tutorial

Yves Rosseel

Table of contents

1 Overview

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).

2 Before you start

Before you start, please read these points carefully:

- First of all, you must have a recent version (4.0.0 or higher) of R installed. You can download the latest version of R from this page.
- Some important features are NOT available (yet) in lavaan:
  - multilevel sem with random slopes (this is under developement)
  - support for variable types other than continuous, binary and ordinal (for example: zero-inflated count data, nominal data, non-Gaussian continuous data); it is unlikely that this will be part of lavaan any time soon, for the simple reason that these variable types need numerical quadrature, and this is too slow to be practical in (pure) R.
  - support for discrete latent variables (mixture models, latent classes) (although you can use the sampling weights and multiple group features to mimic some mixture models)

  We hope to add these features to lavaan in the near future (but please do not ask when).
- The lavaan package is free open-source software. This means (among other things) that there is no warranty whatsoever. On the other hand, you can verify the source code yourself: https://github.com/yrosseel/lavaan/
- If you need help, you can (only) 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. Please do not email me directly.
- I do not offer statistical advice. For general (non lavaan-specific) questions about SEM, consider posting to the SEMNET discussion group.
- If you think you have found a bug, or if you have a suggestion for improvement, you can either email me directly, or open an issue on github (see https://github.com/yrosseel/lavaan/issues). If you report a bug, always provide a minimal reproducible example (a short R script and some data).

3 Installation

The lavaan package is available on CRAN. Therefore, to install lavaan, simply start up R, and type in the R console:

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

You can check if the installation was succesful by typing

```r
library(lavaan)
```

```
This is lavaan 0.6-15
lavaan is FREE software! Please report any bugs.
```

A startup message will be displayed showing the version number (always report this in your papers), and a reminder that this is free software. If you see this message, you are ready to start.

4 Model syntax 1

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
           '
```

There reason why you should use single quotes is that this is the only way (in R) to allow for double quotes inside a string. See `?Quotes` in R for more information.

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.

You may split your model syntax is multiple parts. For example:

```
part1 <- '   # latent variable definitions
               f1 =~ y1 + y2 + y3
               f2 =~ y4 + y5 + y6
               f3 =~ y7 + y8 + y9 + y10
         '
part2 <- '   # fix covariance between f1 and f2 to zero
               f1 ~~ 0*f2
         '
```

When fitting the model, you may then simply concatenate the multiple parts together as follows:

```
fit <- cfa(model = c(part1, part2), data = myData)
```

5 A CFA example

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). 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.

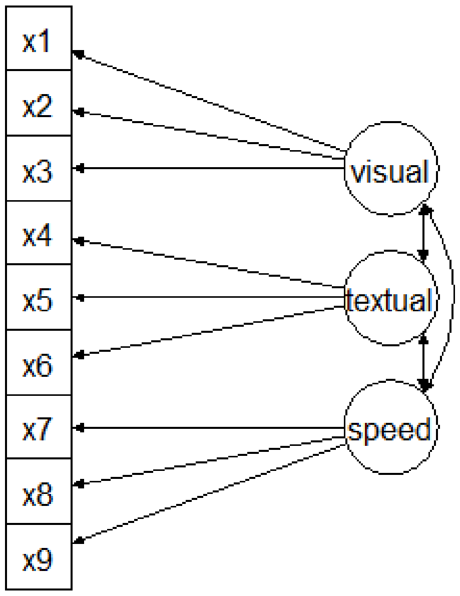

Figure 5.1: A 3 factor CFA example

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 'indicated by' a set of (typically observed) 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 `cfa()` 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 0.6.15 ended normally after 35 iterations

  Estimator                                         ML
  Optimization method                           NLMINB
  Number of model parameters                        21

  Number of observations                           301

Model Test User Model:
  Test statistic                                85.306
  Degrees of freedom                                24
  P-value (Chi-square)                           0.000

Model Test Baseline Model:

  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

  Akaike (AIC)                                7517.490
  Bayesian (BIC)                              7595.339
  Sample-size adjusted Bayesian (SABIC)       7528.739

Root Mean Square Error of Approximation:

  RMSEA                                          0.092
  90 Percent confidence interval - lower         0.071
  90 Percent confidence interval - upper         0.114
  P-value H_0: RMSEA <= 0.050                    0.001
  P-value H_0: RMSEA >= 0.080                    0.840

Standardized Root Mean Square Residual:

  SRMR                                           0.065

Parameter Estimates:

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

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 nine lines are called *the header*. The header contains the following information:

- the lavaan version number
- did optimization end normally or not, and how many iterations were needed
- the estimator that was used (here: `ML`, for maximum likelihood)
- the optimizer that was used to find the best fitting parameter values for this estimator (here: `NLMINB`)
- the number of model parameters (here: `21`)
- the number of observations that were effectively used in the analysis (here: `301`)
- a section called `Model Test User Model:` which provides a test statistic, degrees of freedom, and a p-value for the model that was specified by the user.

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 (technical) information about the method that was used to compute the standard errors. (Most users can safely ignore this information). 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 value equals zero in the population.

Note that in the `Variances:` section, there is a dot before the observed variables names. This is because they are dependent (or endogenous) variables (predicted by the latent variables), and therefore, the value for the variance that is printed in the output is an estimate of the residual variance: the left-over variance that is not explained by the predictor(s). By contrast, there is no dot before the latent variable names, because they are exogenous variables in this model (there are no single-headed arrows pointing to them). The values for the variances here are the estimated *total* variances of the latent variables.

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
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 functions 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.

6 A SEM example

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.

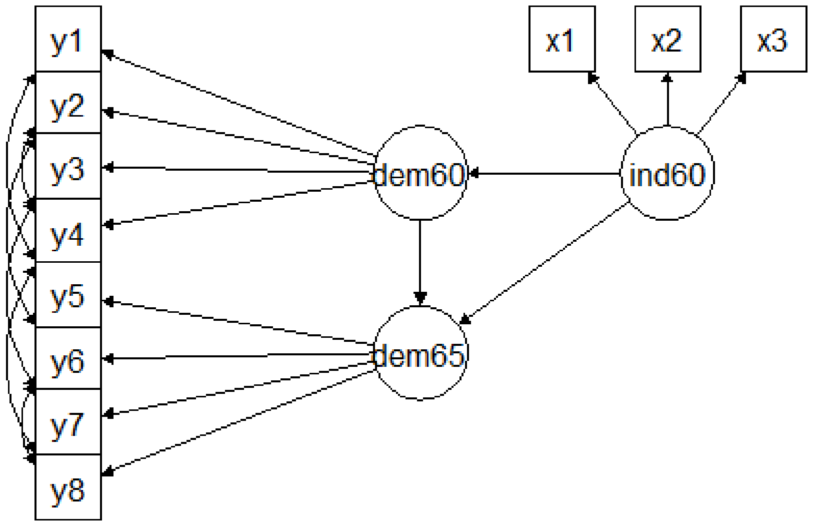

Figure 6.1: Political Democracy SEM example

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`, because the variable on the left of the `~~` operator (`y2`) is the same. 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 0.6.15 ended normally after 68 iterations

  Estimator                                         ML
  Optimization method                           NLMINB
  Number of model parameters                        31

  Number of observations                            75

Model Test User Model:

  Test statistic                                38.125
  Degrees of freedom                                35
  P-value (Chi-square)                           0.329

Parameter Estimates:

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

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:

```
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)
```

7 Model syntax 2

7.1 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 to 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 'missing' and therefore 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 shortcut. 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.

7.2 Starting Values

The lavaan package automatically generates starting values for all free parameters. Normally, this works fine. But if you prefer to 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
```

7.3 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    dem65=~y6 
       2.180        1.819        1.257        1.058        1.265        1.186 
   dem65=~y7    dem65=~y8  dem60~ind60  dem65~ind60  dem65~dem60       y1~~y5 
       1.280        1.266        1.483        0.572        0.837        0.624 
      y2~~y4       y2~~y6       y3~~y7       y4~~y8       y6~~y8       x1~~x1 
       1.313        2.153        0.795        0.348        1.356        0.082 
      x2~~x2       x3~~x3       y1~~y1       y2~~y2       y3~~y3       y4~~y4 
       0.120        0.467        1.891        7.373        5.067        3.148 
      y5~~y5       y6~~y6       y7~~y7       y8~~y8 ind60~~ind60 dem60~~dem60 
       2.351        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 (lhs) of the formula. The middle part is the operator type (op) of the formula, and the third part is the variable in the right-hand side (rhs) 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 "`=~`".)

7.4 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 want to 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.

7.5 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
```

7.6 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:

```r
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:

```r
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:

```r
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 ($\exp(b_2 + b_3)$).

8 Meanstructures

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 0.6.15 ended normally after 35 iterations

  Estimator                                         ML
  Optimization method                           NLMINB
  Number of model parameters                        30

  Number of observations                           301

Model Test User Model:

  Test statistic                                85.306
  Degrees of freedom                                24
  P-value (Chi-square)                           0.000

Parameter Estimates:

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

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.

9 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 0.6.15 ended normally after 57 iterations

  Estimator                                         ML
  Optimization method                           NLMINB
  Number of model parameters                        60

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

Model Test User Model:

  Test statistic                               115.851
  Degrees of freedom                                48
  P-value (Chi-square)                           0.000
  Test statistic for each group:
    Pasteur                                     64.309
    Grant-White                                 51.542

Parameter Estimates:

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

```
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
    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 (which is not recommended), that element will be applied for all groups. If you specify a single label, this will generate a warning as this would imply equality constraints across groups. For example:

```
HS.model <- ' visual  =~ x1 + 0.5*x2 + c(0.6, 0.8)*x3
              textual =~ x4 + start(c(1.2, 0.6))*x5 + c(a1, a2)*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 `a1` in the first group, and `a2` in the second group. It may be tempting to write `a*x6`. But using a single label in a multiple group setting has a double effect: it gives the label `a` to the factor loading of `x6` in both groups, and as a result, those two parameters are now constrained to be equal. Because this may unintended, lavaan will produce a warning message about this. If this is really intended, it is much better to use a vector of labels: `c(a, a)*x6`.

To verify the effects of our modifiers, we refit the model:

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

```
lavaan 0.6.15 ended normally after 45 iterations
```

```
  Estimator                                         ML
  Optimization method                           NLMINB
  Number of model parameters                        56

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

Model Test User Model:

  Test statistic                               118.976
  Degrees of freedom                                52
  P-value (Chi-square)                           0.000
  Test statistic for each group:
    Pasteur                                     64.901
    Grant-White                                 54.075

Parameter Estimates:

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


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        (a1)    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   24.320    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        (a2)    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    6.177    0.000
    speed             0.453    0.117    3.871    0.000
```

9.1 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
```

9.2 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.

9.3 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 0.6.15 ended normally after 42 iterations

  Estimator                                         ML
  Optimization method                           NLMINB
  Number of model parameters                        60
  Number of equality constraints                     6
  Number of observations per group:
    Pasteur                                          156
    Grant-White                                      145

Model Test User Model:

  Test statistic                                 124.044
  Degrees of freedom                                  54
  P-value (Chi-square)                             0.000
  Test statistic for each group:
    Pasteur                                       68.825
    Grant-White                                   55.219

Parameter Estimates:

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


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
```

The `.p2.`, `.p3.`, `.p5`, … labels which appear in the output have been auto-generated to impose the equality constraints. 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:

```r
fit <- cfa(HS.model,
           data = HolzingerSwineford1939,
           group = "school",
           group.equal = c("loadings", "intercepts"),
           group.partial = c("visual=~x2", "x7~1"))
```

9.4 Measurement invariance testing

Before we compare, say, the values of latent means across multiple groups, we first need to establish measurement invariance. When data is continuous, testing for measurement invariance involves a fixed sequence of model comparison tests. A typical sequence involves three models:

1. Model 1: configural invariance. The same factor structure is imposed on all groups.
2. Model 2: weak invariance. The factor loadings are constrained to be equal across groups.
3. Model 3: strong invariance. The factor loadings and intercepts are constrained to be equal across groups.

In lavaan, we can proceed as follows:

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

# configural invariance
fit1 <- cfa(HS.model, data = HolzingerSwineford1939, group = "school")

# weak invariance
fit2 <- cfa(HS.model, data = HolzingerSwineford1939, group = "school",
            group.equal = "loadings")

# strong invariance
fit3 <- cfa(HS.model, data = HolzingerSwineford1939, group = "school",
            group.equal = c("intercepts", "loadings"))

# model comparison tests
lavTestLRT(fit1, fit2, fit3)
```

```
Chi-Squared Difference Test

     Df    AIC    BIC  Chisq Chisq diff    RMSEA Df diff Pr(>Chisq)
fit1 48 7484.4 7706.8 115.85
fit2 54 7480.6 7680.8 124.04      8.192 0.049272       6     0.2244
fit3 60 7508.6 7686.6 164.10     40.059 0.194211       6  4.435e-07 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The `lavTestLRT()` function can be used for model comparison tests. Because we provided three model fits, it will produce two tests: the first test compares the first model versus the second model, while the second test compares the second model versus the third model. Because the first p-value is non-significant, we may conclude that weak invariance (equal factor loadings) is supported in this dataset. However, because the second p-value is significant, strong invariance is not. Therefore, it is unwise to directly compare the values of the latent means across the two groups.

10 Growth curves

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. If `i` and `s` are the only 'latent variables' in the model, we can use the `growth()` function to fit this model:

```
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 0.6.15 ended normally after 29 iterations

  Estimator                                         ML
  Optimization method                           NLMINB
  Number of model parameters                         9

  Number of observations                           400

Model Test User Model:

  Test statistic                                 8.069
  Degrees of freedom                                 5
  P-value (Chi-square)                           0.152

Parameter Estimates:
  Standard errors                             Standard
  Information                                 Expected
  Information saturated (h1) model          Structured

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 complete R code needed to specify and fit this linear growth model with a time-varying covariate is given below:

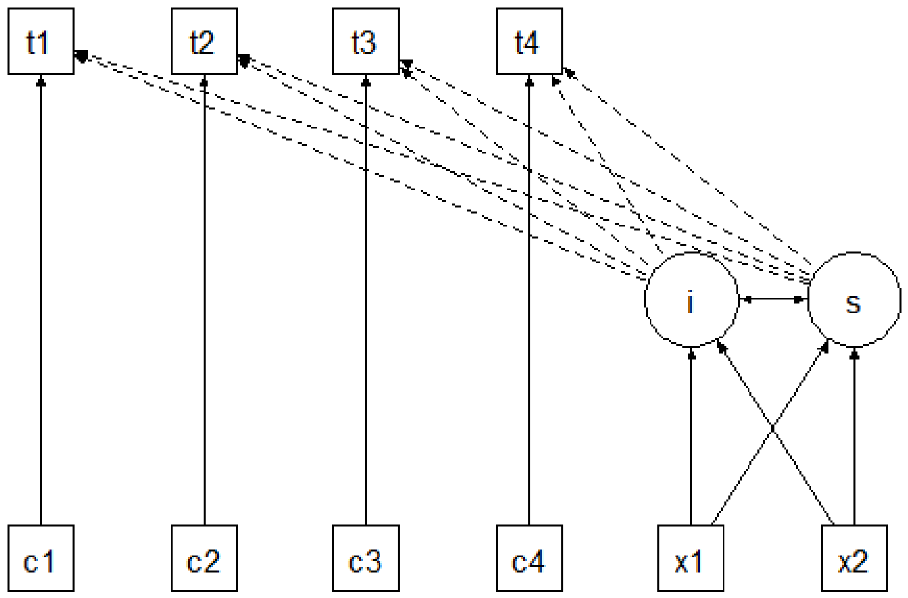

Figure 10.1: A growth curve examples

```
# 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)
```

11 Categorical data

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.

11.1 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.

11.2 Endogenous categorical variables

The lavaan 0.5 series can deal with binary and ordinal (but not nominal) endogenous variables. There are two ways to communicate to lavaan that some of the endogenous variables are to be treated as categorical:

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"))
```

If all the (endogenous) variables are to be treated as categorical, you can use `ordered = TRUE` as a shortcut.

When the `ordered=` argument is used, 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. Other options are unweighted least squares (ULSMV), or pairwise maximum likelihood (PML). Full information maximum likelihood is currently not supported.

12 Covariance matrix 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
```

```r
'
fit <- sem(wheaton.model,
           sample.cov = wheaton.cov,
           sample.nobs = 932)
summary(fit, standardized = TRUE)
```

```
lavaan 0.6.15 ended normally after 84 iterations

  Estimator                                         ML
  Optimization method                           NLMINB
  Number of model parameters                        17

  Number of observations                           932

Model Test User Model:

  Test statistic                                 4.735
  Degrees of freedom                                 4
  P-value (Chi-square)                           0.316

Parameter Estimates:

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

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
```

12.1 The `sample.cov.rescale` argument

If the estimator is `ML` (the default), then the sample variance-covariance matrix will be rescaled by a factor (N-1)/N. The reasoning is the following: the elements in a sample variance-covariance matrix have (usually) been divided by N-1. But the (normal-based) ML estimator would divide the elements by N. Therefore, we need to rescale. If you don't want this to happen (for example in a simulation study), you can provide the argument `sample.cov.rescale = FALSE`.

12.2 Multiple groups

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.

13 Estimators and more

13.1 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
- `"DLS"`: distributionally-weighted least squares
- `"PML"`: pairwise maximum likelihood

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.

13.2 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). If you prefer to use an unbiased covariance matrix, 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 0.6.15 ended normally after 35 iterations

  Estimator                                         ML
  Optimization method                           NLMINB
  Number of model parameters                        21

  Number of observations                           301

Model Test User Model:

  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.

13.3 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.

13.4 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.

Robust standard errors can be requested explicitly by using `se = "robust"`. Similarly, robust test statistics can be requested explicitly by using `test = "robust"`. Many more options are possible. See the help page:

```
?lavOptions
```

13.5 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.

14 Mediation

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 0.6.15 ended normally after 1 iteration

  Estimator                                         ML
  Optimization method                           NLMINB
  Number of model parameters                         5

  Number of observations                           100

Model Test User Model:

  Test statistic                                 0.000
  Degrees of freedom                                 0

Parameter Estimates:

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

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.

15 Modification indices

Modification indices can be requested by adding the argument `modindices = TRUE` in the `summary()` call, or by calling the function `modindices()` directly. By default, modification indices are printed out for each nonfree (or fixed-to-zero) 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).

A typical use of the `modindices()` function is as follows:

```
fit <- cfa(HS.model,
           data = HolzingerSwineford1939)
modindices(fit, sort = TRUE, maximum.number = 5)
```

```
       lhs op rhs     mi    epc sepc.lv sepc.all sepc.nox
30  visual =~  x9 36.411  0.577   0.519    0.515    0.515
76      x7 ~~  x8 34.145  0.536   0.536    0.859    0.859
28  visual =~  x7 18.631 -0.422  -0.380   -0.349   -0.349
78      x8 ~~  x9 14.946 -0.423  -0.423   -0.805   -0.805
33 textual =~  x3  9.151 -0.272  -0.269   -0.238   -0.238
```

This will print out the top 5 parameters (that can be added to the model) that result in the largest modification index, sorted from high to low.

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
```

It is important to realize that the `modindices()` function will only consider fixed-to-zero parameters. If you have equality constraints in the model, and you wish to examine what happens if you release all (or some) of these equality constraints, use the `lavTestScore()` function.

16 Extracting information

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.

16.1 parameterEstimates

The `parameterEstimates()` function returns a data.frame containing all the model parameters in the rows:

```
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
```

The `lhs` (left-hand side), `op` (operator) and `rhs` (right-hand side) columns define the parameter. The `est`, `se`, `z` and `pvalue` columns provide the point estimate, the standard error, the z-value and the p-value for this parameter. The last two columns are the lower and upper bounds of a 95% confidence interval around the point estimate.

16.2 standardizedSolution

The `standardizedSolution()` function is similar to the `parameterEstimates()` function, but only shows the standardized parameter estimates and corresponding standard errors, z-values, p-values and confidence intervals.

16.3 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
```

16.4 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.

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

```
$type
[1] "raw"
```

```
$cov
       x1     x2     x3     x4     x5     x6     x7     x8     x9
x1  0.000
x2 -0.041  0.000
x3 -0.010  0.124  0.000
x4  0.097 -0.017 -0.090  0.000
x5 -0.014 -0.040 -0.219  0.008  0.000
x6  0.077  0.038 -0.032 -0.012  0.005  0.000
x7 -0.177 -0.242 -0.103  0.046 -0.050 -0.017  0.000
x8 -0.046 -0.062 -0.013 -0.079 -0.047 -0.024  0.082  0.000
x9  0.175  0.087  0.167  0.056  0.086  0.062 -0.042 -0.032  0.000
```

The `lavResiduals()` gives more extensive information about the residuals. Per default, it will print both raw and standardized residuals, as well as several summary statistics (including the SRMR and the unbiased SRMR).

16.5 vcov

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

16.6 AIC and BIC

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

16.7 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.ci.level        rmsea.pvalue 
              0.114               0.900               0.001 
     rmsea.close.h0 rmsea.notclose.pvalue     rmsea.notclose.h0 
              0.050               0.840               0.080 
                rmr          rmr_nomean                srmr 
              0.082               0.082               0.065 
       srmr_bentler  srmr_bentler_nomean                crmr 
              0.065               0.065               0.073 
        crmr_nomean          srmr_mplus   srmr_mplus_nomean 
              0.073               0.065               0.065 
              cn_05               cn_01                 gfi 
            129.490             152.654               0.943 
               agfi                pgfi                 mfi 
              0.894               0.503               0.903 
               ecvi 
              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:

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

```
  cfi 
0.931 
```

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

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

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

16.8 lavInspect

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 `lavInspect()` function, with a variety of options. By default, calling `lavInspect()` 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.

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

```r
lavInspect(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

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

```
   id     lhs op     rhs user block group free ustart exo label plabel start
1   1  visual =~      x1    1     1     1    0      1   0         .p1. 1.000
2   2  visual =~      x2    1     1     1    1     NA   0         .p2. 0.778
3   3  visual =~      x3    1     1     1    2     NA   0         .p3. 1.107
4   4 textual =~      x4    1     1     1    0      1   0         .p4. 1.000
5   5 textual =~      x5    1     1     1    3     NA   0         .p5. 1.133
6   6 textual =~      x6    1     1     1    4     NA   0         .p6. 0.924
7   7   speed =~      x7    1     1     1    0      1   0         .p7. 1.000
8   8   speed =~      x8    1     1     1    5     NA   0         .p8. 1.225
9   9   speed =~      x9    1     1     1    6     NA   0         .p9. 0.854
10 10      x1 ~~      x1    0     1     1    7     NA   0        .p10. 0.679
11 11      x2 ~~      x2    0     1     1    8     NA   0        .p11. 0.691
12 12      x3 ~~      x3    0     1     1    9     NA   0        .p12. 0.637
13 13      x4 ~~      x4    0     1     1   10     NA   0        .p13. 0.675
14 14      x5 ~~      x5    0     1     1   11     NA   0        .p14. 0.830
15 15      x6 ~~      x6    0     1     1   12     NA   0        .p15. 0.598
16 16      x7 ~~      x7    0     1     1   13     NA   0        .p16. 0.592
17 17      x8 ~~      x8    0     1     1   14     NA   0        .p17. 0.511
18 18      x9 ~~      x9    0     1     1   15     NA   0        .p18. 0.508
19 19  visual ~~  visual    0     1     1   16     NA   0        .p19. 0.050
```

```
20 20 textual ~~ textual    0     1     1   17     NA   0         .p20. 0.050
21 21   speed ~~   speed    0     1     1   18     NA   0         .p21. 0.050
22 22  visual ~~ textual    0     1     1   19     NA   0         .p22. 0.000
23 23  visual ~~   speed    0     1     1   20     NA   0         .p23. 0.000
24 24 textual ~~   speed    0     1     1   21     NA   0         .p24. 0.000
     est    se
1  1.000 0.000
2  0.554 0.100
3  0.729 0.109
4  1.000 0.000
5  1.113 0.065
6  0.926 0.055
7  1.000 0.000
8  1.180 0.165
9  1.082 0.151
10 0.549 0.114
11 1.134 0.102
12 0.844 0.091
13 0.371 0.048
14 0.446 0.058
15 0.356 0.043
16 0.799 0.081
17 0.488 0.074
18 0.566 0.071
19 0.809 0.145
20 0.979 0.112
21 0.384 0.086
22 0.408 0.074
23 0.262 0.056
24 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 lavInspect options, see the help page:

```
?lavInspect
```

17 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).

17.1 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 0.6.15 ended normally after 36 iterations

  Estimator                                         ML
  Optimization method                           NLMINB
  Number of model parameters                        20

  Number of observations                          2500
  Number of clusters [cluster]                     200
```

```
Model Test User Model:

  Test statistic                                 8.092
  Degrees of freedom                                10
  P-value (Chi-square)                           0.620

Parameter Estimates:

  Standard errors                             Standard
  Information                                 Observed
  Observed information based on                Hessian


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 
```

17.2 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
  '
  ```

17.3 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:

```r
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:

```r
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.

18 ESEM and EFA

If a measurement model contains multiple latent variables (factors), we usually know which indicators belong to each factor. We call this the factor structure. Confirmatory factor analysis can be used to check if this a priori factor structure holds in the data. There are settings, however, where the factor structure is unclear, and we wish to rotate the solution in order to find a suitable structure in a given model. When the model also includes a structural part (i.e., regressions among the latent variables), this is referred to as exploratory structural equation modeling or ESEM. If there is only a measurement part, this is called exploratory factor analysis (EFA). What they have in common is that the factor structure (for one or more blocks) is found by means of rotation.

18.1 ESEM

To illustrate how ESEM works in lavaan, consider the following syntax:

```
model <- '
    # efa block 1
    efa("efa1")*f1 +
    efa("efa1")*f2 =~ x1 + x2 + x3 + x4 + x5 + x6

    # efa block 2
    efa("efa2")*f3 +
    efa("efa2")*f4 =~ y1 + y2 + y3 + y4 + y5 + y6

    # cfa block
    f5 =~ z7 + z8 + z9
    f6 =~ z10 + z11 + z12

    # regressions
    f3 ~ f1 + f2
    f4 ~ f3
'
```

This model syntax defines six latent variables (or factors). For f5 and f6, the factor structure is known, and they belong to a regular CFA block. But for f1 and f2, the factor structure is not known, and we will use a rotation method to find an appropiate structure. The f1 and f2 factors belong together in an EFA block that is (arbitrarily) named `efa1`. The `efa("efa1")*` modifier just before f1 and f2 is used to alert lavaan that these two factors belong to the same EFA block. The factors f3 and f4 belong to a different EFA block (named `efa2`) and will be rotated independently. The structural part of the model is given as usual. To fit this model, we could call the `sem()` function as follows:

```
fit <- sem(model = model, data = myData, rotation = "geomin")
```

Different rotation criteria are available, and many rotation options can be provided (see the manual page for the `efa()` function for an overview).

To illustrate ESEM, we will borrow an example from the Mplus User's Guide (example 5.25). First we read in the data:

```
ex5_25 <- read.table("http://statmodel.com/usersguide/chap5/ex5.25.dat")
names(ex5_25) = paste0("y",1:12)
```

The model syntax contains a single EFA block (`efa1` for factors f1 and f2) and single CFA block (for f3 and f4):

```
model <- '
    # efa block
    efa("efa1")*f1 +
    efa("efa1")*f2 =~ y1 + y2 + y3 + y4 + y5 + y6

    # cfa block
    f3 =~ y7 + y8 + y9
    f4 =~ y10 + y11 + y12

    # regressions
    f3 ~ f1 + f2
    f4 ~ f3
'
```

The following command illustrates the use of various rotation arguments:

```
fit <- sem(model = model, data = ex5_25, rotation = "geomin",
           # mimic Mplus
           information = "observed",
           rotation.args = list(rstarts = 30, row.weights = "none",
                                algorithm = "gpa", std.ov = TRUE,
                                geomin.epsilon = 0.0001))
summary(fit)
```

```
lavaan 0.6.15 ended normally after 35 iterations

  Estimator                                         ML
  Optimization method                           NLMINB
  Number of model parameters                        32

  Rotation method                       GEOMIN OBLIQUE
  Geomin epsilon                                 1e-04
  Rotation algorithm (rstarts)                GPA (30)
  Standardized metric                             TRUE
  Row weights                                     None

  Number of observations                           500

Model Test User Model:

  Test statistic                                51.353
  Degrees of freedom                                46
  P-value (Chi-square)                           0.272

Parameter Estimates:

  Standard errors                             Standard
  Information                                 Observed
  Observed information based on                Hessian

Latent Variables:
                   Estimate  Std.Err  z-value  P(>|z|)
  f1 =~ efa1
    y1                0.751    0.048   15.621    0.000
    y2                0.858    0.042   20.469    0.000
    y3                0.736    0.045   16.343    0.000
    y4                0.036    0.051    0.712    0.476
    y5               -0.028    0.049   -0.564    0.573
    y6                0.002    0.003    0.694    0.488
  f2 =~ efa1
    y1                0.034    0.045    0.758    0.449
    y2               -0.002    0.015   -0.151    0.880
    y3               -0.008    0.035   -0.219    0.827
    y4                0.763    0.050   15.374    0.000
    y5                0.810    0.048   16.796    0.000
    y6                0.802    0.041   19.467    0.000
  f3 =~
    y7                1.000
    y8                0.894    0.021   41.936    0.000
    y9                0.902    0.021   42.479    0.000
  f4 =~
    y10               1.000
    y11               0.734    0.028   26.424    0.000
    y12               0.684    0.028   24.405    0.000

Regressions:
                   Estimate  Std.Err  z-value  P(>|z|)
  f3 ~
    f1                0.493    0.058    8.455    0.000
    f2                0.721    0.057   12.755    0.000
  f4 ~
    f3                0.546    0.032   16.975    0.000

Covariances:
                   Estimate  Std.Err  z-value  P(>|z|)
  f1 ~~
    f2                0.479    0.053    9.072    0.000

Variances:
                   Estimate  Std.Err  z-value  P(>|z|)
   .y1                0.376    0.034   11.064    0.000
   .y2                0.290    0.035    8.239    0.000
   .y3                0.406    0.034   11.817    0.000
   .y4                0.408    0.035   11.742    0.000
   .y5                0.329    0.033   10.046    0.000
   .y6                0.393    0.035   11.073    0.000
   .y7                0.183    0.019    9.796    0.000
   .y8                0.191    0.017   11.269    0.000
   .y9                0.181    0.017   10.812    0.000
   .y10               0.240    0.027    8.746    0.000
   .y11               0.183    0.017   10.791    0.000
   .y12               0.213    0.018   11.998    0.000
    f1                1.000
    f2                1.000
   .f3                0.527    0.049   10.644    0.000
   .f4                0.565    0.049   11.488    0.000
```

18.2 Exploratory factor analysis (EFA)

When there is no structural part (i.e., no regressions among the latent variables) and there is only a single EFA block, then ESEM reduces to exploratory factor analysis (EFA). Using the Holzinger and Swineford data, we could specify an EFA with three factors as follows:

```
efa.model <- '
    efa("efa")*f1 +
    efa("efa")*f2 +
    efa("efa")*f3 =~ x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9
'
fit <- cfa(efa.model, data = HolzingerSwineford1939)
summary(fit, standardized = TRUE)
```

```
lavaan 0.6.15 ended normally after 1 iteration

  Estimator                                         ML
  Optimization method                           NLMINB
  Number of model parameters                        33

  Rotation method                       GEOMIN OBLIQUE
  Geomin epsilon                                 0.001
  Rotation algorithm (rstarts)                GPA (30)
  Standardized metric                             TRUE
  Row weights                                     None

  Number of observations                           301

Model Test User Model:

  Test statistic                                22.897
  Degrees of freedom                                12
  P-value (Chi-square)                           0.029

Parameter Estimates:

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

Latent Variables:
                   Estimate  Std.Err  z-value  P(>|z|)   Std.lv  Std.all
  f1 =~ efa
    x1                0.712    0.092    7.771    0.000    0.712    0.611
    x2                0.628    0.104    6.063    0.000    0.628    0.534
    x3                0.796    0.096    8.255    0.000    0.796    0.705
    x4                0.011    0.011    0.944    0.345    0.011    0.009
    x5               -0.107    0.089   -1.203    0.229   -0.107   -0.083
    x6                0.076    0.073    1.028    0.304    0.076    0.069
    x7               -0.278    0.109   -2.538    0.011   -0.278   -0.255
    x8                0.012    0.008    1.371    0.170    0.012    0.011
    x9                0.314    0.076    4.142    0.000    0.314    0.312
  f2 =~ efa
    x1                0.198    0.103    1.917    0.055    0.198    0.170
    x2                0.039    0.092    0.424    0.672    0.039    0.033
    x3               -0.106    0.111   -0.963    0.335   -0.106   -0.094
    x4                0.981    0.058   16.850    0.000    0.981    0.844
    x5                1.153    0.074   15.545    0.000    1.153    0.895
    x6                0.886    0.062   14.338    0.000    0.886    0.810
    x7                0.011    0.012    0.923    0.356    0.011    0.010
    x8               -0.075    0.066   -1.135    0.256   -0.075   -0.074
    x9               -0.002    0.007   -0.315    0.753   -0.002   -0.002
  f3 =~ efa
    x1                0.015    0.048    0.302    0.762    0.015    0.012
    x2               -0.166    0.092   -1.813    0.070   -0.166   -0.141
    x3                   0.002    0.048    0.036    0.971    0.002    0.002
    x4                   0.004    0.047    0.091    0.927    0.004    0.004
    x5                   0.012    0.036    0.322    0.747    0.012    0.009
    x6                  -0.017    0.041   -0.409    0.683   -0.017   -0.015
    x7                   0.843    0.105    7.999    0.000    0.843    0.775
    x8                   0.752    0.076    9.893    0.000    0.752    0.744
    x9                   0.484    0.070    6.954    0.000    0.484    0.481

Covariances:
                   Estimate  Std.Err  z-value  P(>|z|)   Std.lv  Std.all
  f1 ~~
    f2                   0.373    0.118    3.173    0.002    0.373    0.373
    f3                   0.432    0.097    4.465    0.000    0.432    0.432
  f2 ~~
    f3                   0.306    0.081    3.775    0.000    0.306    0.306

Variances:
                   Estimate  Std.Err  z-value  P(>|z|)   Std.lv  Std.all
   .x1                   0.696    0.087    8.038    0.000    0.696    0.513
   .x2                   1.035    0.102   10.151    0.000    1.035    0.749
   .x3                   0.692    0.097    7.134    0.000    0.692    0.543
   .x4                   0.377    0.048    7.902    0.000    0.377    0.279
   .x5                   0.403    0.061    6.590    0.000    0.403    0.243
   .x6                   0.365    0.042    8.613    0.000    0.365    0.305
   .x7                   0.594    0.106    5.624    0.000    0.594    0.502
   .x8                   0.479    0.080    5.958    0.000    0.479    0.469
   .x9                   0.551    0.060    9.132    0.000    0.551    0.543
    f1                   1.000                               1.000    1.000
    f2                   1.000                               1.000    1.000
    f3                   1.000                               1.000    1.000
```

In version 0.6-13, we added added the `efa()` function to simplify the input, and to produce output that is more in line with traditional EFA software in R. There is no need to create a model syntax. You only need to provide the data, and the number of factors. Instead of a single number, you can also specify a range of numbers. For example:

```r
var.names <- paste("x", 1:9, sep = "")
fit <- efa(data = HolzingerSwineford1939[,var.names], nfactors = 1:3)
summary(fit)
```

```
This is lavaan 0.6.15 -- running exploratory factor analysis

  Estimator                                         ML
  Rotation method                       GEOMIN OBLIQUE
  Geomin epsilon                                 0.001
  Rotation algorithm (rstarts)                GPA (30)
  Standardized metric                               TRUE
  Row weights                                       None

  Number of observations                             301

Overview models:
                    aic      bic    sabic   chisq df pvalue   cfi rmsea
  nfactors = 1 7738.448 7805.176 7748.091 312.264 27  0.000 0.677 0.187
  nfactors = 2 7572.491 7668.876 7586.418 130.306 19  0.000 0.874 0.140
  nfactors = 3 7479.081 7601.416 7496.758  22.897 12  0.029 0.988 0.055

Eigenvalues correlation matrix:

    ev1     ev2     ev3     ev4     ev5     ev6     ev7     ev8     ev9
  3.216   1.639   1.365   0.699   0.584   0.500   0.473   0.286   0.238

Number of factors:  1

Standardized loadings: (* = significant at 1% level)

       f1       unique.var   communalities
x1  0.438*           0.808           0.192
x2      .*           0.951           0.049
x3      .*           0.950           0.050
x4  0.848*           0.281           0.719
x5  0.841*           0.293           0.707
x6  0.838*           0.298           0.702
x7      .*           0.967           0.033
x8      .*           0.960           0.040
x9  0.307*           0.906           0.094

                           f1
Sum of squared loadings 2.586
Proportion of total     1.000
Proportion var          0.287
Cumulative var          0.287

Number of factors:  2

Standardized loadings: (* = significant at 1% level)

       f1      f2       unique.var   communalities
x1      .*  0.430*           0.673           0.327
x2      .       .*           0.906           0.094
x3          0.456*           0.783           0.217
x4  0.851*                   0.274           0.726
x5  0.868*                   0.264           0.736
```

```
x6  0.825*                   0.302           0.698
x7          0.448*           0.802           0.198
x8          0.627*           0.630           0.370
x9          0.734*           0.458           0.542

                              f1    f2 total
Sum of sq (obliq) loadings 2.280 1.629 3.909
Proportion of total        0.583 0.417 1.000
Proportion var             0.253 0.181 0.434
Cumulative var             0.253 0.434 0.434

Factor correlations: (* = significant at 1% level)

       f1      f2
f1  1.000
f2  0.339*  1.000

Number of factors:  3

Standardized loadings: (* = significant at 1% level)

       f1      f2      f3       unique.var   communalities
x1  0.611*      .                    0.513           0.487
x2  0.534*              .            0.513           0.251
x3  0.705*                           0.543           0.457
x4          0.844*                   0.279           0.721
x5          0.895*                   0.243           0.757
x6          0.810*                   0.305           0.695
x7      .           0.775*           0.502           0.498
x8                  0.744*           0.469           0.531
x9  0.312*          0.481*           0.543           0.457

                              f2    f3    f1 total
Sum of sq (obliq) loadings 2.215 1.343 1.297 4.855
Proportion of total        0.456 0.277 0.267 1.000
Proportion var             0.246 0.149 0.144 0.539
Cumulative var             0.246 0.395 0.539 0.539

Factor correlations: (* = significant at 1% level)

       f1      f2      f3
f1  1.000
f2  0.373*  1.000
f3  0.432*  0.306*  1.000
```