17
- make\_restrictions, 20
- parameter-class, 36
- restrictions2draws, 46
- restrictions2model, 49
- S3methodsFAiR, 57
- S4GenericsFAiR, 59

## \*Topic **models**

- Factanal, 9
- FAiR-package, 2
- GPA2FA, 13
- make\_restrictions, 20
- model\_comparison, 33
- Rotate, 52
- S4GenericsFAiR, 59

## \*Topic **multivariate**

- Factanal, 9
- FAiR-package, 2
- GPA2FA, 13
- make\_manifest, 17
- make\_restrictions, 20
- model\_comparison, 33
- restrictions2model, 49
- restrictions2RAM, 51

- Rotate, 52

- S4GenericsFAiR, 59

## \*Topic **package**

- FAiR-package, 2

- apply, 32

- bfgs\_fitS4, 10, 45

- bfgs\_fitS4 (S4GenericsFAiR), 59

- bfgs\_fitS4, restrictions, manifest.basic-method (S4GenericsFAiR), 59

- bfgs\_fitS4-methods (S4GenericsFAiR), 59

- bfgs\_helpS4, 10

- bfgs\_helpS4 (S4GenericsFAiR), 59

- bfgs\_helpS4, restrictions, manifest.basic-method (S4GenericsFAiR), 59

- bfgs\_helpS4-methods (S4GenericsFAiR), 59

- BIC, 34

- BIC, FA-method (create\_FAobject), 3

- cat, 48

- coef, 38

- coef, FA-method (loadings), 15

- coef, parameter.coef-method (loadings), 15

- coef, restrictions-method, 45

- coef, restrictions-method (loadings), 15

- confint, FA-method, 47

- confint, FA-method (create\_FAobject), 3

- confint, summary.FA-method (summary.FA-class), 61

- cormat, 38, 45, 59

- cormat (loadings), 15

- cormat, FA-method (loadings), 15

- cormat, FA.2ndorder-method (loadings), 15

- cormat, manifest.basic-method  
(*manifest-class*), 27
- cormat, parameter.cormat-method  
(*loadings*), 15
- cormat, restrictions-method  
(*loadings*), 15
- cormat, restrictions.2ndorder-method  
(*loadings*), 15
- cormat-methods (*loadings*), 15
- cov, 20
- cov.shrink, 19, 20, 25
- cov.wt, 19, 20
- CovMcd, 19
- covMcd, 20
- CovMcd-class, 18, 29
- create\_FAobject, 3, 4, 42, 45
- create\_FAobject, restrictions, manifest.basic-method  
(*create\_FAobject*), 3
- create\_FAobject-methods, 38
- create\_FAobject-methods  
(*create\_FAobject*), 3
- create\_start, 9, 11, 45
- create\_start (*S4GenericsFAiR*), 59
- create\_start, restrictions, manifest.basic-method  
(*S4GenericsFAiR*), 59
- create\_start-methods  
(*S4GenericsFAiR*), 59
- crossprod, 5
- deviance, FA-method  
(*S3methodsFAiR*), 57
- df.residual, FA-method  
(*S3methodsFAiR*), 57
- df.residual, restrictions-method, 45
- df.residual, restrictions-method  
(*S3methodsFAiR*), 57
- dsyMatrix-class, 29
- environment, 25
- equality\_restriction-class, 7, 37
- FA-class, 11, 14, 16, 17, 34, 39, 42, 46–48, 51, 57, 58, 61, 62
- FA-class, 45
- FA-class (*create\_FAobject*), 3
- FA.2ndorder-class  
(*create\_FAobject*), 3
- FA.EFA-class, 14, 53, 55
- FA.EFA-class (*create\_FAobject*), 3
- FA.general-class  
(*create\_FAobject*), 3
- FA2draws, 6, 46, 61
- FA2draws (*restrictions2draws*), 46
- FA2RAM, 7
- FA2RAM (*restrictions2RAM*), 51
- Factanal, 3–5, 7, 9, 14, 17, 20, 25, 26, 34, 36, 39, 42, 48, 50, 53, 56, 59–62
- factanal, 9, 10, 24, 34, 43
- FAiR (*FAiR-package*), 2
- FAiR-package, 49
- FAiR-package, 2
- fitS4, 10, 45
- fitS4 (*S4GenericsFAiR*), 59
- fitS4, restrictions, manifest.basic-method  
(*S4GenericsFAiR*), 59
- fitS4-methods (*S4GenericsFAiR*), 59
- fitted, FA-method (*S3methodsFAiR*), 57
- fitted, restrictions-method, 45
- fitted, restrictions-method  
(*S3methodsFAiR*), 57
- formals, 32
- genoud, 2, 4, 5, 9–11, 25, 38, 43, 45, 50, 53–55, 59–61
- GPA, 14
- GPA2FA, 7, 13, 44
- GPFoblq, 14
- gr\_fitS4, 45
- gr\_fitS4 (*S4GenericsFAiR*), 59
- gr\_fitS4, restrictions, manifest.basic-method  
(*S4GenericsFAiR*), 59
- gr\_fitS4-methods  
(*S4GenericsFAiR*), 59
- hetcor, 19, 20
- influence, FA-method  
(*S3methodsFAiR*), 57
- loadings, 6, 15, 45, 59
- loadings, ANY-method (*loadings*), 15
- loadings, FA-method (*loadings*), 15
- loadings, FA.general-method  
(*loadings*), 15
- loadings, restrictions-method  
(*loadings*), 15

- loadings, restrictions.general-method (loadings), 15
- loadings-methods (loadings), 15
- logLik, FA-method (create\_FAobject), 3
- make\_manifest, 3, 9, 11, 12, 17, 20, 29, 30, 40, 41, 43
- make\_manifest, data.frame, missing, missing-method (make\_manifest), 17
- make\_manifest, formula, data.frame, missing-method (make\_manifest), 17
- make\_manifest, matrix, missing, missing-method (make\_manifest), 17
- make\_manifest, missing, data.frame, missing-method (make\_manifest), 17
- make\_manifest, missing, matrix, missing-method (make\_manifest), 17
- make\_manifest, missing, missing, CovMcd-method (make\_manifest), 17
- make\_manifest, missing, missing, hetcor-method (make\_manifest), 17
- make\_manifest, missing, missing, list-method (make\_manifest), 17
- make\_manifest, missing, missing, matrix-method (make\_manifest), 17
- make\_manifest-methods (make\_manifest), 17
- make\_parameter, 51
- make\_parameter (parameter-class), 36
- make\_parameter-methods, 42, 50
- make\_parameter-methods (parameter-class), 36
- make\_restrictions, 3, 7-9, 11, 12, 17, 19, 20, 20, 30, 32, 33, 36, 42, 45, 52
- make\_restrictions, manifest.basic, ANY, ANY, ANY, ANY, ANY-method (make\_restrictions), 20
- make\_restrictions, manifest.basic, missing, missing-method (make\_restrictions), 20
- make\_restrictions, manifest.basic, missing, missing-method (make\_restrictions), 20
- make\_restrictions, manifest.basic, missing, missing, parameter.coef, missing, missing-method (make\_restrictions), 20
- make\_restrictions, manifest.basic, missing, missing, parameter.coef, missing, missing-method (S3methodsFAiR), 57
- make\_restrictions, manifest.basic, missing, missing, parameter.coef, parameter-class, 21, 23, 43
- make\_restrictions, manifest.basic, missing, missing, parameter.coef, parameter-class, 22, 44
- make\_restrictions, manifest.basic, missing, missing, parameter.coef, parameter-class, 43, 44
- make\_restrictions, manifest.basic, parameter.scale (make\_restrictions), 20
- make\_restrictions, manifest.basic, parameter.scale (make\_restrictions), 20
- make\_restrictions, manifest.basic, parameter.scale (make\_restrictions), 20
- make\_restrictions, manifest.basic, parameter.scale (make\_restrictions), 20
- make\_restrictions, manifest.basic, parameter.scale (make\_restrictions), 20
- make\_restrictions-methods, 18, 37
- make\_restrictions-methods (make\_restrictions), 20
- manifest-class, 4, 6, 9, 17, 20, 50, 59, 61
- manifest-class, 5, 27
- manifest.basic-class, 22
- manifest.basic-class (manifest-class), 27
- manifest.basic.userW-class (manifest-class), 27
- manifest.data-class (manifest-class), 27
- manifest.data.mcd-class (manifest-class), 27
- manifest.data.ordinal-class (manifest-class), 27
- manifest.data.ranks-class (manifest-class), 27
- mapping\_rule, 30, 38
- mlest, 20
- model.matrix, FA-method (S3methodsFAiR), 57
- model.matrix, manifest.basic-method (manifest-class), 27
- model\_comparison, 3, 6, 24, 33
- na.action, 19
- optim, 10, 50, 53, 59, 60
- paired\_comparison, 6
- paired\_comparison, missing, missing-method (model\_comparison), 33
- parameter.coef, missing, missing-method (parameter.coef), 57
- parameter.coef, missing, missing-method (S3methodsFAiR), 57
- parameter-class, 21, 23, 43
- parameter-class, 22, 44
- parameter.class, 43, 44

- parameter.coef-class  
(parameter-class), 36
- parameter.coef.nl-class, 24, 44
- parameter.coef.nl-class, 43, 44
- parameter.coef.nl-class  
(parameter-class), 36
- parameter.coef.SEFA-class, 24, 33, 44
- parameter.coef.SEFA-class, 43
- parameter.coef.SEFA-class  
(parameter-class), 36
- parameter.coef.SEFA.nl-class, 44
- parameter.coef.SEFA.nl-class, 43
- parameter.coef.SEFA.nl-class  
(parameter-class), 36
- parameter.cormat-class, 22
- parameter.cormat-class, 43
- parameter.cormat-class  
(parameter-class), 36
- parameter.scale-class, 22, 24
- parameter.scale-class, 43, 44
- parameter.scale-class  
(parameter-class), 36
- plot, FA, missing-method  
(create\_FAobject), 3
- plot, manifest.basic, missing-method  
(manifest-class), 27
- plot, summary.FA, ANY-method  
(summary.FA-class), 61
- predict, FA-method  
(create\_FAobject), 3
- print.loadings, 16
- profile, FA-method  
(create\_FAobject), 3
  
- read.CEFA (read.cefa), 39
- read.cefa, 2, 39
- read.fwf, 40
- read.moments, 40
- read.spss, 2
- read.table, 2, 40
- read.triangular, 2, 40, 40
- residuals, FA-method  
(S3methodsFAiR), 57
- restrictions-class, 3, 4, 6, 9, 16, 17, 20, 21, 23–26, 36, 47, 48, 50, 51, 57–61
- restrictions-class, 5, 42, 61
- restrictions.1storder-class, 8, 24
- restrictions.1storder-class  
(restrictions-class), 42
- restrictions.1storder.EFA-class  
(restrictions-class), 42
- restrictions.2ndorder-class, 5, 24
- restrictions.2ndorder-class  
(restrictions-class), 42
- restrictions.factanal-class, 10, 24
- restrictions.factanal-class  
(restrictions-class), 42
- restrictions.general-class, 5, 24
- restrictions.general-class  
(restrictions-class), 42
- restrictions.independent-class, 24
- restrictions.independent-class  
(restrictions-class), 42
- restrictions.orthonormal-class, 24
- restrictions.orthonormal-class  
(restrictions-class), 42
- restrictions2draws, 6, 45, 46
- restrictions2draws-methods  
(restrictions2draws), 46
- restrictions2Mathomatic, 48
- restrictions2mathomatic  
(restrictions2Mathomatic), 48
- restrictions2model, 42, 44, 49, 60
- restrictions2model, restrictions, manifest-method  
(restrictions2model), 49
- restrictions2model-methods  
(restrictions2model), 49
- restrictions2RAM, 7, 45, 51
- restrictions2RAM-methods  
(restrictions2RAM), 51
- Rotate, 3–5, 7, 12, 14, 23, 44, 47, 52
- rowSums, 32
- rstandard, FA-method  
(S3methodsFAiR), 57
  
- S3methodsFAiR, 6, 57
- S4GenericsFAiR, 30, 59
- sample, 19
- scan, 41
- screplot, FA-method  
(create\_FAobject), 3

screepplot, manifest.basic-method  
(*manifest-class*), 27

sem, 51, 52

show, 38, 42

show, equality\_restriction-method  
(*equality\_restriction-class*),  
7

show, FA-method (*create\_FAobject*),  
3

show, manifest.basic-method  
(*manifest-class*), 27

show, parameter.coef-method  
(*parameter-class*), 36

show, parameter.cormat-method  
(*parameter-class*), 36

show, restrictions-method  
(*restrictions-class*), 42

show, summary.FA-method  
(*summary.FA-class*), 61

simulate, 47

simulate, FA-method  
(*S3methodsFAiR*), 57

specify.model, 52

summary, 3

summary, FA-method  
(*create\_FAobject*), 3

summary.FA-class, 47

summary.FA-class, 61

uniquenesses, 45, 59

uniquenesses (*loadings*), 15

uniquenesses, FA-method  
(*loadings*), 15

uniquenesses, FA.general-method  
(*loadings*), 15

uniquenesses, restrictions-method  
(*loadings*), 15

uniquenesses-methods (*loadings*),  
15

vcov, FA-method (*create\_FAobject*),  
3

weights, FA-method  
(*S3methodsFAiR*), 57

write.CEFA (*read.cefa*), 39

write.cefa (*read.cefa*), 39