# The lavaan tutorial

Yves Rosseel
Department of Data Analysis
Ghent University (Belgium)

June 16, 2018

### Abstract

If you are new to **lavaan**, this is the place to start. In this tutorial, we introduce the basic components of lavaan: the model syntax, the fitting functions (cfa, sem and growth), and the main extractor functions (summary, coef, fitted, inspect). After we have provided two simple examples, we briefly discuss some important topics: meanstructures, multiple groups, growth curve models, mediation analysis, and categorical data. Along the way, we hope to give you just enough information to get you started (but no more).

## Contents

## 1   Before you start

Before you start, please read these points carefully:

- First of all, you must have a recent version (3.0.0 or higher) of R installed. You can download the latest version of R from this page: http://cran.r-project.org/.

- The lavaan package is not finished yet. But it is already very useful for most users, or so we hope. However, some important features that are currently *NOT* available in lavaan are:

  - full support for hierarchical/multilevel datasets (multilevel cfa, multilevel sem); however version 0.6-1 supports two-level cfa/sem with random intercepts only, for continuous complete data
  - full support for sampling (and other) weights (0.6-1 has sampling weights, but only for complete continuous and non-clustered data)
  - support for discrete latent variables (mixture models, latent classes)
  - Bayesian estimation (but see the blavaan package)

  We hope to add these features to lavaan in the near future (but please do not ask when).

- We consider the current version as *beta* software. This does NOT mean that you can not trust the results. We believe the results are accurate. It does mean that things may change when new versions come out. For example, we may change the name of the arguments in function calls. And we change the internals of the source code constantly. However, the model syntax is fairly mature and has been stable for a while.

- We do not expect you to be an expert in R. In fact, the lavaan package is designed to be used by users that would normally never use R. Nevertheless, it may help to familiarize yourself a bit with R, just to be comfortable with it. Perhaps the most important skill that you may need to learn is how to import your own datasets (perhaps in an SPSS format) into R. There are many tutorials on the web to teach you just that. Once you have your data in R, you can start specifying your model. We have tried very hard to make it as easy as possible for users to fit their models. Of course, if you have suggestions on how we can improve things, please let us know.

- This document is written for first-time users of the lavaan package. It is not a reference manual, nor does it contain technical material on how things are done in the lavaan package. These documents are currently under preparation.

- The lavaan package is free open-source software. This means (among other things) that there is no warranty whatsoever.

- If you need help, you can ask questions in the lavaan discussion group. Go to https://groups.google.com/d/forum/lavaan/ and join the group. Once you have joined the group, you can email your questions to lavaan@googlegroups.com. If you think you have found a bug, or if you have a suggestion for improvement, you can either email me directly (to alert me), post it to the discussion group (to discuss it), or open an issue on github (see https://github.com/yrosseel/lavaan/issues). The latter is useful once we have agreed it is a bug, and it should be fixed. If you report a bug, it is always very useful to provide a reproducible example (a short R script and some data).

## 2 Installation of the package

Since May 2010, the lavaan package is available on CRAN. Therefore, to install lavaan, simply start up R, and type:

```
install.packages("lavaan", dependencies=TRUE)
```

You can check if the installation was succesful by typing

```
library(lavaan)
```

When the package is loaded, a startup message will be displayed showing the version number, and a reminder that this is beta software:

```
This is lavaan 0.6-1
lavaan is BETA software! Please report any bugs.
```

If you see this message, you are ready to start.

# 3  The model syntax

At the heart of the lavaan package is the 'model syntax'. The model syntax is a description of the model to be estimated. In this section, we briefly explain the elements of the lavaan model syntax. More details are given in the examples that follow.

In the R environment, a regression formula has the following form:

```
y ~ x1 + x2 + x3 + x4
```

In this formula, the tilde ("~") is the regression operator. On the left-hand side of the operator, we have the dependent variable (`y`), and on the right-hand side, we have the independent variables, separated by the "+" operator. In lavaan, a typical model is simply a set (or system) of regression formulas, where some variables (starting with an '`f`' below) may be latent. For example:

```
 y ~ f1 + f2 + x1 + x2
f1 ~ f2 + f3
f2 ~ f3 + x1 + x2
```

If we have latent variables in any of the regression formulas, we must 'define' them by listing their (manifest or latent) indicators. We do this by using the special operator "=~", which can be read as *is measured by*. For example, to define the three latent variabels `f1`, `f2` and `f3`, we can use something like:

```
f1 =~ y1 + y2 + y3
f2 =~ y4 + y5 + y6
f3 =~ y7 + y8 + y9 + y10
```

Furthermore, variances and covariances are specified using a 'double tilde' operator, for example:

```
y1 ~~ y1  # variance
y1 ~~ y2  # covariance
f1 ~~ f2  # covariance
```

And finally, intercepts for observed and latent variables are simple regression formulas with only an intercept (explicitly denoted by the number '`1`') as the only predictor:

```
y1 ~ 1
f1 ~ 1
```

Using these four *formula types*, a large variety of latent variable models can be described. The current set of formula types is summarized in the table below.

| formula type | operator | mnemonic |
|---|---|---|
| latent variable definition | =~ | is measured by |
| regression | ~ | is regressed on |
| (residual) (co)variance | ~~ | is correlated with |
| intercept | ~ 1 | intercept |

A complete lavaan model syntax is simply a combination of these formula types, enclosed between *single* quotes. For example:

```
myModel <- ' # regressions
            y1 + y2 ~ f1 + f2 + x1 + x2
                 f1 ~ f2 + f3
                 f2 ~ f3 + x1 + x2

            # latent variable definitions
               f1 =~ y1 + y2 + y3
               f2 =~ y4 + y5 + y6
               f3 =~ y7 + y8 + y9 + y10
```

```
        # variances and covariances
          y1 ~~ y1
          y1 ~~ y2
          f1 ~~ f2

        # intercepts
          y1 ~ 1
          f1 ~ 1
      ,
```

You can type this syntax interactively at the R prompt, but it is much more convenient to type the whole model syntax first in an external text editor. And when you are done, you can copy/paste it to the R console. If you are using RStudio, open a new 'R script', and type your model syntax (and all other R commands needed for this session) in the source editor of RStudio. And save your script, so you can reuse it later on.

The code piece above will produce a model syntax object, called `myModel` that can be used later when calling a function that actually estimates this model given a dataset. Note that formulas can be split over multiple lines, and you can use comments (starting with the `#` character) and blank lines within the single quotes to improve the readability of the model syntax.

If your model syntax is rather long, or you need to reuse the model syntax over and over again, you may prefer to store it in a separate text file called, say, `myModel.lav`. This text file should be in a human readable format (not a Word document). Within R, you can then read the model syntax from the file as follows:

```
    myModel <- readLines("/mydirectory/myModel.lav")
```

The argument of `readLines` is the full path to the file containing the model syntax. Again, the model syntax object can be used later to fit this model given a dataset.

# 4   A first example: confirmatory factor analysis (CFA)

We start with a simple example of confirmatory factor analysis, using the `cfa()` function, which is a user-friendly function for fitting CFA models. The lavaan package contains a built-in dataset called `HolzingerSwineford1939`. See the help page for this dataset by typing

`?HolzingerSwineford1939`

at the R prompt. This is a 'classic' dataset that is used in many papers and books on Structural Equation Modeling (SEM), including some manuals of commercial SEM software packages. The data consists of mental ability test scores of seventh- and eighth-grade children from two different schools (Pasteur and Grant-White). In our version of the dataset, only 9 out of the original 26 tests are included. A CFA model that is often proposed for these 9 variables consists of three latent variables (or factors), each with three indicators:

- a *visual* factor measured by 3 variables: `x1`, `x2` and `x3`
- a *textual* factor measured by 3 variables: `x4`, `x5` and `x6`
- a *speed* factor measured by 3 variables: `x7`, `x8` and `x9`

The figure below contains a graphical representation of the three-factor model.
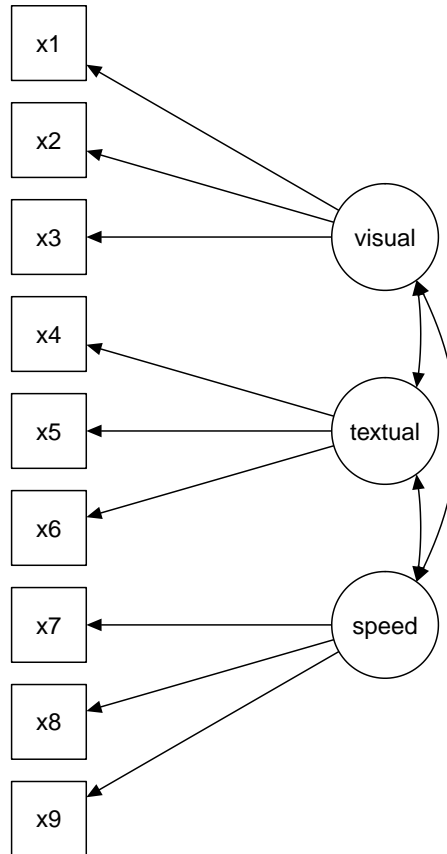
The corresponding lavaan syntax for specifying this model is as follows:

```
 visual =~ x1 + x2 + x3
textual =~ x4 + x5 + x6
  speed =~ x7 + x8 + x9
```

In this example, the model syntax only contains three 'latent variable definitions'. Each formula has the following format:

```
latent variable =~ indicator1 + indicator2 + indicator3
```

We call these expressions *latent variable definitions* because they define how the latent variables are 'manifested by' a set of observed (or manifest) variables, often called 'indicators'. Note that the special "=~" operator in the middle consists of a sign ("=") character and a tilde ("~") character next to each other. The reason why this model syntax is so short, is that behind the scenes, the function will take care of several things. First, by default, the factor loading of the first indicator of a latent variable is fixed to 1, thereby fixing the scale of the latent variable. Second, residual variances are added automatically. And third, all exogenous latent variables are correlated by default. This way, the model syntax can be kept concise. On the other hand, the user remains in control, since all this 'default' behavior can be overriden and/or switched off.
We can enter the model syntax using the single quotes:

```
HS.model <- ' visual  =~ x1 + x2 + x3
              textual =~ x4 + x5 + x6
              speed   =~ x7 + x8 + x9 '
```

We can now fit the model as follows:

```
fit <- cfa(HS.model, data=HolzingerSwineford1939)
```

The `cfa()` function is a dedicated function for fitting confirmatory factor analysis models. The first argument is the user-specified model. The second argument is the dataset that contains the observed variables. Once the model has been fitted, the `summary()` function provides a nice summary of the fitted model:

```
summary(fit, fit.measures=TRUE)
```

The output should look familiar to users of other SEM software. If you find it confusing or esthetically unpleasing, please let us know, and we will try to improve it.

```
lavaan (0.6-1) converged normally after  35 iterations

  Number of observations                           301

  Estimator                                         ML
  Model Fit Test Statistic                      85.306
  Degrees of freedom                                24
  P-value (Chi-square)                           0.000

Model test baseline model:

  Minimum Function Test Statistic              918.852
  Degrees of freedom                                36
  P-value                                        0.000

User model versus baseline model:

  Comparative Fit Index (CFI)                    0.931
  Tucker-Lewis Index (TLI)                       0.896

Loglikelihood and Information Criteria:

  Loglikelihood user model (H0)              -3737.745
  Loglikelihood unrestricted model (H1)      -3695.092

  Number of free parameters                         21
  Akaike (AIC)                                7517.490
  Bayesian (BIC)                              7595.339
  Sample-size adjusted Bayesian (BIC)         7528.739

Root Mean Square Error of Approximation:

  RMSEA                                          0.092
  90 Percent Confidence Interval          0.071  0.114
  P-value RMSEA <= 0.05                          0.001

Standardized Root Mean Square Residual:

  SRMR                                           0.065

Parameter Estimates:

  Information                                 Expected
  Information saturated (h1) model          Structured
  Standard Errors                             Standard

Latent Variables:
                   Estimate  Std.Err  z-value  P(>|z|)
  visual =~
```

```
    x1                1.000
    x2                0.554    0.100    5.554    0.000
    x3                0.729    0.109    6.685    0.000
  textual =~
    x4                1.000
    x5                1.113    0.065   17.014    0.000
    x6                0.926    0.055   16.703    0.000
  speed =~
    x7                1.000
    x8                1.180    0.165    7.152    0.000
    x9                1.082    0.151    7.155    0.000

Covariances:
                   Estimate  Std.Err  z-value  P(>|z|)
  visual ~~
    textual           0.408    0.074    5.552    0.000
    speed             0.262    0.056    4.660    0.000
  textual ~~
    speed             0.173    0.049    3.518    0.000

Variances:
                   Estimate  Std.Err  z-value  P(>|z|)
   .x1                0.549    0.114    4.833    0.000
   .x2                1.134    0.102   11.146    0.000
   .x3                0.844    0.091    9.317    0.000
   .x4                0.371    0.048    7.779    0.000
   .x5                0.446    0.058    7.642    0.000
   .x6                0.356    0.043    8.277    0.000
   .x7                0.799    0.081    9.823    0.000
   .x8                0.488    0.074    6.573    0.000
   .x9                0.566    0.071    8.003    0.000
    visual            0.809    0.145    5.564    0.000
    textual           0.979    0.112    8.737    0.000
    speed             0.384    0.086    4.451    0.000
```

The output consists of three parts. The first six lines are called *the header*. The header contains the following information:

- the lavaan version number
- did lavaan converge normally or not, and how many iterations were needed
- the number of observations that were effectively used in the analysis
- the estimator that was used to obtain the parameter values (here: ML)
- the model test statistic, the degrees of freedom, and a corresponding p-value

The next section contains additional fit measures, and is only shown because we use the optional argument `fit.measures = TRUE`. It starts with the line `Model test baseline model` and ends with the value for the SRMR. The last section contains the parameter estimates. It starts with information about the standard errors (if the information matrix is expected or observed, and if the standard errors are standard, robust, or based on the bootstrap). Then, it tabulates all free (and fixed) parameters that were included in the model. Typically, first the latent variables are shown, followed by covariances and (residual) variances. The first column (`Estimate`) contains the (estimated or fixed) parameter value for each model parameter; the second column (`Std.err`) contains the standard error for each estimated parameter; the third column (`Z-value`) contains the Wald statistic (which is simply obtained by dividing the parameter value by its standard error), and the last column (`P(>|z|)`) contains the p-value for testing the null hypothesis that the parameter equals zero in the population. To wrap up this first example, we summarize the complete code that was needed to fit this three-factor model:

```
# load the lavaan package (only needed once per session)
library(lavaan)

# specify the model
```

7

```
HS.model <- ' visual  =~ x1 + x2 + x3
              textual =~ x4 + x5 + x6
              speed   =~ x7 + x8 + x9 '

# fit the model
fit <- cfa(HS.model, data=HolzingerSwineford1939)

# display summary output
summary(fit, fit.measures=TRUE)
```

Simply copying this code and pasting it in R should work. The syntax illustrates the typical workflow in the lavaan package:

1. Specify your model using the lavaan model syntax. In this example, only *latent variable definitions* have been used. In the following examples, other formula types will be used.

2. Fit the model. This requires a dataset containing the observed variables (or alternatively the sample covariance matrix and the number of observations). In this example, we have used the `cfa()` function. Other funcions in the lavaan package are `sem()` and `growth()` for fitting full structural equation models and growth curve models respectively. All three functions are so-called user-friendly functions, in the sense that they take care of many details automatically, so we can keep the model syntax simple and concise. If you wish to fit non-standard models or if you don't like the idea that things are done for you automatically, you can use the lower-level function `lavaan()` instead, where you have full control.

3. Extract information from the fitted model. This can be a long verbose summary, or it can be a single number only (say, the RMSEA value). In the spirit of R, you only get what you asked for. We try to not print out unnecessary information that you would ignore anyway.

# 5   A second example: a structural equation model (SEM)

In our second example, we will use the built-in `PoliticalDemocracy` dataset. This is a dataset that has been used by Bollen in his 1989 book on structural equation modeling (and elsewhere). To learn more about the dataset, see its help page and the references therein.

The figure below contains a graphical representation of the model that we want to fit.
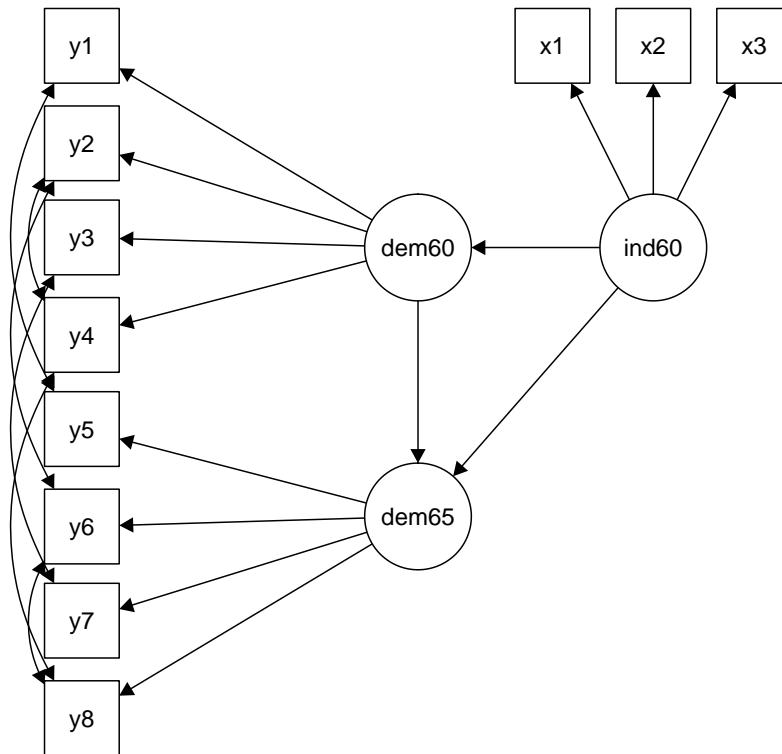
The corresponding lavaan syntax for specifying this model is as follows:

```
model <- '
  # measurement model
    ind60 =~ x1 + x2 + x3
    dem60 =~ y1 + y2 + y3 + y4
    dem65 =~ y5 + y6 + y7 + y8
  # regressions
    dem60 ~ ind60
    dem65 ~ ind60 + dem60
  # residual correlations
    y1 ~~ y5
    y2 ~~ y4 + y6
    y3 ~~ y7
    y4 ~~ y8
    y6 ~~ y8
'
```

In this example, we use three different formula types: latent variabele definitions (using the `=~` operator), regression formulas (using the `~` operator), and (co)variance formulas (using the `~~` operator). The regression formulas are similar to ordinary formulas in R. The (co)variance formulas typically have the following form:

```
variable ~~ variable
```

The variables can be either observed or latent variables. If the two variable names are the same, the expression refers to the variance (or residual variance) of that variable. If the two variable names are different, the expression refers to the (residual) covariance among these two variables. The lavaan package automatically makes the distinction between variances and residual variances.

In our example, the expression y1 ~~ y5 allows the residual variances of the two observed variables to be correlated. This is sometimes done if it is believed that the two variables have something in common that is not captured by the latent variables. In this case, the two variables refer to identical scores, but measured in two different years (1960 and 1965, respectively). Note that the two expressions y2 ~~ y4 and y2 ~~ y6, can be combined into the expression y2 ~~ y4 + y6. This is just a shorthand notation.
We enter the model syntax as follows:

```
model <- '
  # measurement model
    ind60 =~ x1 + x2 + x3
    dem60 =~ y1 + y2 + y3 + y4
    dem65 =~ y5 + y6 + y7 + y8
  # regressions
    dem60 ~ ind60
    dem65 ~ ind60 + dem60
  # residual correlations
    y1 ~~ y5
    y2 ~~ y4 + y6
    y3 ~~ y7
    y4 ~~ y8
    y6 ~~ y8
'
```

To fit the model and see the results we can type:

```
fit <- sem(model, data=PoliticalDemocracy)
summary(fit, standardized=TRUE)
```

```
lavaan (0.6-1) converged normally after  68 iterations

  Number of observations                          75

  Estimator                                       ML
  Model Fit Test Statistic                    38.125
  Degrees of freedom                              35
  P-value (Chi-square)                         0.329

Parameter Estimates:

  Information                               Expected
  Information saturated (h1) model        Structured
  Standard Errors                           Standard

Latent Variables:
                 Estimate  Std.Err  z-value  P(>|z|)   Std.lv  Std.all
  ind60 =~
    x1              1.000                               0.670    0.920
    x2              2.180    0.139   15.742    0.000    1.460    0.973
    x3              1.819    0.152   11.967    0.000    1.218    0.872
  dem60 =~
    y1              1.000                               2.223    0.850
    y2              1.257    0.182    6.889    0.000    2.794    0.717
    y3              1.058    0.151    6.987    0.000    2.351    0.722
    y4              1.265    0.145    8.722    0.000    2.812    0.846
  dem65 =~
    y5              1.000                               2.103    0.808
    y6              1.186    0.169    7.024    0.000    2.493    0.746
    y7              1.280    0.160    8.002    0.000    2.691    0.824
    y8              1.266    0.158    8.007    0.000    2.662    0.828
```

```
Regressions:
                   Estimate  Std.Err  z-value  P(>|z|)   Std.lv   Std.all
  dem60 ~
    ind60             1.483    0.399    3.715    0.000    0.447    0.447
  dem65 ~
    ind60             0.572    0.221    2.586    0.010    0.182    0.182
    dem60             0.837    0.098    8.514    0.000    0.885    0.885

Covariances:
                   Estimate  Std.Err  z-value  P(>|z|)   Std.lv   Std.all
 .y1 ~~
   .y5                0.624    0.358    1.741    0.082    0.624    0.296
 .y2 ~~
   .y4                1.313    0.702    1.871    0.061    1.313    0.273
   .y6                2.153    0.734    2.934    0.003    2.153    0.356
 .y3 ~~
   .y7                0.795    0.608    1.308    0.191    0.795    0.191
 .y4 ~~
   .y8                0.348    0.442    0.787    0.431    0.348    0.109
 .y6 ~~
   .y8                1.356    0.568    2.386    0.017    1.356    0.338

Variances:
                   Estimate  Std.Err  z-value  P(>|z|)   Std.lv   Std.all
   .x1               0.082    0.019    4.184    0.000    0.082    0.154
   .x2               0.120    0.070    1.718    0.086    0.120    0.053
   .x3               0.467    0.090    5.177    0.000    0.467    0.239
   .y1               1.891    0.444    4.256    0.000    1.891    0.277
   .y2               7.373    1.374    5.366    0.000    7.373    0.486
   .y3               5.067    0.952    5.324    0.000    5.067    0.478
   .y4               3.148    0.739    4.261    0.000    3.148    0.285
   .y5               2.351    0.480    4.895    0.000    2.351    0.347
   .y6               4.954    0.914    5.419    0.000    4.954    0.443
   .y7               3.431    0.713    4.814    0.000    3.431    0.322
   .y8               3.254    0.695    4.685    0.000    3.254    0.315
    ind60            0.448    0.087    5.173    0.000    1.000    1.000
   .dem60            3.956    0.921    4.295    0.000    0.800    0.800
   .dem65            0.172    0.215    0.803    0.422    0.039    0.039
```

The function `sem()` is very similar to the function `cfa()`. In fact, the two functions are currently almost identical, but this may change in the future. In the `summary()` function, we omitted the `fit.measures=TRUE` argument. Therefore, you only get the basic chi-square test statistic. The argument `standardized=TRUE` augments the output with standardized parameter values. Two extra columns of standardized parameter values are printed. In the first column (labeled `Std.lv`), only the latent variables are standardized. In the second column (labeled `Std.all`), both latent and observed variables are standardized. The latter is often called the 'completely standardized solution'.

The complete code to specify and fit this model is printed again below:

```r
library(lavaan) # only needed once per session
model <- '
  # measurement model
    ind60 =~ x1 + x2 + x3
    dem60 =~ y1 + y2 + y3 + y4
    dem65 =~ y5 + y6 + y7 + y8
  # regressions
    dem60 ~ ind60
    dem65 ~ ind60 + dem60
  # residual correlations
    y1 ~~ y5
    y2 ~~ y4 + y6
```

```
    y3 ~~ y7
    y4 ~~ y8
    y6 ~~ y8
'
fit <- sem(model, data=PoliticalDemocracy)
summary(fit, standardized=TRUE)
```

# 6  More about the syntax

**Fixing parameters**  Consider a simple one-factor model with 4 indicators. By default, lavaan will always fix the factor loading of the first indicator to 1. The other three factor loadings are free, and their values are estimated by the model. But suppose that you have good reasons the fix all the factor loadings to 1. The syntax below illustrates how this can be done:

```
f =~ y1 + 1*y2 + 1*y3 + 1*y4
```

In general, to fix a parameter in a lavaan formula, you need to pre-multiply the corresponding variable in the formula by a numerical value. This is called the pre-multiplication mechanism and will be used for many purposes. As another example, consider again the three-factor Holzinger and Swineford CFA model. Recall that, by default, all exogenous latent variables in a CFA model are correlated. But if you wish to fix the correlation (or covariance) between a pair of latent variables to zero, you need to explicity add a covariance-formula for this pair, and fix the parameter to zero. In the syntax below, we allow the covariance between the latent variables `visual` and `textual` to be free, but the two other covariances are fixed to zero. In addition, we fix the variance of the factor `speed` to unity. Therefore, there is no need anymore to set the factor loading of its first indicator (`x7`) equal to one. To force this factor loading to be free, we pre-multiply it with `NA`, as a hint to lavaan that the value of this parameter is still unknown.

```
# three-factor model
   visual =~ x1 + x2 + x3
  textual =~ x4 + x5 + x6
  speed   =~ NA*x7 + x8 + x9
# orthogonal factors
   visual ~~ 0*speed
  textual ~~ 0*speed
# fix variance of speed factor
     speed ~~ 1*speed
```

If you need to constrain all covariances of the latent variables in a CFA model to be orthogonal, there is a short-cut. You can omit the covariance formulas in the model syntax and simply add an argument `orthogonal=TRUE` to the function call:

```
HS.model <- '  visual =~ x1 + x2 + x3
             textual =~ x4 + x5 + x6
             speed   =~ x7 + x8 + x9 '

fit.HS.ortho <- cfa(HS.model,
                    data = HolzingerSwineford1939,
                    orthogonal = TRUE)
```

Similarly, if you want to fix the variances of *all* the latent variables in a CFA model to unity, there is again a shortcut. Simply add the argument `std.lv=TRUE` to the function call:

```
HS.model <- ' visual  =~ x1 + x2 + x3
             textual =~ x4 + x5 + x6
             speed   =~ x7 + x8 + x9 '
fit <- cfa(HS.model,
           data = HolzingerSwineford1939,
           std.lv = TRUE)
```

If the argument `std.lv=TRUE` is used, the factor loadings of the first indicator of each latent variable will no longer be fixed to 1.

**Starting Values**  The lavaan package automatically generates starting values for all free parameters. Normally, this works fine. But if you must provide your own starting values, you are free to do so. The way it works is based on the pre-multiplication mechanism that we discussed before. But the numeric constant is now the argument of a special function `start()`. An example will make this clear:

```
 visual =~ x1 + start(0.8)*x2 + start(1.2)*x3
textual =~ x4 + start(0.5)*x5 + start(1.0)*x6
speed   =~ x7 + start(0.7)*x8 + start(1.8)*x9
```

**Parameter labels**  A nice property of the lavaan package is that all free parameters are automatically named according to a simple set of rules. This is convenient, for example, if equality constraints are needed (see the next subsection). To see how the naming mechanism works, we will use the model that we used for the Politcal Democracy data.

```
model <- '
  # latent variable definitions
    ind60 =~ x1 + x2 + x3
    dem60 =~ y1 + y2 + y3 + y4
    dem65 =~ y5 + y6 + y7 + y8
  # regressions
    dem60 ~ ind60
    dem65 ~ ind60 + dem60
  # residual (co)variances
    y1 ~~ y5
    y2 ~~ y4 + y6
    y3 ~~ y7
    y4 ~~ y8
    y6 ~~ y8
'
fit <- sem(model,
           data = PoliticalDemocracy)
coef(fit)
```

```
    ind60=~x2     ind60=~x3     dem60=~y2     dem60=~y3     dem60=~y4
        2.180         1.819         1.257         1.058         1.265
    dem65=~y6     dem65=~y7     dem65=~y8    dem60~ind60   dem65~ind60
        1.186         1.280         1.266         1.483         0.572
   dem65~dem60       y1~~y5        y2~~y4        y2~~y6        y3~~y7
        0.837         0.624         1.313         2.153         0.795
       y4~~y8        y6~~y8        x1~~x1        x2~~x2        x3~~x3
        0.348         1.356         0.082         0.120         0.467
       y1~~y1        y2~~y2        y3~~y3        y4~~y4        y5~~y5
        1.891         7.373         5.067         3.148         2.351
       y6~~y6        y7~~y7        y8~~y8  ind60~~ind60  dem60~~dem60
        4.954         3.431         3.254         0.448         3.956
 dem65~~dem65
        0.172
```

The function `coef()` extracts the estimated values of the free parameters in the model, together with their names. Each name consists of three parts and reflects the part of the formula where the parameter was involved. The first part is the variable name that appears on the left-hand side of the formula. The middle part is the operator type of the formula, and the third part is the variable in the right-hand side of the formula that corresponds with the parameter.

Often, it is convenient to choose your own labels for specific parameters. The way this works is similar to fixing a parameter. But instead of pre-multiplying with a numerical constant, we use a character string (the label) instead. In the example below, we 'label' the factor loading of the `x3` indicator with the label `myLabel`:

```
model <- '
  # latent variable definitions
    ind60 =~ x1 + x2 + myLabel*x3
```

```
    dem60 =~ y1 + y2 + y3 + y4
    dem65 =~ y5 + y6 + y7 + y8
  # regressions
    dem60 ~ ind60
    dem65 ~ ind60 + dem60
  # residual (co)variances
    y1 ~~ y5
    y2 ~~ y4 + y6
    y3 ~~ y7
    y4 ~~ y8
    y6 ~~ y8
,
```

It is important that labels start with a letter (a-zA-Z), and certainly not with a digit. For example '13bis' is not a valid label, and will confuse the lavaan syntax parser. Note: before version 0.4-8, it was necessary to use the modifier `label()` to specify a custom label. Although it is still supported, it is not recommended anymore. The only reason why it should be used in new syntax is if the label contains an operator like "~" or "=~".

**Modifiers** We have seen the use of the pre-multiplication mechanism (using the `*` operator) a number of times: to fix a parameter, to provide a starting value, and to label a parameter. We refer to these operations as *modifiers*, because they modify some properties of certain model parameters. More modifiers will be introduced later.

Each term on the right hand side in a formula can have one modifier only. If you must specify more modifiers for the same parameter, you need to list the term multiple times in the same formula. For example:

```
f =~ y1 + y2 + myLabel*y3 + start(0.5)*y3 + y4
```

The indicator `y3` was listed twice, each time with a different modifier. The parser will accumulate all the different modifiers, but still treat `y3` as a single indicator.

**Simple equality constraints** In some applications, it is useful to impose equality constraints on one or more otherwise free parameters. Consider again the three-factor H&S CFA model. Suppose a user has a priori reasons to believe that the factor loadings of the `x2` and `x3` indicators are equal to each other. Instead of estimating two free parameters, lavaan should only estimate a single free parameter, and use that value for both factor loadings. The main mechanism to specify this type of (simple) equality constraints is by using labels: if two parameters have the same label, they will be considered to be the same, and only one value will be computed for them. This is illustrated in the following syntax:

```
 visual =~ x1 + v2*x2 + v2*x3
textual =~ x4 + x5 + x6
speed   =~ x7 + x8 + x9
```

Remember: all parameters having the same label will be constrained to be equal.
An alternative approach is to use the `equal()` modifier. This is useful if no custom label has been specified, and one needs to refer to the automatically generated label. For example:

```
 visual =~ x1 + x2 + equal("visual=~x2")*x3
textual =~ x4 + x5 + x6
speed   =~ x7 + x8 + x9
```

**Nonlinear equality and inequality constraints** Consider the following regression:

```
y ~ b1*x1 + b2*x2 + b3*x3
```

where we have explicitly labeled the regression coefficients as `b1`, `b2` and `b3`. We create a toy dataset containing these four variables and fit the regression model:

14

```
set.seed(1234)
Data <- data.frame(y = rnorm(100),
                   x1 = rnorm(100),
                   x2 = rnorm(100),
                   x3 = rnorm(100))
model <- ' y ~ b1*x1 + b2*x2 + b3*x3 '
fit <- sem(model, data=Data)
coef(fit)
```

```
    b1     b2     b3   y~~y
-0.052  0.084  0.139  0.970
```

Suppose that we need to impose the following two (nonlinear) constraints on $b_1$: $b_1 = (b_2 + b_3)^2$ and $b_1 \geq \exp(b_2 + b_3)$. The first constraint is an equality constraint. The second is an inequality constraint. To specify these constraints, you can use the following syntax:

```
model.constr <- ' # model with labeled parameters
                   y ~ b1*x1 + b2*x2 + b3*x3
                 # constraints
                   b1 == (b2 + b3)^2
                   b1 > exp(b2 + b3) '
```

To see the effect of the constraints, we refit the model:

```
model.constr <- ' # model with labeled parameters
                   y ~ b1*x1 + b2*x2 + b3*x3
                 # constraints
                   b1 == (b2 + b3)^2
                   b1 > exp(b2 + b3) '
fit <- sem(model.constr, data=Data)
coef(fit)
```

```
   b1     b2     b3   y~~y
 0.495 -0.405 -0.299  1.610
```

The reader can verify that the constraints are indeed respected. The equality constraint holds exactly. The inequality constraint has resulted in an equality between the left-hand side ($b_1$) and the right-hand side ($\exp(b_2 + b_3)$).

# 7   Bringing in the means

By and large, structural equation models are used to model the covariance matrix of the observed variables in a dataset. But in some applications, it is useful to bring in the means of the observed variables too. One way to do this is to explicitly refer to intercepts in the lavaan syntax. This can be done by including 'intercept formulas' in the model syntax. An intercept formula has the following form:

```
variable ~ 1
```

The left part of the expression contains the name of the observed or latent variable. The right part contains the number 1, representing the intercept. For example, in the three-factor H&S CFA model, we can add the intercepts of the observed variables as follows:

```
# three-factor model
   visual =~ x1 + x2 + x3
  textual =~ x4 + x5 + x6
  speed   =~ x7 + x8 + x9
# intercepts
  x1 ~ 1
  x2 ~ 1
  x3 ~ 1
```

```
x4 ~ 1
x5 ~ 1
x6 ~ 1
x7 ~ 1
x8 ~ 1
x9 ~ 1
```

However, it is more convenient to omit the intercept formulas in the model syntax (unless you want to fix their values), and to add the argument `meanstructure = TRUE` in the fitting function. For example, we can refit the three-factor H&S CFA model as follows:

```
fit <- cfa(HS.model,
           data = HolzingerSwineford1939,
           meanstructure = TRUE)
summary(fit)
```

```
lavaan (0.6-1) converged normally after  35 iterations

  Number of observations                            301

  Estimator                                          ML
  Model Fit Test Statistic                       85.306
  Degrees of freedom                                 24
  P-value (Chi-square)                            0.000

Parameter Estimates:

  Information                                  Expected
  Information saturated (h1) model           Structured
  Standard Errors                              Standard

Latent Variables:
                   Estimate  Std.Err  z-value  P(>|z|)
  visual =~
    x1                1.000
    x2                0.554    0.100    5.554    0.000
    x3                0.729    0.109    6.685    0.000
  textual =~
    x4                1.000
    x5                1.113    0.065   17.014    0.000
    x6                0.926    0.055   16.703    0.000
  speed =~
    x7                1.000
    x8                1.180    0.165    7.152    0.000
    x9                1.082    0.151    7.155    0.000

Covariances:
                   Estimate  Std.Err  z-value  P(>|z|)
  visual ~~
    textual           0.408    0.074    5.552    0.000
    speed             0.262    0.056    4.660    0.000
  textual ~~
    speed             0.173    0.049    3.518    0.000

Intercepts:
                   Estimate  Std.Err  z-value  P(>|z|)
   .x1                4.936    0.067   73.473    0.000
   .x2                6.088    0.068   89.855    0.000
   .x3                2.250    0.065   34.579    0.000
   .x4                3.061    0.067   45.694    0.000
```

```
    .x5                4.341    0.074   58.452   0.000
    .x6                2.186    0.063   34.667   0.000
    .x7                4.186    0.063   66.766   0.000
    .x8                5.527    0.058   94.854   0.000
    .x9                5.374    0.058   92.546   0.000
     visual            0.000
     textual           0.000
     speed             0.000

Variances:
                    Estimate  Std.Err  z-value  P(>|z|)
    .x1                0.549    0.114    4.833   0.000
    .x2                1.134    0.102   11.146   0.000
    .x3                0.844    0.091    9.317   0.000
    .x4                0.371    0.048    7.779   0.000
    .x5                0.446    0.058    7.642   0.000
    .x6                0.356    0.043    8.277   0.000
    .x7                0.799    0.081    9.823   0.000
    .x8                0.488    0.074    6.573   0.000
    .x9                0.566    0.071    8.003   0.000
     visual            0.809    0.145    5.564   0.000
     textual           0.979    0.112    8.737   0.000
     speed             0.384    0.086    4.451   0.000
```

As you can see in the output, the model includes intercept parameters for both the observed and latent variables. By default, the `cfa()` and `sem()` functions fix the latent variable intercepts (which in this case correspond to the latent *means*) to zero. Otherwise, the model would not be estimable. Note that the chi-square statistic and the number of degrees of freedom is the same as in the original model (without a mean structure). The reason is that we brought in some new data (a mean value for each of the 9 observed variables), but we also added 9 additional parameters to the model (an intercept for each of the 9 observed variables). The end result is an identical fit. In practice, the only reason why a user would add intercept-formulas in the model syntax, is because some constraints must be specified on them. For example, suppose that we wish to fix the intercepts of the variables `x1`, `x2`, `x3` and `x4` to, say, 0.5. We would write the model syntax as follows:

```
# three-factor model
   visual =~ x1 + x2 + x3
  textual =~ x4 + x5 + x6
  speed   =~ x7 + x8 + x9
# intercepts with fixed values
  x1 + x2 + x3 + x4 ~ 0.5*1
```

where we have used the left-hand side of the formula to 'repeat' the right-hand side for each element of the left-hand side.

# 8   Multiple groups

The lavaan package has full support for multiple groups. To request a multiple group analysis, you need to add the name of the group variable in your dataset to the argument `group` in the fitting function. By default, the same model is fitted in all groups. In the following example, we fit the H&S CFA model for the two schools (Pasteur and Grant-White).

```
HS.model <- '  visual =~ x1 + x2 + x3
             textual =~ x4 + x5 + x6
             speed   =~ x7 + x8 + x9 '
fit <- cfa(HS.model,
          data = HolzingerSwineford1939,
          group = "school")
summary(fit)


lavaan (0.6-1) converged normally after  57 iterations
```

```
  Number of observations per group
  Pasteur                                              156
  Grant-White                                          145

  Estimator                                             ML
  Model Fit Test Statistic                         115.851
  Degrees of freedom                                    48
  P-value (Chi-square)                               0.000

Chi-square for each group:

  Pasteur                                           64.309
  Grant-White                                       51.542

Parameter Estimates:

  Information                                     Expected
  Information saturated (h1) model              Structured
  Standard Errors                                 Standard


Group 1 [Pasteur]:

Latent Variables:
                 Estimate  Std.Err  z-value  P(>|z|)
  visual =~
    x1              1.000
    x2              0.394    0.122    3.220    0.001
    x3              0.570    0.140    4.076    0.000
  textual =~
    x4              1.000
    x5              1.183    0.102   11.613    0.000
    x6              0.875    0.077   11.421    0.000
  speed =~
    x7              1.000
    x8              1.125    0.277    4.057    0.000
    x9              0.922    0.225    4.104    0.000

Covariances:
                 Estimate  Std.Err  z-value  P(>|z|)
  visual ~~
    textual         0.479    0.106    4.531    0.000
    speed           0.185    0.077    2.397    0.017
  textual ~~
    speed           0.182    0.069    2.628    0.009

Intercepts:
                 Estimate  Std.Err  z-value  P(>|z|)
   .x1             4.941    0.095   52.249    0.000
   .x2             5.984    0.098   60.949    0.000
   .x3             2.487    0.093   26.778    0.000
   .x4             2.823    0.092   30.689    0.000
   .x5             3.995    0.105   38.183    0.000
   .x6             1.922    0.079   24.321    0.000
   .x7             4.432    0.087   51.181    0.000
   .x8             5.563    0.078   71.214    0.000
   .x9             5.418    0.079   68.440    0.000
    visual         0.000
    textual        0.000
```

18

```
      speed             0.000

  Variances:
                     Estimate  Std.Err  z-value  P(>|z|)
     .x1               0.298    0.232    1.286    0.198
     .x2               1.334    0.158    8.464    0.000
     .x3               0.989    0.136    7.271    0.000
     .x4               0.425    0.069    6.138    0.000
     .x5               0.456    0.086    5.292    0.000
     .x6               0.290    0.050    5.780    0.000
     .x7               0.820    0.125    6.580    0.000
     .x8               0.510    0.116    4.406    0.000
     .x9               0.680    0.104    6.516    0.000
      visual           1.097    0.276    3.967    0.000
      textual          0.894    0.150    5.963    0.000
      speed            0.350    0.126    2.778    0.005


Group 2 [Grant-White]:

Latent Variables:
                     Estimate  Std.Err  z-value  P(>|z|)
   visual =~
     x1                1.000
     x2                0.736    0.155    4.760    0.000
     x3                0.925    0.166    5.583    0.000
   textual =~
     x4                1.000
     x5                0.990    0.087   11.418    0.000
     x6                0.963    0.085   11.377    0.000
   speed =~
     x7                1.000
     x8                1.226    0.187    6.569    0.000
     x9                1.058    0.165    6.429    0.000

Covariances:
                     Estimate  Std.Err  z-value  P(>|z|)
   visual ~~
     textual           0.408    0.098    4.153    0.000
     speed             0.276    0.076    3.639    0.000
   textual ~~
     speed             0.222    0.073    3.022    0.003

Intercepts:
                     Estimate  Std.Err  z-value  P(>|z|)
     .x1               4.930    0.095   51.696    0.000
     .x2               6.200    0.092   67.416    0.000
     .x3               1.996    0.086   23.195    0.000
     .x4               3.317    0.093   35.625    0.000
     .x5               4.712    0.096   48.986    0.000
     .x6               2.469    0.094   26.277    0.000
     .x7               3.921    0.086   45.819    0.000
     .x8               5.488    0.087   63.174    0.000
     .x9               5.327    0.085   62.571    0.000
      visual           0.000
      textual          0.000
      speed            0.000

Variances:
                     Estimate  Std.Err  z-value  P(>|z|)
```

```
       .x1                0.715    0.126    5.676    0.000
       .x2                0.899    0.123    7.339    0.000
       .x3                0.557    0.103    5.409    0.000
       .x4                0.315    0.065    4.870    0.000
       .x5                0.419    0.072    5.812    0.000
       .x6                0.406    0.069    5.880    0.000
       .x7                0.600    0.091    6.584    0.000
       .x8                0.401    0.094    4.249    0.000
       .x9                0.535    0.089    6.010    0.000
        visual            0.604    0.160    3.762    0.000
        textual           0.942    0.152    6.177    0.000
        speed             0.461    0.118    3.910    0.000
```

If you want to fix parameters, or provide starting values, you can use the same pre-multiplication techniques, but the single argument is now replaced by a *vector* of arguments, one for each group. If you use a single element instead of a vector, that element will be applied for all groups (note: this is NOT true for labels, since this would imply equality constraints). For example:

```
HS.model <- '  visual =~ x1 + 0.5*x2 + c(0.6, 0.8)*x3
              textual =~ x4 + start(c(1.2, 0.6))*x5 + a*x6
              speed   =~ x7 + x8 + x9 '
```

In the definition of the latent factor `visual`, we have fixed the factor loading of the indicator `x3` to the value '0.6' in the first group, and to the value '0.8' in the second group, while the factor loading of the indicator `x2` is fixed to the value '0.5' in both groups. In the definition of the `textual` factor, two different starting values are provided for the `x5` indicator; one for each group. In addition, we have labeled the factor loading of the `x6` indicator as 'a', but this label is only given to the parameter of the first group. If you want to provide labels to each of the two groups, you can write something like `c(a1,a2)*x6`. Be careful: if you write `c(a,a)*x6`, both parameters (in the first and second group) will get the same label, and hence they will be treated as a single parameter. To verify the effects of these modifiers, we refit the model:

```
fit <- cfa(HS.model,
           data = HolzingerSwineford1939,
           group = "school")
 summary(fit)
```

```
lavaan (0.6-1) converged normally after  48 iterations

  Number of observations per group
  Pasteur                                        156
  Grant-White                                    145

  Estimator                                       ML
  Model Fit Test Statistic                   118.976
  Degrees of freedom                              52
  P-value (Chi-square)                         0.000

Chi-square for each group:

  Pasteur                                     64.901
  Grant-White                                 54.075

Parameter Estimates:

  Information                               Expected
  Information saturated (h1) model        Structured
  Standard Errors                           Standard


Group 1 [Pasteur]:
```

```
Latent Variables:
                Estimate  Std.Err  z-value  P(>|z|)
  visual =~
    x1              1.000
    x2              0.500
    x3              0.600
  textual =~
    x4              1.000
    x5              1.185    0.102   11.598    0.000
    x6        (a)   0.876    0.077   11.409    0.000
  speed =~
    x7              1.000
    x8              1.129    0.279    4.055    0.000
    x9              0.931    0.227    4.103    0.000

Covariances:
                Estimate  Std.Err  z-value  P(>|z|)
  visual ~~
    textual         0.460    0.103    4.479    0.000
    speed           0.182    0.076    2.408    0.016
  textual ~~
    speed           0.181    0.069    2.625    0.009

Intercepts:
                Estimate  Std.Err  z-value  P(>|z|)
   .x1             4.941    0.094   52.379    0.000
   .x2             5.984    0.100   59.945    0.000
   .x3             2.487    0.092   26.983    0.000
   .x4             2.823    0.092   30.689    0.000
   .x5             3.995    0.105   38.183    0.000
   .x6             1.922    0.079   24.321    0.000
   .x7             4.432    0.087   51.181    0.000
   .x8             5.563    0.078   71.214    0.000
   .x9             5.418    0.079   68.440    0.000
    visual         0.000
    textual        0.000
    speed          0.000

Variances:
                Estimate  Std.Err  z-value  P(>|z|)
   .x1             0.388    0.129    3.005    0.003
   .x2             1.304    0.155    8.432    0.000
   .x3             0.965    0.120    8.016    0.000
   .x4             0.427    0.069    6.153    0.000
   .x5             0.454    0.086    5.270    0.000
   .x6             0.289    0.050    5.763    0.000
   .x7             0.824    0.124    6.617    0.000
   .x8             0.510    0.116    4.417    0.000
   .x9             0.677    0.105    6.479    0.000
    visual         1.001    0.172    5.803    0.000
    textual        0.892    0.150    5.953    0.000
    speed          0.346    0.125    2.768    0.006


Group 2 [Grant-White]:

Latent Variables:
                Estimate  Std.Err  z-value  P(>|z|)
  visual =~
```

```
   x1              1.000
   x2              0.500
   x3              0.800
 textual =~
   x4              1.000
   x5              0.990    0.087   11.425    0.000
   x6              0.963    0.085   11.374    0.000
 speed =~
   x7              1.000
   x8              1.228    0.188    6.539    0.000
   x9              1.081    0.168    6.417    0.000

Covariances:
                  Estimate  Std.Err  z-value  P(>|z|)
 visual ~~
   textual         0.454    0.099    4.585    0.000
   speed           0.315    0.079    4.004    0.000
 textual ~~
   speed           0.222    0.073    3.049    0.002

Intercepts:
                  Estimate  Std.Err  z-value  P(>|z|)
  .x1              4.930    0.097   50.688    0.000
  .x2              6.200    0.089   69.616    0.000
  .x3              1.996    0.086   23.223    0.000
  .x4              3.317    0.093   35.625    0.000
  .x5              4.712    0.096   48.986    0.000
  .x6              2.469    0.094   26.277    0.000
  .x7              3.921    0.086   45.819    0.000
  .x8              5.488    0.087   63.174    0.000
  .x9              5.327    0.085   62.571    0.000
   visual          0.000
   textual         0.000
   speed           0.000

Variances:
                  Estimate  Std.Err  z-value  P(>|z|)
  .x1              0.637    0.115    5.539    0.000
  .x2              0.966    0.120    8.076    0.000
  .x3              0.601    0.091    6.591    0.000
  .x4              0.316    0.065    4.877    0.000
  .x5              0.418    0.072    5.805    0.000
  .x6              0.407    0.069    5.887    0.000
  .x7              0.609    0.091    6.658    0.000
  .x8              0.411    0.094    4.385    0.000
  .x9              0.522    0.089    5.887    0.000
   visual          0.735    0.132    5.544    0.000
   textual         0.942    0.152    6.176    0.000
   speed           0.453    0.117    3.871    0.000
```

**Fixing parameters in some groups, but not all**  Sometimes, we wish to fix the value of a parameter in all groups, except for one particular group. In this group, we wish to freely estimate the value of that parameter. The modifier for this parameter is again a vector containing the fixed values for this parameter for each group, but we can use NA to force a parameter to be free in one (or more) group(s). Suppose for example we have four groups. We define a latent variable (say f) with three indicators. We wish to fix the factor loading of indicator item2 to 1.0 in all but the second group. We can write something like

```
f =~ item1 + c(1,NA,1,1)*item2 + item3
```

**Constraining a single parameter to be equal across groups**  If you want to constrain one or more parameters to be equal across groups, you need to give them the same label. For example, to constrain the factor loading of the indicator `x3` to be equal across (two) groups, you can write:

```
HS.model <- '  visual =~ x1 + x2 + c(v3,v3)*x3
             textual =~ x4 + x5 + x6
             speed   =~ x7 + x8 + x9 '
```

Again, identical labels imply identical parameters, both within and across groups.

**Constraining groups of parameters to be equal across groups**  Although providing identical labels is a very flexible method to specify equality constraints for a few parameters, there is a more convenient way to impose equality constraints on a whole set of parameters (for example: all factor loadings, or all intercepts). We call these type of constraints *group equality constraints* and they can be specified by the argument `group.equal` in the fitting function. For example, to constrain (all) the factor loadings to be equal across groups, you can proceed as follows:

```
HS.model <- '  visual =~ x1 + x2 + x3
             textual =~ x4 + x5 + x6
             speed   =~ x7 + x8 + x9 '
fit <- cfa(HS.model,
           data = HolzingerSwineford1939,
           group = "school",
           group.equal = c("loadings"))
summary(fit)
```

```
lavaan (0.6-1) converged normally after  42 iterations

  Number of observations per group
  Pasteur                                          156
  Grant-White                                      145

  Estimator                                         ML
  Model Fit Test Statistic                     124.044
  Degrees of freedom                                54
  P-value (Chi-square)                           0.000

Chi-square for each group:

  Pasteur                                       68.825
  Grant-White                                   55.219

Parameter Estimates:

  Information                                 Expected
  Information saturated (h1) model          Structured
  Standard Errors                             Standard


Group 1 [Pasteur]:

Latent Variables:
                   Estimate  Std.Err  z-value  P(>|z|)
  visual =~
    x1                1.000
    x2      (.p2.)    0.599    0.100    5.979    0.000
    x3      (.p3.)    0.784    0.108    7.267    0.000
  textual =~
    x4                1.000
    x5      (.p5.)    1.083    0.067   16.049    0.000
```

```
  x6        (.p6.)    0.912    0.058   15.785    0.000
 speed =~
  x7                  1.000
  x8        (.p8.)    1.201    0.155    7.738    0.000
  x9        (.p9.)    1.038    0.136    7.629    0.000


Covariances:
                    Estimate  Std.Err  z-value  P(>|z|)
 visual ~~
  textual             0.416    0.097    4.271    0.000
  speed               0.169    0.064    2.643    0.008
 textual ~~
  speed               0.176    0.061    2.882    0.004


Intercepts:
                    Estimate  Std.Err  z-value  P(>|z|)
 .x1                  4.941    0.093   52.991    0.000
 .x2                  5.984    0.100   60.096    0.000
 .x3                  2.487    0.094   26.465    0.000
 .x4                  2.823    0.093   30.371    0.000
 .x5                  3.995    0.101   39.714    0.000
 .x6                  1.922    0.081   23.711    0.000
 .x7                  4.432    0.086   51.540    0.000
 .x8                  5.563    0.078   71.087    0.000
 .x9                  5.418    0.079   68.153    0.000
  visual              0.000
  textual             0.000
  speed               0.000


Variances:
                    Estimate  Std.Err  z-value  P(>|z|)
 .x1                  0.551    0.137    4.010    0.000
 .x2                  1.258    0.155    8.117    0.000
 .x3                  0.882    0.128    6.884    0.000
 .x4                  0.434    0.070    6.238    0.000
 .x5                  0.508    0.082    6.229    0.000
 .x6                  0.266    0.050    5.294    0.000
 .x7                  0.849    0.114    7.468    0.000
 .x8                  0.515    0.095    5.409    0.000
 .x9                  0.658    0.096    6.865    0.000
  visual              0.805    0.171    4.714    0.000
  textual             0.913    0.137    6.651    0.000
  speed               0.305    0.078    3.920    0.000


Group 2 [Grant-White]:

Latent Variables:
                    Estimate  Std.Err  z-value  P(>|z|)
 visual =~
  x1                  1.000
  x2        (.p2.)    0.599    0.100    5.979    0.000
  x3        (.p3.)    0.784    0.108    7.267    0.000
 textual =~
  x4                  1.000
  x5        (.p5.)    1.083    0.067   16.049    0.000
  x6        (.p6.)    0.912    0.058   15.785    0.000
 speed =~
  x7                  1.000
  x8        (.p8.)    1.201    0.155    7.738    0.000
```

```
    x9      (.p9.)    1.038    0.136    7.629    0.000
```

Covariances:

|  | Estimate | Std.Err | z-value | P(>\|z\|) |
|---|---|---|---|---|
| visual ~~ |  |  |  |  |
|   textual | 0.437 | 0.099 | 4.423 | 0.000 |
|   speed | 0.314 | 0.079 | 3.958 | 0.000 |
| textual ~~ |  |  |  |  |
|   speed | 0.226 | 0.072 | 3.144 | 0.002 |

Intercepts:

|  | Estimate | Std.Err | z-value | P(>\|z\|) |
|---|---|---|---|---|
| .x1 | 4.930 | 0.097 | 50.763 | 0.000 |
| .x2 | 6.200 | 0.091 | 68.379 | 0.000 |
| .x3 | 1.996 | 0.085 | 23.455 | 0.000 |
| .x4 | 3.317 | 0.092 | 35.950 | 0.000 |
| .x5 | 4.712 | 0.100 | 47.173 | 0.000 |
| .x6 | 2.469 | 0.091 | 27.248 | 0.000 |
| .x7 | 3.921 | 0.086 | 45.555 | 0.000 |
| .x8 | 5.488 | 0.087 | 63.257 | 0.000 |
| .x9 | 5.327 | 0.085 | 62.786 | 0.000 |
| visual | 0.000 |  |  |  |
| textual | 0.000 |  |  |  |
| speed | 0.000 |  |  |  |

Variances:

|  | Estimate | Std.Err | z-value | P(>\|z\|) |
|---|---|---|---|---|
| .x1 | 0.645 | 0.127 | 5.084 | 0.000 |
| .x2 | 0.933 | 0.121 | 7.732 | 0.000 |
| .x3 | 0.605 | 0.096 | 6.282 | 0.000 |
| .x4 | 0.329 | 0.062 | 5.279 | 0.000 |
| .x5 | 0.384 | 0.073 | 5.270 | 0.000 |
| .x6 | 0.437 | 0.067 | 6.576 | 0.000 |
| .x7 | 0.599 | 0.090 | 6.651 | 0.000 |
| .x8 | 0.406 | 0.089 | 4.541 | 0.000 |
| .x9 | 0.532 | 0.086 | 6.202 | 0.000 |
| visual | 0.722 | 0.161 | 4.490 | 0.000 |
| textual | 0.906 | 0.136 | 6.646 | 0.000 |
| speed | 0.475 | 0.109 | 4.347 | 0.000 |

More 'group equality constraints' can be added. In addition to the factor loadings, the following keywords are supported in the `group.equal` argument:

- `intercepts`: the intercepts of the observed variables
- `means`: the intercepts/means of the latent variables
- `residuals`: the residual variances of the observed variables
- `residual.covariances`: the residual covariances of the observed variables
- `lv.variances`: the (residual) variances of the latent variables
- `lv.covariances`: the (residual) covariances of the latent varibles
- `regressions`: all regression coefficients in the model

If you omit the `group.equal` argument, all parameters are freely estimated in each group (but the model structure is the same).

But what if you want to constrain a whole group of parameters (say all factor loadings and intercepts) across groups, except for one or two parameters that need to stay free in all groups. For this scenario, you can use the argument `group.partial`, containing the names of those parameters that need to remain free. For example:

```
fit <- cfa(HS.model,
           data = HolzingerSwineford1939,
           group = "school",
```

```
                group.equal = c("loadings", "intercepts"),
                group.partial = c("visual=~x2", "x7~1"))
```

**Measurement Invariance**  If you are interested in testing the measurement invariance of a CFA model
across several groups, you can use the function `measurementInvariance()` which performs a number of multi-
ple group analyses in a particular sequence, with increasingly more restrictions on the parameters. (Note: from
the 0.5 series onwards, the `measurementInvariance()` function has been moved to the `semTools` package.)
Each model is compared to the baseline model and the previous model using chi-square difference tests. In
addition, the difference in the fit measure is also shown. Although the current implementation of the func-
tion is still a bit primitive, it does illustrate how the various components of the lavaan package can be used
as building blocks for constructing higher level functions (such as the `measurementInvariance()` function),
something that is often very hard to accomplish with commercial software.

```
library(semTools)
measurementInvariance(HS.model,
                      data = HolzingerSwineford1939,
                      group = "school")
```

```
Measurement invariance models:

Model 1 : fit.configural
Model 2 : fit.loadings
Model 3 : fit.intercepts
Model 4 : fit.means

Chi Square Difference Test

               Df    AIC     BIC  Chisq Chisq diff Df diff Pr(>Chisq)
fit.configural 48 7484.4 7706.8 115.85
fit.loadings   54 7480.6 7680.8 124.04      8.192       6     0.2244
fit.intercepts 60 7508.6 7686.6 164.10     40.059       6  4.435e-07 ***
fit.means      63 7543.1 7710.0 204.61     40.502       3  8.338e-09 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1


Fit measures:

                 cfi rmsea cfi.delta rmsea.delta
fit.configural 0.923 0.097        NA          NA
fit.loadings   0.921 0.093     0.002       0.004
fit.intercepts 0.882 0.107     0.038       0.015
fit.means      0.840 0.122     0.042       0.015
```

By adding the `group.partial` argument, you can test for partial measurement invariance by allowing a few
parameters to remain free.

# 9   Growth curve models

Another important type of latent variable models are latent growth curve models. Growth modeling is often
used to analyze longitudinal or developmental data. In this type of data, an outcome measure is measured on
several occasions, and we want to study the change over time. In many cases, the trajectory over time can be
modeled as a simple linear or quadratic curve. Random effects are used to capture individual differences. The
random effects are conveniently represented by (continuous) latent variables, often called *growth factors*. In
the example below, we use an artifical dataset called `Demo.growth` where a score (say, a standardized score on
a reading ability scale) is measured on 4 time points. To fit a linear growth model for these four time points,
we need to specify a model with two latent variables: a random intercept, and a random slope:

```
# linear growth model with 4 timepoints
# intercept and slope with fixed coefficients
 i =~ 1*t1 + 1*t2 + 1*t3 + 1*t4
 s =~ 0*t1 + 1*t2 + 2*t3 + 3*t4
```

In this model, we have fixed all the coefficients of the growth functions. To fit this model, the lavaan package provides a special `growth()` function:

```
model <- ' i =~ 1*t1 + 1*t2 + 1*t3 + 1*t4
           s =~ 0*t1 + 1*t2 + 2*t3 + 3*t4 '
fit <- growth(model, data=Demo.growth)
summary(fit)
```

```
lavaan (0.6-1) converged normally after  29 iterations

  Number of observations                           400

  Estimator                                         ML
  Model Fit Test Statistic                       8.069
  Degrees of freedom                                 5
  P-value (Chi-square)                           0.152

Parameter Estimates:

  Information                                 Expected
  Information saturated (h1) model          Structured
  Standard Errors                             Standard

Latent Variables:
                   Estimate  Std.Err  z-value  P(>|z|)
  i =~
    t1                1.000
    t2                1.000
    t3                1.000
    t4                1.000
  s =~
    t1                0.000
    t2                1.000
    t3                2.000
    t4                3.000

Covariances:
                   Estimate  Std.Err  z-value  P(>|z|)
  i ~~
    s                 0.618    0.071    8.686    0.000

Intercepts:
                   Estimate  Std.Err  z-value  P(>|z|)
   .t1                0.000
   .t2                0.000
   .t3                0.000
   .t4                0.000
    i                 0.615    0.077    8.007    0.000
    s                 1.006    0.042   24.076    0.000

Variances:
                   Estimate  Std.Err  z-value  P(>|z|)
   .t1                0.595    0.086    6.944    0.000
   .t2                0.676    0.061   11.061    0.000
   .t3                0.635    0.072    8.761    0.000
```

```
     .t4              0.508    0.124    4.090    0.000
      i               1.932    0.173   11.194    0.000
      s               0.587    0.052   11.336    0.000
```

Technically, the `growth()` function is almost identical to the `sem()` function. But a mean structure is automatically assumed, and the observed intercepts are fixed to zero by default, while the latent variable intercepts/means are freely estimated. A slightly more complex model adds two regressors (`x1` and `x2`) that influence the latent growth factors. In addition, a time-varying covariate `c` that influences the outcome measure at the four time points has been added to the model. A graphical representation of this model is presented below.
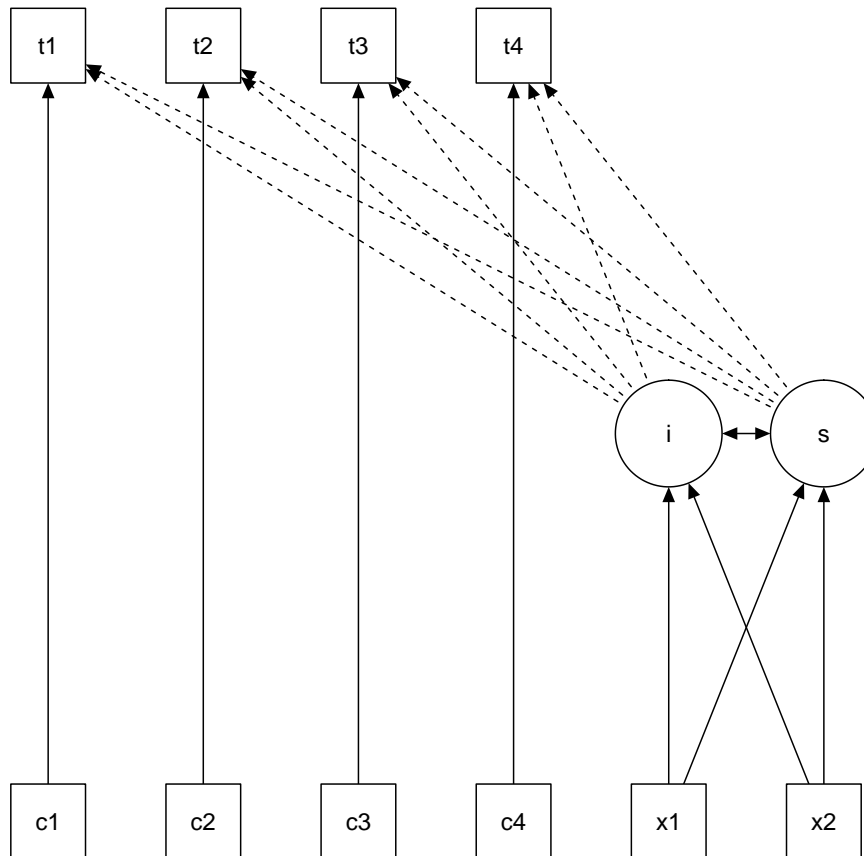


The corresponding syntax is the following:

```
# intercept and slope
# with fixed coefficients
  i =~ 1*t1 + 1*t2 + 1*t3 + 1*t4
  s =~ 0*t1 + 1*t2 + 2*t3 + 3*t4
# regressions
  i ~ x1 + x2
  s ~ x1 + x2
# time-varying covariates
  t1 ~ c1
  t2 ~ c2
  t3 ~ c3
  t4 ~ c4
```

For ease of copy/pasting, the complete R code needed to specify and fit this linear growth model with a time-varying covariate is printed again below:

```
# a linear growth model with a time-varying covariate
model <- '
  # intercept and slope with fixed coefficients
    i =~ 1*t1 + 1*t2 + 1*t3 + 1*t4
    s =~ 0*t1 + 1*t2 + 2*t3 + 3*t4
  # regressions
    i ~ x1 + x2
    s ~ x1 + x2
  # time-varying covariates
    t1 ~ c1
    t2 ~ c2
    t3 ~ c3
    t4 ~ c4
'
fit <- growth(model, data = Demo.growth)
summary(fit)
```

# 10   Using categorical variables

Binary, ordinal and nominal variables are considered categorical (not continuous). It makes a big difference if these categorical variables are exogenous (independent) or endogenous (dependent) in the model.

**Exogenous categorical variables**   If you have a binary exogenous covariate (say, gender), all you need to do is to recode it as a dummy (0/1) variable. Just like you would do in a classic regression model. If you have an exogenous ordinal variable, you can use a coding scheme reflecting the order (say, 1,2,3,...) and treat it as any other (numeric) covariate. If you have a nominal categorical variable with $K > 2$ levels, you need to replace it by a set of $K - 1$ dummy variables, again, just like you would do in classical regression.

**Endogenous categorical variables**   The lavaan 0.5 series can deal with binary and ordinal (but not nominal) endogenous variables. Only the three-stage WLS approach is currently supported, including some 'robust' variants. To use binary/ordinal data, you have two choices:

1. declare them as 'ordered' (using the `ordered` function, which is part of base R) in your data.frame before you run the analysis; for example, if you need to declare four variables (say, `item1`, `item2`, `item3`, `item4`) as ordinal in your data.frame (called `Data`), you can use something like:

   ```
   Data[,c("item1",
           "item2",
           "item3",
           "item4")] <-
       lapply(Data[,c("item1",
                      "item2",
                      "item3",
                      "item4")], ordered)
   ```

2. use the `ordered` argument when using one of the fitting functions (cfa/sem/growth/lavaan), for example, if you have four binary or ordinal variables (say, `item1`, `item2`, `item3`, `item4`), you can use:

   ```
   fit <- cfa(myModel, data = myData,
              ordered=c("item1","item2",
                        "item3","item4"))
   ```

In both cases, lavaan will automatically switch to the `WLSMV` estimator: it will use diagonally weighted least squares (`DWLS`) to estimate the model parameters, but it will use the full weight matrix to compute robust standard errors, and a mean- and variance-adjusted test stastistic.

# 11 Using a covariance matrix as input

If you have no full dataset, but you do have a sample covariance matrix, you can still fit your model. If you wish to add a mean structure, you need to provide a mean vector too. Importantly, if only sample statistics are provided, you must specify the number of observations that were used to compute the sample moments. The following example illustrates the use of a sample covariance matrix as input. First, we read in the lower half of the covariance matrix (including the diagonal):

```
lower <- '
 11.834
  6.947   9.364
  6.819   5.091  12.532
  4.783   5.028   7.495   9.986
 -3.839  -3.889  -3.841  -3.625   9.610
-21.899 -18.831 -21.748 -18.775  35.522 450.288 '

wheaton.cov <-
    getCov(lower, names = c("anomia67", "powerless67",
                            "anomia71", "powerless71",
                            "education", "sei"))
```

The getCov() function makes it easy to create a full covariance matrix (including variable names) if you only have the lower-half elements (perhaps pasted from a textbook or a paper). Note that the lower-half elements are written between two single quotes. Therefore, you have some additional flexibility. You can add comments, and blank lines. If the numbers are separated by a comma, or a semi-colon, that is fine too. For more information about getCov(), see the online manual page.

Next, we can specify our model, estimate it, and request a summary of the results:

```
# classic wheaton et al model
wheaton.model <- '
  # latent variables
    ses     =~ education + sei
    alien67 =~ anomia67 + powerless67
    alien71 =~ anomia71 + powerless71
  # regressions
    alien71 ~ alien67 + ses
    alien67 ~ ses
  # correlated residuals
    anomia67 ~~ anomia71
    powerless67 ~~ powerless71
'
fit <- sem(wheaton.model,
           sample.cov = wheaton.cov,
           sample.nobs = 932)
summary(fit, standardized = TRUE)


lavaan (0.6-1) converged normally after  73 iterations

  Number of observations                           932

  Estimator                                         ML
  Model Fit Test Statistic                       4.735
  Degrees of freedom                                 4
  P-value (Chi-square)                           0.316

Parameter Estimates:

  Information                                 Expected
  Information saturated (h1) model          Structured
  Standard Errors                             Standard
```

```
Latent Variables:
                 Estimate  Std.Err  z-value  P(>|z|)   Std.lv  Std.all
  ses =~
    education       1.000                               2.607    0.842
    sei             5.219    0.422   12.364    0.000    13.609    0.642
  alien67 =~
    anomia67        1.000                               2.663    0.774
    powerless67     0.979    0.062   15.895    0.000     2.606    0.852
  alien71 =~
    anomia71        1.000                               2.850    0.805
    powerless71     0.922    0.059   15.498    0.000     2.628    0.832

Regressions:
                 Estimate  Std.Err  z-value  P(>|z|)   Std.lv  Std.all
  alien71 ~
    alien67         0.607    0.051   11.898    0.000     0.567    0.567
    ses            -0.227    0.052   -4.334    0.000    -0.207   -0.207
  alien67 ~
    ses            -0.575    0.056  -10.195    0.000    -0.563   -0.563

Covariances:
                 Estimate  Std.Err  z-value  P(>|z|)   Std.lv  Std.all
 .anomia67 ~~
   .anomia71        1.623    0.314    5.176    0.000     1.623    0.356
 .powerless67 ~~
   .powerless71     0.339    0.261    1.298    0.194     0.339    0.121

Variances:
                 Estimate  Std.Err  z-value  P(>|z|)   Std.lv  Std.all
   .education       2.801    0.507    5.525    0.000     2.801    0.292
   .sei           264.597   18.126   14.597    0.000   264.597    0.588
   .anomia67        4.731    0.453   10.441    0.000     4.731    0.400
   .powerless67     2.563    0.403    6.359    0.000     2.563    0.274
   .anomia71        4.399    0.515    8.542    0.000     4.399    0.351
   .powerless71     3.070    0.434    7.070    0.000     3.070    0.308
    ses             6.798    0.649   10.475    0.000     1.000    1.000
   .alien67         4.841    0.467   10.359    0.000     0.683    0.683
   .alien71         4.083    0.404   10.104    0.000     0.503    0.503
```

If you have multiple groups, the `sample.cov` argument must be a list containing the sample variance-covariance matrix of each group as a separate element in the list. If a mean structure is needed, the `sample.mean` argument must be a list containing the sample means of each group. Finally, the `sample.nobs` argument can be either a list or an integer vector containing the number of observations for each group.

# 12  Estimators, standard errors and missing values

**Estimators**  If all data is continuous, the default estimator in the lavaan package is maximum likelihood (`estimator = "ML"`). Alternative estimators available in lavaan are:

- `"GLS"`: generalized least squares. For complete data only.
- `"WLS"`: weighted least squares (sometimes called ADF estimation). For complete data only.
- `"DWLS"`: diagonally weighted least squares
- `"ULS"`: unweighted least squares

Many estimators have 'robust' variants, meaning that they provide robust standard errors and a scaled test statistic. For example, for the maximum likelihood estimator, lavaan provides the following robust variants:

- `"MLM"`: maximum likelihood estimation with robust standard errors and a Satorra-Bentler scaled test statistic. For complete data only.

- `"MLMVS"`: maximum likelihood estimation with robust standard errors and a mean- and variance adjusted test statistic (aka the Satterthwaite approach). For complete data only.
- `"MLMV"`: maximum likelihood estimation with robust standard errors and a mean- and variance adjusted test statistic (using a scale-shifted approach). For complete data only.
- `"MLF"`: for maximum likelihood estimation with standard errors based on the first-order derivatives, and a conventional test statistic. For both complete and incomplete data.
- `"MLR"`: maximum likelihood estimation with robust (Huber-White) standard errors and a scaled test statistic that is (asymptotically) equal to the Yuan-Bentler test statistic. For both complete and incomplete data.

For the `DWLS` and `ULS` estimators, lavaan also provides 'robust' variants: `WLSM`, `WLSMVS`, `WLSMV`, `ULSM`, `ULSMVS`, `ULSMV`. Note that for the robust `WLS` variants, we use the diagonal of the weight matrix for estimation, but we use the full weight matrix to correct the standard errors and to compute the test statistic.

**ML estimation: Wishart versus Normal** If maximum likelihood estimation is used (`"ML"` or any of its robusts variants), the default behavior of lavaan is to base the analysis on the so-called *biased* sample covariance matrix, where the elements are divided by $n$ instead of $n-1$. This is done internally, and should not be done by the user. In addition, the chi-square statistic is computed by multiplying the minimum function value with a factor $n$ (instead of $n-1$). This is similar to the Mplus program. If you prefer to use an unbiased covariance, and $n-1$ as the multiplier to compute the chi-square statistic, you need to specify the `likelihood = "wishart"` argument when calling the fitting functions. For example:

```
fit <- cfa(HS.model,
           data = HolzingerSwineford1939,
           likelihood = "wishart")
fit
```

```
lavaan (0.6-1) converged normally after  35 iterations

  Number of observations                           301

  Estimator                                         ML
  Model Fit Test Statistic                      85.022
  Degrees of freedom                                24
  P-value (Chi-square)                           0.000
```

The value of the test statistic will be closer to the value reported by programs like EQS, LISREL or AMOS, since they all use the 'Wishart' approach when using the maximum likelihood estimator. The program Mplus, on the other hand, uses the 'normal' approach to maximum likelihood estimation.

**Missing values** If the data contain missing values, the default behavior is listwise deletion. If the missing mechanism is MCAR (missing completely at random) or MAR (missing at random), the lavaan package provides case-wise (or 'full information') maximum likelihood estimation. You can turn this feature on, by using the argument `missing = "ML"` when calling the fitting function. An unrestricted (h1) model will automatically be estimated, so that all common fit indices are available.

**Standard errors** Standard errors are (by default) based on the expected information matrix. The only exception is when data are missing and full information ML is used (via `missing = "ML"`). In this case, the observed information matrix is used to compute the standard errors. The user can change this behavior by using the `information` argument, which can be set to `"expected"` or `"observed"`.
If the estimator is simply `"ML"`, you can request robust standard errors by using the `se` argument, which can be set to `"robust.sem"`, `"robust.huber.white"` , `"first.order"` or `"bootstrap"`.
Or simply to `"none"` if you don't need them. This will not affect the test statistic. In fact, you can choose the test statistic independently by using the `test` argument, which can be set to `"standard"`, `"Satorra-Bentler"`, `"Yuan-Bentler"` or `"bootstrap"`.

**Bootstrapping** There are two ways for using the bootstrap in lavaan. Either you can set `se = "bootstrap"` or `test = "bootstrap"` when fitting the model (and you will get bootstrap standard errors, and/or a bootstrap based p-value respectively), or you can you the `bootstrapLavaan()` function, which needs an already fitted lavaan object. The latter function can be used to 'bootstrap' any statistic (or vector of statistics) that you can extract from a fitted lavaan object.

# 13 Indirect effects and mediation analysis

Consider a classical mediation setup with three variables: Y is the dependent variable, X is the predictor, and M is a mediator. For illustration, we create a toy dataset containing these three variables, and fit a path analysis model that includes the direct effect of X on Y and the indirect effect of X on Y via M.

```r
set.seed(1234)
X <- rnorm(100)
M <- 0.5*X + rnorm(100)
Y <- 0.7*M + rnorm(100)
Data <- data.frame(X = X, Y = Y, M = M)
model <- ' # direct effect
            Y ~ c*X
          # mediator
            M ~ a*X
            Y ~ b*M
          # indirect effect (a*b)
            ab := a*b
          # total effect
            total := c + (a*b)
        '
fit <- sem(model, data = Data)
summary(fit)
```

```
lavaan (0.6-1) converged normally after  12 iterations

  Number of observations                              100

  Estimator                                            ML
  Model Fit Test Statistic                          0.000
  Degrees of freedom                                    0
  Minimum Function Value               0.0000000000000

Parameter Estimates:

  Information                                    Expected
  Information saturated (h1) model             Structured
  Standard Errors                                Standard

Regressions:
                   Estimate  Std.Err  z-value  P(>|z|)
  Y ~
    X          (c)    0.036    0.104    0.348    0.728
  M ~
    X          (a)    0.474    0.103    4.613    0.000
  Y ~
    M          (b)    0.788    0.092    8.539    0.000

Variances:
                   Estimate  Std.Err  z-value  P(>|z|)
   .Y                 0.898    0.127    7.071    0.000
   .M                 1.054    0.149    7.071    0.000

Defined Parameters:
                   Estimate  Std.Err  z-value  P(>|z|)
    ab                0.374    0.092    4.059    0.000
    total             0.410    0.125    3.287    0.001
```

The example illustrates the use of the `":="` operator in the lavaan model syntax. This operator 'defines' new parameters which take on values that are an arbitrary function of the original model parameters. The function,

however, must be specified in terms of the parameter *labels* that are explicitly mentioned in the model syntax. By default, the standard errors for these defined parameters are computed by using the so-called Delta method. As with other models, bootstrap standard errors can be requested simply by specifying `se = "bootstrap"` in the fitting function.

# 14  Modification Indices

Modification indices can be requested by adding the argument `modindices = TRUE` in the `summary()` call, or by calling the function `modindices()` directly. The `modindices()` function returns a data frame, which you can sort or filter to extract what you want. For example, to see only the modification indices for the factor loadings, you can use something like this:

```
fit <- cfa(HS.model,
           data = HolzingerSwineford1939)
mi <- modindices(fit)
mi[mi$op == "=~",]
```

```
        lhs op rhs     mi    epc sepc.lv sepc.all sepc.nox
25   visual =~  x4  1.211  0.077   0.069    0.059    0.059
26   visual =~  x5  7.441 -0.210  -0.189   -0.147   -0.147
27   visual =~  x6  2.843  0.111   0.100    0.092    0.092
28   visual =~  x7 18.631 -0.422  -0.380   -0.349   -0.349
29   visual =~  x8  4.295 -0.210  -0.189   -0.187   -0.187
30   visual =~  x9 36.411  0.577   0.519    0.515    0.515
31  textual =~  x1  8.903  0.350   0.347    0.297    0.297
32  textual =~  x2  0.017 -0.011  -0.011   -0.010   -0.010
33  textual =~  x3  9.151 -0.272  -0.269   -0.238   -0.238
34  textual =~  x7  0.098 -0.021  -0.021   -0.019   -0.019
35  textual =~  x8  3.359 -0.121  -0.120   -0.118   -0.118
36  textual =~  x9  4.796  0.138   0.137    0.136    0.136
37    speed =~  x1  0.014  0.024   0.015    0.013    0.013
38    speed =~  x2  1.580 -0.198  -0.123   -0.105   -0.105
39    speed =~  x3  0.716  0.136   0.084    0.075    0.075
40    speed =~  x4  0.003 -0.005  -0.003   -0.003   -0.003
41    speed =~  x5  0.201 -0.044  -0.027   -0.021   -0.021
42    speed =~  x6  0.273  0.044   0.027    0.025    0.025
```

Modification indices are printed out for each nonfree (or nonredundant) parameter. The modification indices are supplemented by the expected parameter change (EPC) values (column `epc`). The last three columns contain the standardized EPC values (`sepc.lv`: only standardizing the latent variables; `sepc.all`: standardizing all variables; `sepc.nox`: standardizing all but exogenous observed variables)

# 15  Extracting information from a fitted model

The `summary()` function gives a nice overview of a fitted model, but is for display only. If you need the actual numbers for further processing, you may prefer to use one of several 'extractor' functions. We have already seen the `coef()` function which extracts the estimated parameters of a fitted model. Other extractor functions are discussed below.

**parameterEstimates**   The `parameterEstimates` function extracts not only the values of the estimated parameters, but also the standard errors, the z-values, the standardized parameter values, and returns everything conveniently as a data frame. For example:

```
fit <- cfa(HS.model, data=HolzingerSwineford1939)
parameterEstimates(fit)
```

```
       lhs op   rhs   est    se     z pvalue ci.lower ci.upper
1   visual =~   x1 1.000 0.000    NA     NA    1.000    1.000
```

```
2    visual =~    x2 0.554 0.100  5.554    0   0.358  0.749
3    visual =~    x3 0.729 0.109  6.685    0   0.516  0.943
4   textual =~    x4 1.000 0.000     NA   NA   1.000  1.000
5   textual =~    x5 1.113 0.065 17.014    0   0.985  1.241
6   textual =~    x6 0.926 0.055 16.703    0   0.817  1.035
7     speed =~    x7 1.000 0.000     NA   NA   1.000  1.000
8     speed =~    x8 1.180 0.165  7.152    0   0.857  1.503
9     speed =~    x9 1.082 0.151  7.155    0   0.785  1.378
10       x1 ~~    x1 0.549 0.114  4.833    0   0.326  0.772
11       x2 ~~    x2 1.134 0.102 11.146    0   0.934  1.333
12       x3 ~~    x3 0.844 0.091  9.317    0   0.667  1.022
13       x4 ~~    x4 0.371 0.048  7.779    0   0.278  0.465
14       x5 ~~    x5 0.446 0.058  7.642    0   0.332  0.561
15       x6 ~~    x6 0.356 0.043  8.277    0   0.272  0.441
16       x7 ~~    x7 0.799 0.081  9.823    0   0.640  0.959
17       x8 ~~    x8 0.488 0.074  6.573    0   0.342  0.633
18       x9 ~~    x9 0.566 0.071  8.003    0   0.427  0.705
19   visual ~~ visual 0.809 0.145  5.564    0   0.524  1.094
20  textual ~~ textual 0.979 0.112  8.737    0   0.760  1.199
21    speed ~~  speed 0.384 0.086  4.451    0   0.215  0.553
22   visual ~~ textual 0.408 0.074  5.552    0   0.264  0.552
23   visual ~~  speed 0.262 0.056  4.660    0   0.152  0.373
24  textual ~~  speed 0.173 0.049  3.518    0   0.077  0.270
```

**standardizedSolution**   The `standardizedSolution()` function is similar to the `parameterEstimates()` function, but only shows the unstandardized and standardized parameter estimates.

**fitted.values**   The `fitted()` and `fitted.values()` functions return the model-implied (fitted) covariance matrix (and mean vector) of a fitted model:

```
fit <- cfa(HS.model, data=HolzingerSwineford1939)
fitted(fit)
```

```
$cov
    x1    x2    x3    x4    x5    x6    x7    x8    x9
x1 1.358
x2 0.448 1.382
x3 0.590 0.327 1.275
x4 0.408 0.226 0.298 1.351
x5 0.454 0.252 0.331 1.090 1.660
x6 0.378 0.209 0.276 0.907 1.010 1.196
x7 0.262 0.145 0.191 0.173 0.193 0.161 1.183
x8 0.309 0.171 0.226 0.205 0.228 0.190 0.453 1.022
x9 0.284 0.157 0.207 0.188 0.209 0.174 0.415 0.490 1.015
```

```
$mean
x1 x2 x3 x4 x5 x6 x7 x8 x9
 0  0  0  0  0  0  0  0  0
```

**residuals**   The `resid()` or `residuals()` functions return (unstandardized) residuals of a fitted model. This is simply the difference between the observed and implied covariance matrix and mean vector. If the estimator is maximum likelihood, it is also possible to obtain the normalized and the standardized residuals (note: you may observe several `NA` values, but they can be safely ignored)

```
fit <- cfa(HS.model, data=HolzingerSwineford1939)
resid(fit, type="standardized")
```

```
$type
[1] "standardized"
```

```
$cov
      x1     x2     x3     x4     x5     x6     x7     x8     x9
x1     NA
x2 -2.196  0.000
x3 -1.199  2.692  0.000
x4  2.465 -0.283 -1.948  0.000
x5 -0.362 -0.610 -4.443  0.856  0.000
x6  2.032  0.661 -0.701     NA  0.633  0.000
x7 -3.787 -3.800 -1.882  0.839 -0.837 -0.321  0.000
x8 -1.456 -1.137 -0.305 -2.049 -1.100 -0.635  3.804  0.000
x9  4.062  1.517  3.328  1.237  1.723  1.436 -2.771     NA  0.000


$mean
x1 x2 x3 x4 x5 x6 x7 x8 x9
 0  0  0  0  0  0  0  0  0
```

**vcov**   The function `vcov()` returns the estimated covariance matrix of the parameter estimates.

**AIC and BIC**   The `AIC()` and `BIC()` functions return the AIC and BIC values of a fitted model.

**fitMeasures**   The `fitMeasures()` function returns all the fit measures computed by lavaan as a named numeric vector.

```
fit <- cfa(HS.model, data=HolzingerSwineford1939)
fitMeasures(fit)
```

```
              npar                   fmin                  chisq
            21.000                  0.142                 85.306
                df                 pvalue         baseline.chisq
            24.000                  0.000                918.852
       baseline.df         baseline.pvalue                    cfi
            36.000                  0.000                  0.931
               tli                   nnfi                    rfi
             0.896                  0.896                  0.861
               nfi                   pnfi                    ifi
             0.907                  0.605                  0.931
               rni                   logl       unrestricted.logl
             0.931              -3737.745              -3695.092
               aic                    bic                 ntotal
          7517.490               7595.339                301.000
              bic2                  rmsea         rmsea.ci.lower
          7528.739                  0.092                  0.071
    rmsea.ci.upper           rmsea.pvalue                    rmr
             0.114                  0.001                  0.082
        rmr_nomean                   srmr          srmr_bentler
             0.082                  0.065                  0.065
 srmr_bentler_nomean            srmr_bollen     srmr_bollen_nomean
             0.065                  0.065                  0.065
        srmr_mplus      srmr_mplus_nomean                  cn_05
             0.065                  0.065                129.490
             cn_01                    gfi                   agfi
           152.654                  0.943                  0.894
              pgfi                    mfi                   ecvi
             0.503                  0.903                  0.423
```

If you only want the value of a single fit measure, say, the CFI, you give the name (in lower case) as the second argument:

```
fit <- cfa(HS.model, data=HolzingerSwineford1939)
fitMeasures(fit, "cfi")
```

```
  cfi
0.931
```

Or you can provide a vector of fit measures, as in

```
fitMeasures(fit, c("cfi","rmsea","srmr"))
```

```
  cfi rmsea  srmr
0.931 0.092 0.065
```

**inspect** If you want to peek inside a fitted lavaan object (the object that is returned by a call to `cfa()`, `sem()`or `growth()`), you can use the `inspect()` function, with a variety of options. By default, calling `inspect()` on a fitted lavaan object returns a list of the model matrices that are used internally to represent the model. The free parameters are nonzero integers.

```
fit <- cfa(HS.model, data=HolzingerSwineford1939)
inspect(fit)
```

```
$lambda
   visual textul speed
x1      0      0     0
x2      1      0     0
x3      2      0     0
x4      0      0     0
x5      0      3     0
x6      0      4     0
x7      0      0     0
x8      0      0     5
x9      0      0     6


$theta
   x1 x2 x3 x4 x5 x6 x7 x8 x9
x1  7
x2  0  8
x3  0  0  9
x4  0  0  0 10
x5  0  0  0  0 11
x6  0  0  0  0  0 12
x7  0  0  0  0  0  0 13
x8  0  0  0  0  0  0  0 14
x9  0  0  0  0  0  0  0  0 15


$psi
        visual textul speed
visual  16
textual 19     17
speed   20     21     18
```

To see the starting values of parameters in each model matrix, type

```
inspect(fit, what="start")
```

```
$lambda
   visual textul speed
x1  1.000  0.000 0.000
x2  0.778  0.000 0.000
```

```
x3  1.107  0.000 0.000
x4  0.000  1.000 0.000
x5  0.000  1.133 0.000
x6  0.000  0.924 0.000
x7  0.000  0.000 1.000
x8  0.000  0.000 1.225
x9  0.000  0.000 0.854


$theta
    x1    x2    x3    x4    x5    x6    x7    x8    x9
x1 0.679
x2 0.000 0.691
x3 0.000 0.000 0.637
x4 0.000 0.000 0.000 0.675
x5 0.000 0.000 0.000 0.000 0.830
x6 0.000 0.000 0.000 0.000 0.000 0.598
x7 0.000 0.000 0.000 0.000 0.000 0.000 0.592
x8 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.511
x9 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.508


$psi
        visual textul speed
visual   0.05
textual 0.00    0.05
speed   0.00    0.00    0.05
```

To see how lavaan internally represents a model, you can type

```
inspect(fit, what="list")
```

```
   id      lhs op      rhs user block group free ustart exo label plabel
1   1   visual =~      x1    1     1     1    0      1   0           .p1.
2   2   visual =~      x2    1     1     1    1     NA   0           .p2.
3   3   visual =~      x3    1     1     1    2     NA   0           .p3.
4   4  textual =~      x4    1     1     1    0      1   0           .p4.
5   5  textual =~      x5    1     1     1    3     NA   0           .p5.
6   6  textual =~      x6    1     1     1    4     NA   0           .p6.
7   7    speed =~      x7    1     1     1    0      1   0           .p7.
8   8    speed =~      x8    1     1     1    5     NA   0           .p8.
9   9    speed =~      x9    1     1     1    6     NA   0           .p9.
10 10       x1 ~~      x1    0     1     1    7     NA   0          .p10.
11 11       x2 ~~      x2    0     1     1    8     NA   0          .p11.
12 12       x3 ~~      x3    0     1     1    9     NA   0          .p12.
13 13       x4 ~~      x4    0     1     1   10     NA   0          .p13.
14 14       x5 ~~      x5    0     1     1   11     NA   0          .p14.
15 15       x6 ~~      x6    0     1     1   12     NA   0          .p15.
16 16       x7 ~~      x7    0     1     1   13     NA   0          .p16.
17 17       x8 ~~      x8    0     1     1   14     NA   0          .p17.
18 18       x9 ~~      x9    0     1     1   15     NA   0          .p18.
19 19   visual ~~  visual    0     1     1   16     NA   0          .p19.
20 20  textual ~~ textual    0     1     1   17     NA   0          .p20.
21 21    speed ~~   speed    0     1     1   18     NA   0          .p21.
22 22   visual ~~ textual    0     1     1   19     NA   0          .p22.
23 23   visual ~~   speed    0     1     1   20     NA   0          .p23.
24 24  textual ~~   speed    0     1     1   21     NA   0          .p24.
    start   est   se
1   1.000 1.000 0.000
2   0.778 0.554 0.100
3   1.107 0.729 0.109
4   1.000 1.000 0.000
```

```
5  1.133 1.113 0.065
6  0.924 0.926 0.055
7  1.000 1.000 0.000
8  1.225 1.180 0.165
9  0.854 1.082 0.151
10 0.679 0.549 0.114
11 0.691 1.134 0.102
12 0.637 0.844 0.091
13 0.675 0.371 0.048
14 0.830 0.446 0.058
15 0.598 0.356 0.043
16 0.592 0.799 0.081
17 0.511 0.488 0.074
18 0.508 0.566 0.071
19 0.050 0.809 0.145
20 0.050 0.979 0.112
21 0.050 0.384 0.086
22 0.000 0.408 0.074
23 0.000 0.262 0.056
24 0.000 0.173 0.049
```

This is equivalent to the `parTable(fit)` function. The table that is returned here is called the 'parameter table'.

For more inspect options, see the help page for the lavaan class which you can find by typing the following:

```
class?lavaan
```

# 16   Multilevel SEM

If the data is clustered, one way to handle the clustering is to use a multilevel modeling approach. In the SEM framework, this leads to multilevel SEM. The multilevel capabilities of lavaan are still limited, but you can fit a two-level SEM with random intercepts (note: only when all data is continuous and complete; listwise deletion is currently used for cases with missing values).

**Multilevel SEM model syntax**   To fit a two-level SEM, you must specify a model for both levels, as follows:

```
    model <- '
        level: 1
            fw =~ y1 + y2 + y3
            fw ~ x1 + x2 + x3
        level: 2
            fb =~ y1 + y2 + y3
            fb ~ w1 + w2
    '
```

This model syntax contains two blocks, one for level 1, and one for level 2. Within each block, you can specify a model just like in the single-level case. To fit this model, using a toy dataset `Demo.twolevel` that is part of the lavaan package, you need to add the `cluster=` argument to the sem/lavaan function call:

```
    fit <- sem(model = model, data = Demo.twolevel, cluster = "cluster")
```

The output looks similar to a multigroup SEM output, but where the two groups are now the within and the between level respectively.

```
    summary(fit)
```

```
## lavaan (0.6-1) converged normally after  36 iterations
##
```

```
##    Number of observations                              2500
##    Number of clusters [cluster]                         200
##
##    Estimator                                             ML
##    Model Fit Test Statistic                           8.092
##    Degrees of freedom                                    10
##    P-value (Chi-square)                               0.620
##
## Parameter Estimates:
##
##    Information                                     Observed
##    Observed information based on                    Hessian
##    Standard Errors                                 Standard
##
##
## Level 1 [within]:
##
## Latent Variables:
##                    Estimate  Std.Err  z-value  P(>|z|)
##   fw =~
##     y1                1.000
##     y2                0.774    0.034   22.671    0.000
##     y3                0.734    0.033   22.355    0.000
##
## Regressions:
##                    Estimate  Std.Err  z-value  P(>|z|)
##   fw ~
##     x1                0.510    0.023   22.037    0.000
##     x2                0.407    0.022   18.273    0.000
##     x3                0.205    0.021    9.740    0.000
##
## Intercepts:
##                    Estimate  Std.Err  z-value  P(>|z|)
##    .y1               0.000
##    .y2               0.000
##    .y3               0.000
##    .fw               0.000
##
## Variances:
##                    Estimate  Std.Err  z-value  P(>|z|)
##    .y1               0.986    0.046   21.591    0.000
##    .y2               1.066    0.039   27.271    0.000
##    .y3               1.011    0.037   27.662    0.000
##    .fw               0.546    0.040   13.539    0.000
##
##
## Level 2 [cluster]:
##
## Latent Variables:
##                    Estimate  Std.Err  z-value  P(>|z|)
##   fb =~
##     y1                1.000
##     y2                0.717    0.052   13.824    0.000
##     y3                0.587    0.048   12.329    0.000
##
## Regressions:
##                    Estimate  Std.Err  z-value  P(>|z|)
##   fb ~
##     w1                0.165    0.079    2.093    0.036
##     w2                0.131    0.076    1.715    0.086
```

```
##
## Intercepts:
##                   Estimate  Std.Err  z-value  P(>|z|)
##     .y1              0.024    0.075    0.327    0.743
##     .y2             -0.016    0.060   -0.269    0.788
##     .y3             -0.042    0.054   -0.777    0.437
##     .fb              0.000
##
## Variances:
##                   Estimate  Std.Err  z-value  P(>|z|)
##     .y1              0.058    0.047    1.213    0.225
##     .y2              0.120    0.031    3.825    0.000
##     .y3              0.149    0.028    5.319    0.000
##     .fb              0.899    0.118    7.592    0.000
```

After fitting the model, you can inspect the intra-class correlations:

```
    lavInspect(fit, "icc")
```

```
##    y1    y2    y3    x1    x2    x3
## 0.331 0.263 0.232 0.000 0.000 0.000
```

The see the unrestricted (h1) within and between means and covariances, you can use

```
    lavInspect(fit, "h1")
```

```
## $within
## $within$cov
##     y1     y2     y3     x1     x2     x3
## y1  2.000
## y2  0.789  1.674
## y3  0.749  0.564  1.557
## x1  0.489  0.393  0.376  0.982
## x2  0.416  0.322  0.299  0.001  1.011
## x3  0.221  0.160  0.155 -0.006  0.008  1.045
##
## $within$mean
##     y1     y2     y3     x1     x2     x3
##  0.001 -0.002 -0.001 -0.007 -0.003  0.020
##
##
## $cluster
## $cluster$cov
##     y1     y2     y3     w1     w2
## y1  0.992
## y2  0.668  0.598
## y3  0.548  0.391  0.469
## w1  0.125  0.119  0.036  0.870
## w2  0.086  0.057  0.130 -0.128  0.931
##
## $cluster$mean
##     y1     y2     y3     w1     w2
##  0.019 -0.017 -0.043  0.052 -0.091
```

**Important notes**

- note that in `level: 1` the colon follows the `level` keyword; if you type `level 1:`, you will get an error

- you must specify a model for each level; the following syntax is not allowed and will produce an error:

```
model <- '
    level: 1
        fw =~ y1 + y2 + y3
        fw ~ x1 + x2 + x3
    level: 2
'
```

- if you do not have a model in mind for level 2, you can specify a saturated level by adding all variances and covariances of the endogenous variables (here: y1, y2 and y3):

```
model <- '
    level: 1
        fw =~ y1 + y2 + y3
        fw ~ x1 + x2 + x3
    level: 2
        y1 ~~ y1 + y2 + y3
        y2 ~~ y2 + y3
        y3 ~~ y3
'
```

**Convergence issues and solutions**  By default, the current version of lavaan (0.6) uses a quasi-Newton procedure to maximize the loglikelihood of the data given the model (just like in the single-level case). For most model and data combinations, this will work fine (and fast). However, every now and then, you may experience convergence issues.

Non-convergence is typically a sign that something is not quite right with either your model, or your data. Typical settings are: a small number of clusters, in combination with (almost) no variance of an endogenous variable at the between level.

However, if you believe nothing is wrong, you may want to try another optimization procedure. The current version of lavaan allows for using the Expectation Maximization (EM) algorithm as an alternative. To switch to the EM algorithm, you can use:

```
fit <- sem(model = model, data = Demo.twolevel, cluster = "cluster",
           verbose = TRUE, optim.method = "em")
```

As the EM algorithm is not accelerated yet, this may take a long time. It is not unusual that more than 10000 iterations are needed to reach a solution. To control when the EM algorithm stops, you can set the stopping criteria as follows:

```
fit <- sem(model = model, data = Demo.twolevel, cluster = "cluster",
           verbose = TRUE, optim.method = "em", em.iter.max = 20000,
           em.fx.tol = 1e-08, em.dx.tol = 1e-04)
```

The `em.fx.tol` argument is used to monitor the change in loglikelihood between the current step and the previous step. If this change is smaller than `em.fx.tol`, the algorithm stops. The `em.dx.tol` argument is used to monitor the (unscaled) gradient. When a solution is reached, all elements of the gradient should be near zero. When the largest gradient element is smaller than `em.dx.tol`, the algorithm stops.

A word of caution: the EM algorithm can always be forced to 'converge' (perhaps after changing the stopping criteria), but that does not mean you have a model/dataset combination that deserves to converge.