# A cheat-sheet for useful R functions

The purpose of this chapter is to summarize the usage of the three main R libraries that are discussed in this book: ggm (ref), lavaan (ref) and xxM (ref).

## The ggm (Graphical Gaussian Models) library

| Function | page |
|---|---|
| DAG() | |
| basiSet() | |
| dSep() | |
| shipley.test() | |

There is no R library that is dedicated to the d-sep test and its generalizations. However, the ggm library of R does have a number of useful functions that can help you. First, the basiSet() function can always be used to obtain the union basis set of d-separation claims that lies at the heart of the d-sep test. This function requires that you specify the DAG and this is done using the DAG() function. The dSep() function is not required but is a very useful little function for helping you test your understanding of this important concept. The shipley.test() function impliments the d-sep test for the special case in which your DAG only involves normally distributed variables and linear relationships.

### The DAG() function

The R function DAG(), for imputing a directed acyclic graph. The output of this function is the adjacency matrix of the DAG, i.e. a square Boolean matrix of order equal to the number of nodes of the graph and a one in position *(i,j)* if there is an arrow from *i* to *j* and zero otherwise. The rownames of the adjacency matrix are the nodes of the DAG.

DAG(..., order = FALSE)

Arguments

**… =** a sequence of model formulae, using the regression ("~") operator. For each formula the right-hand response defines an effect node (a child in the DAG) and the left-hand explanatory variables the parents of that node. If the regressions are not recursive (i.e. if there is a feedback loop in the DAG) then the function returns an error message.

**order =** logical, defaulting to FALSE. If TRUE then nodes of the DAG are permuted according to the topological order. If FALSE then nodes are in the order they first appear in the model formulae (from left to right). This argument is used for purely esthetic reasons.

Consider this simple causal chain model: X->Y->Z. To input this model as a DAG you would specify:

`My.DAG<-DAG(Y~X, Z~Y)` Note that this uses the same syntax as when specifying linear models in R. For each child node (dependent variable, endogenous variable) you specify: dependent variable ~ parent variable1 + parent variable2 etc.

Here are some examples :

```
DAG(y ~ x + z + u, x ~ u, z ~ u)

## A DAG with an isolated node (v):

DAG(v ~ v, y ~ x + z, z ~ w + u)

## There can be repetitions of the same dependent variable

DAG(y ~ x + u + v, y ~ z, u ~ v + z)

## Interactions are ignored

DAG(y ~ x*z + z*v, x ~ z)

## A cyclic graph returns an error!

## Not run: DAG(y ~ x, x ~ z, z ~ y)

## The order can be changed

DAG(y ~ z, y ~ x + u + v,  u ~ v + z)

## If you want to order the nodes (topological sort of the DAG)

DAG(y ~ z, y ~ x + u + v,  u ~ v + z, order=TRUE)
```

basiSet() function: basiSet(amat)

This function outputs the union basis set for the conditional independencies implied by a directed acyclic graph; i.e. a minimal set of independencies whose null probabilities are themselves independent, and that imply all the other ones in the DAG. This function is useful if you must apply the d-sep test manually rather than via the shipley.test() function. The output is a list of vectors representing the conditional independence statements forming the basis set. Each output vector contains the names of two non-adjacent nodes followed by the names of nodes in the conditioning set (which may be empty).

**Arguments**

**amat =** a square matrix with dimnames representing the adjacency matrix of a DAG. This is normally created via the DAG() function.

Example:

```
> basiSet(amat=DAG(B~A,C~B,D~B,E~C+D))

[[1]]

[1] "A" "D" "B"   # means A is d-sep D given B

[[2]]

[1] "A" "C" "B"

[[3]]

[1] "A" "E" "D" "C"

[[4]]

[1] "B" "E" "A" "D" "C" # means B is d-sep E given {A,D,C}

[[5]]

[1] "D" "C" "B"
```

The dSep() function: dSep(amat, first, second, cond)

This function takes, as input, an adjacency graph created using the DAG() function and a query about if some set of nodes in the DAG are d-separated from some other set of nodes, given a third set of conditioning nodes. The output is the answer in the form of a logical variable (TRUE/FALSE).

Arguments:

**amat =** a Boolean matrix with dimnames, representing the adjacency matrix of a directed acyclic graph. The function does not check if this is the case. See the function isAcyclic.

**first =** a vector representing a subset of nodes of the DAG. The vector is normally a character vector of the names of the variables matching the names of the nodes in rownames(amat). It can be also a numeric vector of indices.

**second =** a vector representing another subset of nodes of the DAG. The set second must be disjoint from first. The mode of second must match the mode of first.

**cond =** a vector representing a conditioning subset of nodes. The set cond must be disjoint from the other two sets and must share the same mode. If there are no conditioning nodes then cond = NULL.

**Examples**

```
dSep(DAG(y ~ x, x ~ z), first="y", second="z", cond = "x")

dSep(DAG(y ~ x, y ~ z), first="x", second="z", cond = NULL)
```

```
## Example using the DAG in Figure 3.1 :

> my.dag<-DAG(B~A,C~B,D~B,E~C+D)

> dSep(my.dag,first="A",second="C",cond="B")

[1] TRUE

> dSep(my.dag,first="A",second="C",cond=NULL)

[1] FALSE

> dSep(my.dag,first="A",second="C",cond="D")

[1] FALSE

> dSep(my.dag,first="B",second="E",cond=c("C","D"))

[1] TRUE
```

## The shipley.test() function: shipley.test(amat, S, n)

The function computes a simultaneous test of all independence relationships implied by a given Gaussian model (i.e. assuming normally distributed variables and linear relationships) defined according to a directed acyclic graph, based on the sample covariance matrix. The output is the Fisher C-statistic, the degrees of freedom and the null probability. Note that this function does not output the (partial) correlation coefficients and associated null probabilities for each element of the basis set.

Arguments

**amat =** a square Boolean matrix, of the same dimension as S, representing the adjacency matrix of a DAG. This is normally created via the DAG() function.

**S =** the sample covariance matrix, which must be a symmetric positive definite matrix. This is normally created via the cov() function of R.

**n =** the sample size, which is a positive integer.

Example, using data containing 100 lines that are found in a data frame called my.dat (note that the variable names of this data frame must match those in the DAG):

```
> my.dag<-DAG(B~A,C~B,D~B,E~C+D)

> shipley.test(amat=my.dag,S=cov(my.dat),n=100)

$ctest

[1] 14.37551
```

```
$df

[1] 10

$pvalue

[1] 0.1565421
```

## The lavaan library

As I write, the lavaan library is still under development and so the information here is based on version 3.1.2. The following information is likely to change in future versions (hopefully not too much) and so you should always check the help files. The key reference is Rosseel (2012). The general structure of an R session using the lavaan library, as presented in this book, has three parts:

**Comment [JS1]:** Yves Rosseel (2012). lavaan: An R Package for Structural Equation Modeling. Journal of Statistical Software, 48(2), 1-36.

1. The creation of a model object that specifies the model structure. The model structure is enclosed in quotes.
2. A call to the sem() function. The sem() function inputs the model object, the data object, and other arguments. Using this information, it fits the model to the data and calculates the various output statistics.
3. A call to one or several extractor functions in order to obtain various types of information about the model fit.

A simple example of this sequence is:

```
#input this simple model: X→Y→Z and save it as an object called "my.model"

my.model<-"Y~X

Z~Y"

# fit "my.model" to a data set called "input.data" using the sem() function

model.fit<-sem(model=my.model,data=input.data)

# obtain a summary of the model fit

summary(model.fit)
```

The next section summarizes the various details in lavaan related to specifying the model (i.e. model syntax), choosing values for the arguments of the sem() function that control the fitting of the model to the data, and the various extractor functions that allow you to see various details of the resulting fit.

**Specifying the model structure: model syntax**

**Formula types**

| Formula type | Operator | Example | Causal graph | Meaning |
|---|---|---|---|---|
| Latent variable definition | =~ | L =~ x+y | L→y<br>L→y | "*latent cause, is measured by*". Latent variable "L" *causes* and *is measured by* observed variables x and y. |
| regression | ~ | y ~ x | x→y | "*is caused by, is regressed on*". Observed variable y *is caused by*, and *is regressed on*, x. |
| (residual) (co)variance | ~~ | x ~~ y, x~~x | x←→y, x←→x | "*free covariance, free variance*". |
| intercept | | x ~ 1 | | "*estimate the intercept of*" x. |
| New parameter | := | Total:=a+b | | "*create a new free parameter*". *Create a new free parameter* called "Total" which is constrained to be the sum of old free parameters a and b. |

**Fixing a parameter value**

Whenever you specify a structural equation via the model syntax of lavaan you implicitly define free parameters. One exception is when you use the "=~" operator since the path coefficient of the first observed variable on the right hand side of the operator is fixed to unity by default; see "allowing the first indicator of a latent variable to be free". Thus, a model statement like "y~x+z" implicitly defines two free parameters which are the path coefficients associated with x and z.

In order to force a parameter to take a specific value ("fixing it") rather than allowing it to be estimated from the data, you "multiply" the variable by the desired fixed value. Thus, "y~x+6*z" means that the path coefficient associated with the variable "z" is fixed to a value of 6 and can't be changed during the process of parameter estimation.

Examples:

`y~1.5*x` → the path coefficient associated with x is fixed at a value of 1.5.

`y~~1*y` → the (residual) variance of y is fixed at a value of 1.

`Y~~0*x` → the covariance between (the residuals of) y and x is fixed at zero.

## Specifying starting values

The iterative process of estimating the values of free parameters requires specifying the initial (starting) values of these free parameters. By default, lavaan sets all starting values to unity. However, more complicated models can fail to converge and one reason for this is that the starting values were simply too far away from the final values. In such cases, one must supply better initial values. This is done by via the "start()" argument, which is "multiplied" to the variable.

Example:

`y~start(0.1)*x` →the path coefficient associated with the variable "x" is free but its starting value during the iterative fitting process is equal to 0.1.

`y~~start(10)*y` → the starting value of the free (residual) variance of y is equal to 10.

`L=~start(0.001)*x+y` → the starting value of the free path coefficient associated with x is 0.001 (see also "allowing the first indicator of a latent to be free".

## Specifying starting values in multigroup models

In order to fit a model involving more than one group, you need to have a grouping variable in the data frame. If you want the starting value of a parameter to be the same in all groups then simply give a single start value. To specify different start values for each group, you specify these as a vector: start(c(0.1,0.2,…))*x

## Preventing exogenous variables from freely covarying

By default, all exogenous variables in lavaan are assumed to have non-zero covariances. The default occurs because the default value of the argument "fixed.x" in the sem() function is: fixed.x=TRUE. This is a poor default choice and so you should always explicitly specify the state of these exogenous covariances based on your conception of the causal process. To do this, you must specify sem(…,fixed.x=FALSE) in the sem() function and then explicitly specify these covariances in the model syntax as either fixed (to zero or some other value) or free; see "fixing a parameter value".

Example:

```
My.model<-"z~x+y
```

```
# the next line fixes the covariance between the two exogenous variables to
#zero

x~~0*y "

sem(model.syntax=My.model, data=, fixed.x=FALSE)
```

**Specifying parameter labels**

Every parameter in your model syntax has a name. For example, these are the parameter names that you see when you use the summary() function. The default name for a parameter is simply a concantination of variable name 1 + operator + variable name 2. In other words if, in your model syntax, you have a line like "y~x+z" then the first path coefficient is named "y~x" and the second path coefficient is named "y~z".

However, you can specify your own names for these parameters. To do this, simply "multiply" the variable name, using the usual naming conventions of R, by the label that you want to use for its associated parameter.

Examples:

`y ~ x + z` → the (free) path coefficient associated with variable x is called "y~x" and the (free) path coefficient associated with variable z is called "y~z".

`y ~ a*x + b*z` →the (free) path coefficient associated with variable x is called "a" and the (free) path coefficient associated with variable z is called "b".

You can combine these "multiplication" conventions. For instance: y ~ start(2)*a*x both specifies a starting value and a label name for the free path coefficient associated with x.

**Specifying parameter labels in multiple groups**

In order to fit a multigroup model, you need to have a grouping variable in the data frame. If you want different labels for each group then specify these different labels as a vector whose length is equal to the number of groups in the model: c(ag1,ag2,…)*x. BE CAREFUL; if you give the same label to more than one group then this will force the parameter estimation be equal across these groups sharing the same label name.

**Specifying simple equality constraints**

You can force combinations of free parameters to be equal during model fitting. In such cases, the fitted values of the parameters are still chosen so as to minimize the difference between the observed and model covariance matrices, but the chosen values are constrained to be equal to each other. This is most often done when fitting multigroup models but can be done whenever your causal hypothesis requires it. There are different but equivalent ways of doing this.

1: Simply give the same parameter label to the parameters whose values are to be equal. For example, y ~ a*x + a*z, will force the values of the estimated path coefficients associated with both variables x and z to be equal since the labels of both parameters to be equal.

2. Use the "equal()" function. For example, y ~ x + equal("y~x")*z forces the fitted value of the path coefficient associated with variable z to equal the value associated with variable x.

3. Use the "==" operator:

```
y ~ a*x + b*z

a == b
```

## Specifying nonlinear equality or inequality constraints

1. Give explicit labels to the parameters in question
2. Specify the desired constraint using the "==", "<" or ">" operators and the parameter labels
```
# give names (a1, a2, a3) to the three free path coefficients
y ~ a1*x + a2*z +a3*e
# here is a nonlinear equality constraint
a1 == (a2 + a3)^2
# here is a nonlinear inequality constraint
a1 > exp(a2 + a3)
```

## Preventing a free variance from being negative

By definition, a variance cannot be negative but the various algorithms used by lavaan to estimate parameter values don't know this fact. As a result, it sometimes happens that the estimated value of free residual variance is negative. This is usually a sign of a poorly fitting model or some problem that has occurred during the iterative process of estimation. However, it is possible that the model fits the data well but that the true value of the residual variance is very close to zero. If this happens then the estimate can become negative because of sampling fluctuations. If you think that this is the case then you force lavaan to maintain non-negative variance estimates by specifying a nonlinear constraint on this residual variance as follows:

```
# name the parameter label for the variance

x ~~ varx*x

# force this parameter value to remain non-negative
```

```
varx >= 0
```

## Calculating compound (indirect, total) effects

To calculate compound effects, such as indirect or total effects, and their standard errors, you must first give labels to the coefficients in question and then use the ":=" operator to calculate the desired compound effects. For instance, consider the simple path model: x—(a)→y, y—(b)→z, x—(c)→z where a, b and c are the label names for the three direct effects. There is both a direct effect of x on z (x→z) and an indirect effect (x→y→z). Only the direct effect is calculated by default in lavaan (i.e. the value of the path coefficient associated with variable z). To calculate the direct, indirect and total effects of x on z you would do:

```
My.model<-"

# give label names for the path coefficients

y ~ a*x

z ~c*x  + b*y

# define the parameter measuring the indirect effect (a*b)

indirect.effect := a*b

# define the parameter measuring the total effect

total.effect := c + (a*b)"
```

When you fit this model then the values and standard errors of the two new defined parameters (indirect.effect and total.effect) will be output.

## Allowing the first indicator of a latent to be free

When specifying a latent variable via the "=~" operator, the default in lavaan is to fix the path coefficient of the first indicator variable on the right hand side of the operator to unity in order to fix the scale of the latent. If you want to fix the scale of the latent in this way, then you don't have to do anything except to make sure that the first observed variable on the right hand side is the variable whose scale you want to use. However, if you don't want to do this (for instance, you want to identify the latent by fixing its variance to unity) then the usual (and easiest) way is to explicitly fix the latent variance to unity via the "std.lv" argument in the sem() function: sem(…,std.lv=TRUE). This fixes the standard deviation of all of the latent variables in the model to unity.

However, there are times in which this method is not appropriate. For instance, you might have more than one latent in your model whose scales you want to fix in different ways. Alternatively, you might want to fix the scale of the latent using some value of an observed scale other than unity. In such instances, you can explicitly force that the path coefficient of the first observed variable on the right hand side of the "=~" operator be free by "multiplying" it by NA (to free it) or by a number (to fix its scale to a value other than unity).. Thus, L =~ NA*x + … tells lavaan that the path coefficient associated with x (the first observed variable on the right hand side) is NOT fixed. Similarly, L=~2.5x+… tells lavaan that the path coefficient associated with x is fixed at 2.5.

**Arguments used when fitting the model via sem()**

The sem() function is actually a wrapper for another, more general function, called lavaan(). There are two other wrapper functions, called cfa() and growth(), that I won't discuss here. The sem() function contains very many arguments and several of these arguments deal with advanced topics that are not discussed in this book. Most of these arguments have default values and there are complicated interactions between these default values affecting things like the method of parameter estimation, the types of test statistics that are calculated, and so on. There are also some arguments that deal with advanced topics that have not been discussed in this book. Which of these arguments can be safely kept at their default values, and which need to be specified will depend on the complexity of your model. I have indicated those arguments that refer to topics that are discussed in this book with an asterisk.

Here is the full function, which you will see if you use type `help(sem)` in R:

```
sem(model = NULL, data = NULL,
    meanstructure = "default", fixed.x = "default",
    orthogonal = FALSE, std.lv = FALSE,
    parameterization = "default", std.ov = FALSE,
    missing = "default", ordered = NULL,
    sample.cov = NULL, sample.cov.rescale = "default",
    sample.mean = NULL, sample.nobs = NULL,
    ridge = 1e-05, group = NULL,
    group.label = NULL, group.equal = "", group.partial = "",
    group.w.free = FALSE, cluster = NULL, constraints = '',
    estimator = "default", likelihood = "default", link = "default",
    information = "default", se = "default", test = "default",
    bootstrap = 1000L, mimic = "default", representation = "default",
    do.fit = TRUE, control = list(), WLS.V = NULL, NACOV = NULL,
    zero.add = "default", zero.keep.margins = "default",
    zero.cell.warn = TRUE,
    start = "default", verbose = FALSE, warn = TRUE, debug = FALSE)
```

**model**$^*$ = the name of the object holding the model description. This must always be specified.

**Data**$^*$ = the name of the data frame holding the observations, including (if applicable) the grouping structure. You must always either provide this data frame or else provide (1) the covariance matrix via the sample.cov argument, (2) the vector of sample means for each variable via the sample.mean argument and (3) the number of observations used to calculate the sample covariance matrix via the sample.obs argument.  These last three arguments are described below.

**meanstructure*** = FALSE by default; if TRUE then the means (intercepts) are also modelled.

**fixed.x*** = If TRUE, the exogenous variables are not considered random variables and so the means, variances and covariances of these variables are not estimated but are rather fixed to their sample values. This is different from the way these variables are treated in this book.  If FALSE (which is the choice for the way they are treated in this book), they are considered random, and the means, variances and covariances are free parameters. If "default", the value is set depending on the mimic option (see below).

**orthogonal* =** If TRUE, the exogenous latent variables are assumed to be uncorrelated; i.e. the covariances between them are fixed at zero.

**std.ov* = TRUE** (the default is FALSE) only if you want all observed variables to be standardized (unit variance, zero mean) before the analysis.  This would give standardized coefficients.

**parameterization =** an argument used to treat categorical data and not discussed in this book.

**missing =** If "listwise", cases with missing values are removed listwise from the data frame before analysis. If "direct" or "ml" or "fiml" and the estimator (see below) is maximum likelihood, Full Information Maximum Likelihood (FIML) estimation is used using all available data in the data frame. This is only valid if the data are *missing completely at random* (MCAR) or *missing at random* (MAR). If "default", the value is set depending on the estimator and the mimic option (see below). This is only justified if the missing values are given the precise definitions of the terms "*missing at random*" or *"completely at random"* are explained in Rubin () and Schafer ().  A value is missing *completely at random* if its probability of being missing is unrelated to any other variable – observed or unobserved. For example, if you missed certain values because your measuring devise broke down one day then the pattern of missed values is probably missing completely at random. Missing values of a variable are said to be *missing at random* if the values of the other variables in the data set can predict the pattern of missingness.  If you are missing data on seed output because some plants already shed their seeds before you started measurements, but you also have information on (say) the date of flowering, then this would probably be a case of *missing at random*.   If, however, you have no other information in your data set that is related to the phenology of reproduction, then your missing values would not accord with this necessary assumption.

**Comment [JS2]:** Rubin, D.B. (1976) Inference and missing data. *Biometrika*, 63, 581-592.
Rubin, D.B. (1987) *Multiple Imputation for Nonresponse in Surveys*. J. Wiley & Sons, New York.
Schafer, J.L. (1997) *Analysis of Incomplete Multivariate Data*. Chapman & Hall, London.

**ordered =** character vector and only used if the data is in a data frame. Treat these variables as ordered (ordinal) variables, if they are endogenous in the model. Importantly, all other variables will be treated as numeric (unless they are declared as ordered in the original data frame).

**sample.cov** = a sample covariance if you want to input this instead of the actual data set.

**sample.cov.rescale =** If TRUE, the sample covariance matrix provided by the user is internally rescaled by multiplying it with a factor (N-1)/N. If "default", the value is set depending on the estimator and the likelihood option: it is set to TRUE if maximum likelihood estimation is used and likelihood="normal", and FALSE otherwise.

**sample.mean =** a sample mean vector if you want to model means and you have input the sample covariance instead of the actual data set.

**sample.nobs=** the number of observations used to calculate the sample covariance matrix if you have input this instead of the actual data set.

**ridge =** a small numeric constant used for ridging.  This is only used if the sample covariance matrix is non positive definite.

**group**[*] = the name of the variable in your data frame that codes the group names (only if you are doing a multigroup model).  If you do not specify this argument then lavaan assumes that you have only one group.

**group.label**[*] **=** a character vector. You can use this to specify which group (or factor) levels need to be selected from the grouping variable, and in which order. If NULL (the default), all grouping levels are selected, in the order as they appear in the data.

**group.equal**[*] = a vector of character strings. This is only used in a multigroup analysis and is used to specify the pattern of equality constraints across multiple groups.  Choices can be one or more of the following:

> =“loadings” means that the path coefficients from latents to indicators are equal across groups, as specified by the “=~” operator in the model syntax
>
> =”regressions” means that all regression coefficients are equal across groups, as specified by the “ ~” operator in the model syntax
>
> =”residuals” means that the residual variances of the observed variables are equal across groups
>
> =”residual.covariances” means that the covariances of the observed variables are equal
>
> =”lv.variances” means that the (residual) variances of the latents are equal
>
> =”lv.covariances” means that the (residual) covariances of the latent variables are equal
>
> =”means” means that the intercepts/means of the latent variables are equal
>
> =”intercepts” means that the intercepts of the observed variables are equal
>
> = “threshholds” refers to categorical variables and this is not discussed in this book.

**group.partial**[*] **=** a vector of character strings containing the labels of the parameters which should be free in all groups; this is used to override the group.equal argument for some specific parameters.

**group.w.free =** Logical. If TRUE, the group frequencies are considered to be free parameters in the model. In this case, a Poisson model is fitted to estimate the group frequencies. If FALSE (the default), the group frequencies are fixed to their observed values. This is not discussed in this book.

**cluster =** an argument that has not yet been implimented.

**constraints*** = additional (in)equality constraints not yet included in the model syntax. This is an alternative to including such constraints in the model syntax.

**estimator*** = The estimator to be used; the default is "ML" for maximum likelihood. There are many choices here, only some of which are discussed in this book. The options are: "ML" for maximum likelihood, "GLS" for generalized least squares, "WLS" for weighted least squares (sometimes called ADF estimation), "ULS" for unweighted least squares and "DWLS" for diagonally weighted least squares. These are the main options that affect the estimation. For convenience, the "ML" option can be extended as "MLM", "MLMV", "MLMVS", "MLF", and "MLR". The estimation will still be plain "ML", but now with robust standard errors and a robust (scaled) test statistic. For "MLM", "MLMV", "MLMVS", classic robust standard errors are used (se="robust.sem"); for "MLF", standard errors are based on first-order derivatives (se="first.order"); for "MLR", 'Huber-White' robust standard errors are used (se="robust.huber.white"). In addition, "MLM" will compute a Satorra-Bentler scaled (mean adjusted) test statistic (test="satorra.bentler") , "MLMVS" will compute a mean and variance adjusted test statistic (Satterthwaite style) (test="mean.var.adjusted"), "MLMV" will compute a mean and variance adjusted test statistic (scaled and shifted) (test="scaled.shifted"), and "MLR" will compute a test statistic which is asymptotically equivalent to the Yuan-Bentler T2-star test statistic. Analogously, the estimators "WLSM" and "WLSMV" imply the "DWLS" estimator (not the "WLS" estimator) with robust standard errors and a mean or mean and variance adjusted test statistic. Estimators "ULSM" and "ULSMV" imply the "ULS" estimator with robust standard errors and a mean or mean and variance adjusted test statistic.

**likelihood =** Only relevant for ML estimation. If "wishart", the wishart likelihood approach is used. In this approach, the covariance matrix has been divided by N-1, and both standard errors and test statistics are based on N-1; this is the approach described in this book. If "normal", the normal likelihood approach is used. Here, the covariance matrix has been divided by N, and both standard errors and test statistics are based on N. If "default", it depends on the mimic option: if mimic="lavaan" or mimic="Mplus", normal likelihood is used; otherwise, wishart likelihood is used.

**link =** Currently only used if the chosen estimator is MML and you have binary or ordered observed endogenous variables; this topic has not been discussed in this book. If "logit", a logit link is used for binary and ordered observed variables. If "probit", a probit link is used. If "default", it is currently set to "probit".

**information =** If "expected", the expected information matrix is used (to compute the standard errors). If "observed", the observed information matrix is used. If "default", the value is set depending on the estimator and the mimic option.

**se =** specifies how the standard errors of the parameter estimates are to be calculated.  If "standard" (the default), conventional standard errors are computed based on inverting the (expected or observed) information matrix. If "first.order", standard errors are computed based on first-order derivatives. If "robust.sem", conventional robust standard errors are computed. If "robust.huber.white", standard errors are computed based on the 'mlr' (aka pseudo ML, Huber-White) approach.  If "robust", either "robust.sem" or "robust.huber.white" is used depending on the estimator, the mimic option, and whether the data are complete or not. If "boot" or "bootstrap", bootstrap standard errors are computed using standard bootstrapping (unless Bollen-Stine bootstrapping is requested for the test statistic; in this case bootstrap standard errors are computed using model-based bootstrapping).  If "none", no standard errors are computed.

**test*** **=** If "standard", a conventional chi-square test is computed. If "Satorra.Bentler", a Satorra-Bentler scaled test statistic is computed. If "Yuan.Bentler", a Yuan-Bentler scaled test statistic is computed. If "mean.var.adjusted" or "Satterthwaite", a mean and variance adjusted test statistic is compute. If "scaled.shifted", an alternative mean and variance adjusted test statistic is computed (as in Mplus version 6 or higher). If "boot" or "bootstrap" or "Bollen.Stine", the Bollen-Stine bootstrap is used to compute the bootstrap probability value of the test statistic. If "default", the value depends on the values of other arguments.

**bootstrap*** **=** Number of bootstrap draws, if bootstrapping is used.

**mimic =** If "Mplus", an attempt is made to mimic the Mplus program. If "EQS", an attempt is made to mimic the EQS program. If "default", the value is (currently) set to to "lavaan", which is very close to"Mplus".

**representation =** If "LISREL" the classical LISREL matrix representation is used to represent the model (using the all-y variant).

**do.fit =** If FALSE, the model is not fit, and the current starting values of the model parameters are preserved.

**control =** A list containing control parameters passed to the optimizer. By default, lavaan uses "nlminb". See the R help file of nlminb for an overview of the control parameters. A different optimizer can be chosen by setting the value of optim.method. For unconstrained optimization (the model syntax does not include any "==", ">" or "<" operators), the available options are "nlminb" (the default), "BFGS" and "L-BFGS-B". See the manpage of the optim function for the control parameters of the latter two options. For constrained optimization, the only available option is "nlminb.constr".

**WLS.V =** A user provided weight matrix to be used by estimator "WLS"; if the estimator is "DWLS", only the diagonal of this matrix will be used. For a multiple group analysis, a list with a weight matrix for each group. The elements of the weight matrix should be in the following order (if all data is continuous): first the means (if a meanstructure is involved), then the lower triangular elements of the covariance matrix including the diagonal, ordered column by column. In the categorical case: first the thresholds (including the means for continuous variables), then the slopes (if any), the variances of continuous variables (if any), and finally the lower triangular elements of the correlation/covariance matrix excluding the diagonal, ordered column by column.

**NACOV =** A user provided matrix containing the elements of (N times) the asymptotic variance-covariance matrix of the sample statistics. For a multiple group analysis, a list with an asymptotic variance-covariance matrix for each group. See the WLS.V argument for information about the order of the elements.

**zero.add =** A numeric vector containing two values. These values affect the calculation of polychoric correlations when some frequencies in the bivariate table are zero. The first value only applies for 2x2 tables. The second value for larger tables. This value is added to the zero frequency in the bivariate table. If "default", the value is set depending on the "mimic" option. By default, lavaan uses zero.add = c(0.5. 0.0).

**zero.keep.margins =** Logical. This argument only affects the computation of polychoric correlations for 2x2 tables with an empty cell, and where a value is added to the empty cell. If TRUE, the other values of the frequency table are adjusted so that all margins are unaffected. If "default", the value is set depending on the "mimic". The default is TRUE.

**zero.cell.warn =** Logical. Only used if some observed endogenous variables are categorical. If TRUE, give a warning if one or more cells of a bivariate frequency table are empty.

**start*** **=** If it is a character string, the two options are currently "simple" and "Mplus". In the first case, all parameter values are set to zero, except the factor loadings (set to one), the variances of latent variables (set to 0.05), and the residual variances of observed variables (set to half the observed variance). If "Mplus", we use a similar scheme, but the factor loadings are estimated using the fabin3 estimator (tsls) per factor. If start is a fitted object of class lavaan, the estimated values of the corresponding parameters will be extracted. If it is a model list, for example the output of the paramaterEstimates() function, the values of the est or start or ustart column (whichever is found first) will be extracted.

**verbose =** If TRUE, the function value is printed out during each iteration.

**warn =** If TRUE, some (possibly harmless) warnings are printed out during the iterations.

**debug =** If TRUE, debugging information is printed out.

**Extractor functions**

Once you have fit the model to the data using the sem() function and saved it as an object (we'll call it "fit"), you can extract different types of information about the resulting fit using different extractor functions.

summary(fit, `standardized=FALSE, fit.measures=FALSE, rsquare=FALSE, modindices=FALSE).` This is the basic extractor function that outputs (as defaults) information on convergence, the basic test statistics (which depend on which estimator you specified in the sem() function) and the parameter estimates and their standard errors. The additional arguments allow you to also obtain standardized parameter estimates, additional fit statistics, the proportion of the total variance ($R^2$) of the endogenous variables of the model that are explained, and modification indices (Lagrange multipliers).

coef(fit). This outputs the fitted coefficients only.

parameterEstimates(fit). This outputs the parameter estimates, their SE and confidence intervals.

standardizedSolution(fit). This outputs the standardized estimates of the parameters.

residuals(fit, type=”standardized”). This outputs the standardized differences between the observed and predicted covariance matrices.

AIC(fit), BIC(fit). These output the AIC or BIC values of the model.

fitMeasures(fit). This outputs all of the various fit measures.

inspect(fit,”r2”). This outputs the $R^2$ (proportion of variance explained) associated with each endogenous variable.