Introduction to 'simsem' package

Beta version 0.0.2

Sunthud Pornprasertmanit

This R package is developed for data simulation for structural equation modeling (SEM). This package will help analysts to create simulated data from their hypotheses or their analytic results from obtained data. The simulated data can be used for different purposes, such as power analysis, model evaluation, and planned missing design. The material in this version of the introduction will emphasize on building simulated sampling distribution (SSD) for fit indices in order to evaluate model fit. This will help researchers tailor their fit indices cutoff based on a priori alpha level. In this manual, we will show how to find SSD for absolute model fit with various model specification. Also, we will show how to find power on parameter estimates and fit indices. We will introduce this package with examples.

Installing simsem package

Find the appropriate compressed file (`tar.gz` from Linux or `zip` for Windows).

For Linux users, install the package by typing this line in R.

```
install.packages("simsem_0.0-2.tar.gz", repos=NULL)
```

You may change the file name by including the correct directory.

For Windows users, open R, go to the menu bar, click on `Packages` → `Install package(s) for local zip files…`, and choose the `simsem_0.0-2.zip` file.

After installing the package, when you open `R` session, you can use the package by

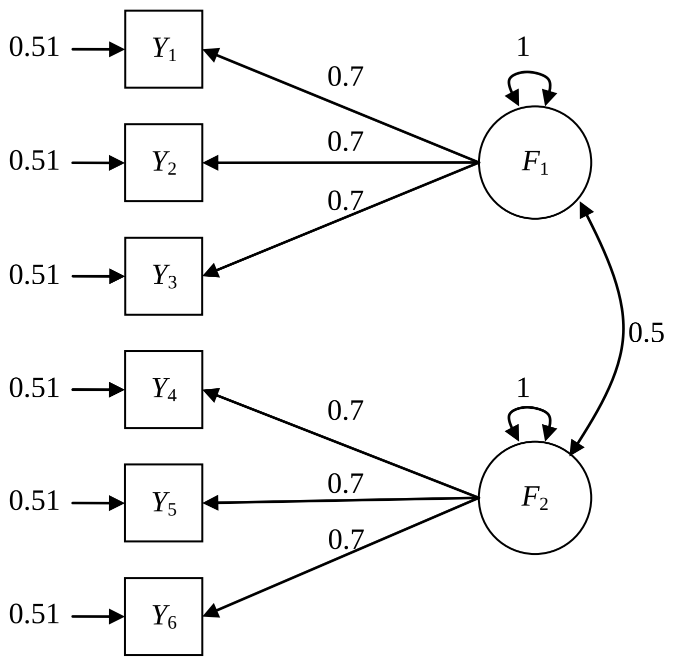
```
library(simsem)
```

Getting Started

Example 1

Let's start with very simple example, confirmatory factor analysis (CFA) model with two factors and three indicators each. Factor loadings are .7. Error variances are .51 to make indicator variances equal to 1. Factor correlation is .5.

Latest Updated: November 16, 2011



This package will specify SEM model in matrix format, like LISREL. To specify a CFA model, three matrices are required: factor loading, error covariance, and factor covariance matrices. However, this package will specify matrices similar but not exactly the same as LISREL specification. I will describe when there are any differences.

In creating a matrix, this program will call matrix as matrix object. Matrix object has two components: parameters and starting values. In the parameters part, elements of the matrix can be divided to two types: NA and numbers. NA means that the element is freely estimated in the model. Number means that the element is fixed as a specified number, usually as 0. For example, with fixed factor method of scaling identification, factor loading matrix parameters will be NA in elements (1,1), (2,1), (3,1), (4,2), (5,2), (6,2). Other elements in the matrix are 0. This can be scripted in R as

```
loading <- matrix(0, 6, 2)
loading[1:3, 1] <- NA
loading[4:6, 2] <- NA
```

The second part is the parameter values (for data generation) or starting values (for data analysis) of free parameters. In data simulation, these parameter/starting values will be used as data generation model. The elements can be numbers for fixed parameters or distribution object for random parameters (which will be clarified in next example). In this example, all starting values of factor loading matrix are 0.7. Thus, a new matrix with 6 rows and 2 columns is created and the (1,1), (2,1), (3,1), (4,2), (5,2), and (6,2) elements are specified as 0.7. The R script is

```
loadingValues <- matrix(0, 6, 2)
loadingValues[1:3, 1] <- 0.7
loadingValues[4:6, 2] <- 0.7
```

Next, combine two parts to create the factor loading matrix object by `simMatrix` command as

```
LX <- simMatrix(loading, loadingValues)
```

If the parameter/starting values of a matrix are the same for all parameters, instead of a matrix, one starting value can be put in the `simMatrix` command as

```
LX <- simMatrix(loading, 0.7)
```

Users can view all specifications in a matrix object by summary function as

```
summary(LX)
```

In the parameter/starting values part, you will notice that if an element is not free, the parameter/starting value will be automatically set as blanks.

For error covariance matrix, this program will separate error covariance matrix into two parts: a vector of error variance (or indicator variance) and error correlation, which is different from LISREL. By default, indicator variances (as well as factor variances, which will be described later) are set to be 1. Thus, factor loading can be interpreted as standardized factor loading. Error variances by default are free parameters. From this example, the error variances are .51, which implies that indicator variances are 1 ($.7 \times 1 \times .7 + .51$). Therefore, we will not set error variances (or indicator variances) and use program default by skipping specifying error variances and set only error correlation. There is no error correlation in this example; therefore, error correlation is set to be identity matrix without any free parameters.

```
error.cor <- matrix(0, 6, 6)
diag(error.cor) <- 1
```

Because of no free parameters in error correlation matrix, parameter/starting values are not applicable. Next, make error correlation matrix as symmetric matrix object by `symMatrix` function as

```
TD <- symMatrix(error.cor)
```

The `symMatrix` structure is similar to `simMatrix`. The main difference is to make more control on free parameters and constants such that elements above and below diagonal elements are equal. The parameter/starting values are not required in this command so there is only one attribute of free parameters in this function.

The last matrix is factor covariance matrix. Again, factor covariance matrix is separated to two parts: factor variances (or factor residual variances) vector and factor correlation (or factor residual correlation). The default in this program is that the factor variances are constrained to be 1. All exogenous and endogenous factors variances are fixed parameters. Therefore, the only thing we need to specify is factor correlation. For all correlation matrices, diagonal elements are 1. As in the model, we allow the only factor correlation to be freely estimated and to have parameter/starting value of 0.5. Thus, latent correlation matrix can be specified as

```
latent.cor <- matrix(NA, 2, 2)
diag(latent.cor) <- 1
```

The symmetric matrix object is created for this factor correlation by

```
PH <- symMatrix(latent.cor, 0.5)
```

At this point, all required matrices for CFA are specified. The next step is to create set of matrices object. For this example, `simSetCFA` will be used. This can be scripted as

```
CFA.Model <- simSetCFA(LX = LX, PH = PH, TD = TD)
```

Similar to LISREL notation, `LX` means factor loading matrix, `PH` means factor correlation matrix, and `TD` means error correlation matrix. You may notice that `PH` and `TD` means covariance matrices in LISREL. We use them as correlation matrices here. This step applies the default of program by freeing error variances and fixing factor variances. This default can be seen by `summary` function as

```
summary(CFA.Model)
```

The summary will show all starting values in the models, including defaults. You may notice that all *X*-side in LISREL notation are changed to *Y*-side notation automatically. The `simSetCFA` can be specified as *Y*-side also as

```
CFA.Model <- simSetCFA(LY = LX, PS = PH, TE = TD)
```

This set of CFA matrices will be used to create data generation object and analysis model object. Basically, we will put CFA model to data generation object to create data and to analysis model object to be the basis of analysis model on simulated data. The data and model objects do not need to have the same analysis (e.g., CFA) model. However, in this example, I will use the same model, CFA two factors with three indicators each without any additional constraints, in both data and model objects.

First, data object can be specified by `simData` function as

```
SimData <- simData(200, CFA.Model)
```

The first argument is desired sample size, which is 200 in this example. The second argument is the matrix set. You can see the specification of the data object by `summary` function as well. From this, you are ready to simulate data from the analysis model in this example by `run` command as

```
run(SimData)
```

You may save this data by

```
Sample <- run(SimData)
```

Next, model object can be specified by simModel function as

```
SimModel <- simModel(CFA.Model)
```

This program is expected to run by many SEM packages. In this version of this program, the model can be only run by `lavaan` package and it is also the default of this program. Although `lavaan` is the default package such that it will be automatically install when you installed `simsem`, please make sure that you have installed `lavaan` package in R. You may see the specification of this model object by `summary` function also. You may run the saved data by this model object by run command as

```
out <- run(SimModel, Sample)
```

The result can be summarized as

```
summary(out)
```

The simulated data was analyzed by specified CFA model. We will not go to the details of the outputs here. Finally, we need to use desired data and desired analysis model to create SSD. That is the aim of result object. We can create result object by `simResult` function as

```
Output <- simResult(SimData, SimModel, 1000)
```

The first attribute is desired data object. The second attribute is desired model object. The third attribute is number of replications. After submitting this command, the program will simulate and run 1000 data. After it is done, you can use `summary` function to see specification of this result object.

The result object contains SSD. You can find fit indices cutoff based on percentile point of SSD. For example, we wish to find 95th percentile (alpha level = .05). The `getCutoff` function can be used by

```
getCutoff(Output, 0.05)
```

The first argument is the result object. The second argument is the alpha level. You can see the SSD with cutoff in figures by

```
plotCutoff(Output, 0.05)
```

The result object is set in a specific seed number. Therefore, the SSD is expected to be the same. The `summary` of the result object can be asked by

```
summary(Output)
```

The summary has mainly two sections: fit indices cutoffs based on each alpha level and summary of parameter estimates and standard errors. For the cutoffs, not that the larger the alpha level, the more lenient the cutoffs are. For the parameter estimates and standard errors, there are seven columns provided:

1) Average of parameter estimates
2) Standard deviation of parameter estimates
3) Average of standard errors of each parameter estimate
4) The proportion of significant parameter estimates
5) Parameter values underlying simulated data
6) Average bias of parameter estimates
7) Proportion of confidence interval covered the parameter values.

Note that the columns 5-7 are not provided if users provide list of data frame instead of data object in the `simResult` function. Also, those values in columns 5-7 have different meanings when parameters are treated as random, which are shown in the example 3.

If users want the parameter estimates and standard errors of all replications only, `summaryParam` function can be used as

```
summaryParam(Output)
```

You might round the number in the `summaryParam` function by

```
round(summaryParam(Output), 3)
```

Script

The summary of the whole script in this example is

```
1     library(simsem)
2
3     loading <- matrix(0, 6, 2)
4     loading[1:3, 1] <- NA
5     loading[4:6, 2] <- NA
6     LX <- simMatrix(loading, 0.7)
7
8     latent.cor <- matrix(NA, 2, 2)
9     diag(latent.cor) <- 1
10    PH <- symMatrix(latent.cor, 0.5)
11
12    error.cor <- matrix(0, 6, 6)
13    diag(error.cor) <- 1
14    TD <- symMatrix(error.cor)
15
16    CFA.Model <- simSetCFA(LX = LX, PH = PH, TD = TD)
17    SimData <- simData(200, CFA.Model)
18    SimModel <- simModel(CFA.Model)
19    Output <- simResult(SimData, SimModel, 1000)
20    getCutoff(Output, 0.05)
21    plotCutoff(Output, 0.05)
22    summaryParam(Output)
```

Remark

1) Users may want to explicitly specify error variances and factor variances. This can be done by changing lines 15-16 to

```
error.var <- rep(NA, 6)
VTD <- simVector(error.var, 0.51)

factor.var <- rep(1, 2)
VPH <- simVector(factor.var)

CFA.Model <- simSetCFA(LX = LX, PH = PH, TD = TD, VTD = VTD, VPH = VPH)
```

where `VTD` (or `VTE`) is the vector of error variance and `VPH` (or `VPS`) is the vector of factor variance

2) Users may want to include indicators intercepts or factor intercepts (or means) by changing lines 15-16 to

```
intercept <- rep(NA, 6)
TX <- simVector(intercept, 0)

factor.mean <- rep(0, 2)
KA <- simVector(factor.mean)

CFA.Model <- simSetCFA(LX = LX, PH = PH, TD = TD, TX = TX, KA = KA)
```

where `TX` (or `TY`) is the vector of indicator intercepts and `KA` (or `AL`) is the vector of factor intercepts

3) This program can directly specify indicator variances (instead of error variances) by

```
indicator.var <- rep(NA, 6)
VX <- simVector(indicator.var, 1)

CFA.Model <- simSetCFA(LX = LX, PH = PH, TD = TD, VX = VX)
```

where `VX` (or `VY`) is the vector of indicator variances. You cannot specify error variances of indicators and overall indicators variances at the same time.

4) This program can directly specify indicator means (instead of measurement intercepts) by

```
indicator.mean <- rep(NA, 6)
MX <- simVector(indicator.mean, 0)

CFA.Model <- simSetCFA(LX = LX, PH = PH, TD = TD, MX = MX)
```

where `MX` (or `MY`) is the vector of indicator means. You cannot specify indicator intercepts and overall indicators means at the same time.

5) In the `summaryParam` function, relative bias, standardized bias, and relative bias in standard errors can be calculated by set `detail` as `TRUE`.

```
summaryParam(Output, detail=TRUE)
```

Details of how they are calculated are available in help file of the `summaryParam` function as

```
?summaryParam
```

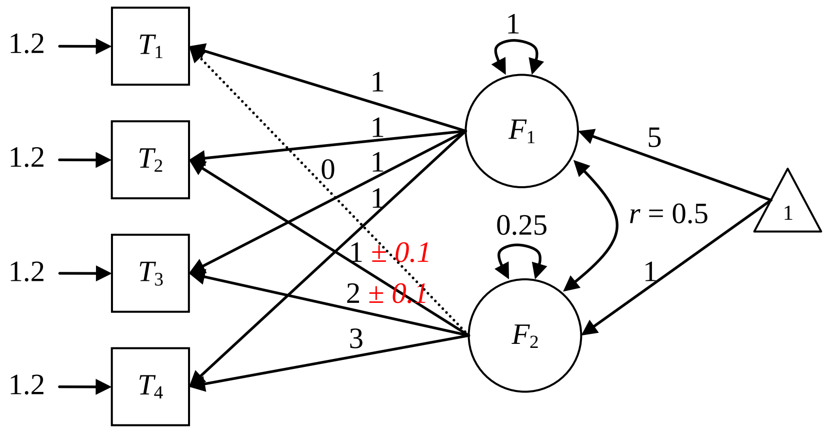
Functions Recap

Functions	Usage
`simMatrix`	Create matrix object
`symMatrix`	Create symmetric matrix object
`simSetCFA`	Create set of matrices for CFA
`simData`	Create data template in order to simulate data
`simModel`	Create analysis model
`run`	Run all objects in the `simsem` package
`summary`	Summarize all objects in the `simsem` package
`simResult`	Create result of simulation
`getCutoff`	Get the fit indices cutoff with a priori alpha level
`plotCutoff`	Plot the sampling distribution of fit indices
`summaryParam`	Summary parameter estimates and standard errors

Example 2

In this example, we will focus on a special kind of CFA: latent curve model. Latent curve model has two factors as intercept and slope. Factor loadings of the intercept factor are 1. Factor loadings of the slope factor are 0, 1, 2, and 3, representing linear change across time. In population model, intercept factor has mean of 5 and variance of 1. Slope factor has mean of 1 and variance of 0.25. Error variances are 1.2.

In this model, we will add trivially misspecification on the model. In other words, we will make population model such that specified model is still a good approximation of those population. For example, the changes may not exactly follow linear trend but the specification as linear trend is still a good approximation of population model. As shown in Figure below, we will add minor model misspecification that the factor loadings from slope factor to Time 2 and 3 deviated from 1 and 2 by ±0.1. For example, (0, 0.9, 2.05, 3) or (0, 1.05, 1.94, 3) is example of population model which is well approximated by (0, 1, 2, 3).



Factor loading matrix can be specified as

```
factor.loading <- matrix(NA, 4, 2)
factor.loading[,1] <- 1
factor.loading[,2] <- 0:3
LY <- simMatrix(factor.loading)
```

Factor variance vector can be specified as

```
factor.var <- rep(NA, 2)
factor.var.starting <- c(1, 0.25)
VPS <- simVector(factor.var, factor.var.starting)
```

Factor correlation matrix can be specified as

```
factor.cor <- matrix(NA, 2, 2)
diag(factor.cor) <- 1
PS <- symMatrix(factor.cor, 0.5)
```

Factor mean can be specified as

```
factor.mean <- rep(NA, 2)
factor.mean.starting <- c(5, 1)
AL <- simVector(factor.mean, factor.mean.starting)
```

Error variance vectors can be specified as

```
VTE <- simVector(rep(NA, 4), 1.2)
```

As you can see, the `rep` function can be put directly in parameter attribute. Next, error correlation matrix can be specified as

```
TE <- symMatrix(diag(4))
```

The `diag` function is used to create identity matrix. Its attribute means numbers of row and columns. Finally, indicator intercepts can be specified as

```
TY <- simVector(rep(0, 4))
```

CFA object that represents latent curve model can be specified as

```
LCA.Model <- simSetCFA(LY=LY, PS=PS, VPS=VPS, AL=AL, VTE=VTE, TE=TE, TY=TY)
```

where `LY` is factor loading matrix, `PS` is factor correlation matrix, `TE` is error correlation matrix, `VPS` is factor variance vector, `VTE` is error variance vector, `AL` is factor mean vector, and `TY` is measurement intercept vector.

As previous example, data, model, and result objects can be specified as

```
Data.True <- simData(300, LCA.Model)
SimModel <- simModel(LCA.Model)
Output <- simResult(Data.True, SimModel, 1000)
getCutoff(Output, 0.05)
plotCutoff(Output, 0.05)
summaryParam(Output)
```

This example uses sample size of 300 and 1000 replications. The next step is to add trivially misspecification. That is, factor loading from slope factor to time 2 and 3 varies ±0.1. Thus, the object representing variation from the target parameter is needed. That is the aim of distribution object. For this example, the uniform distribution with lower bound of -0.1 and upper bound of 0.1 is needed. This can be specified by `simUnif` function as

```
u1 <- simUnif(-0.1, 0.1)
```

The first attribute is lower bound and the second attribute is upper bound. Other distribution is also available such as normal distribution (`simNorm` function). We can use this object to random a number from this distribution by `run` function.

```
run(u1)
```

You can also use `summary` function to see specification of this object. Next, we need a factor loading matrix that contains this distribution object. Thus, the process is similar to building matrix object. The only difference is to put object name as parameter/starting value as

```
loading.trivial <- matrix(0, 4, 2)
loading.trivial[2:3, 2] <- NA
loading.mis <- simMatrix(loading.trivial, "u1")
```

Make sure that you put single or double quotation in the parameter/starting value specification. You can use `run` function to see how this matrix randomly draws numbers. Because this example has only one matrix that represents trivially misspecification, we are ready to create an object with set of misspecified matrices, called misspecified matrix object. The misspecified matrix objects are categorized based on analysis model. This example uses simMisspecCFA function to represent misspecification in CFA model by

```
LCA.Mis <- simMisspecCFA(LY = loading.mis)
```

You can use `summary` function to see the specification of this object. Let's add the trivially misspecification in the data object in `misspec` attribute as

```
Data.Mis <- simData(300, LCA.Model, misspec = LCA.Mis)
```

From here, you may run this object to create data from population with trivially misspecification. Model object is still the same object. Thus, we are ready to create new result object and see some results by

```
Output.Mis <- simResult(Data.Mis, SimModel, 1000)
getCutoff(Output.Mis, 0.05)
plotCutoff(Output.Mis, 0.05)
summaryParam(Output.Mis)
```

You can notice that the fit indices cutoff from data without trivially misspecification is a little more stringent than from data with trivially misspecification.

Script

The summary of the whole script in this example is

```
library(simsem)

factor.loading <- matrix(NA, 4, 2)
factor.loading[,1] <- 1
factor.loading[,2] <- 0:3
LY <- simMatrix(factor.loading)

factor.mean <- rep(NA, 2)
factor.mean.starting <- c(5, 1)
AL <- simVector(factor.mean, factor.mean.starting)

factor.var <- rep(NA, 2)
factor.var.starting <- c(1, 0.25)
VPS <- simVector(factor.var, factor.var.starting)

factor.cor <- matrix(NA, 2, 2)
diag(factor.cor) <- 1
PS <- symMatrix(factor.cor, 0.5)

VTE <- simVector(rep(NA, 4), 1.2)

TE <- symMatrix(diag(4))

TY <- simVector(rep(0, 4))

LCA.Model <- simSetCFA(LY=LY, PS=PS, VPS=VPS, AL=AL, VTE=VTE, TE=TE, TY=TY)

SimModel <- simModel(LCA.Model)

### Get the number sign out if you wish to run the model without misspecification
# Data.True <- simData(300, LCA.Model)
# Output <- simResult(Data.True, SimModel, 1000)
# getCutoff(Output, 0.05)
# plotCutoff(Output, 0.05)
# summaryParam(Output)

u1 <- simUnif(-0.1, 0.1)

loading.trivial <- matrix(0, 4, 2)
loading.trivial[2:3, 2] <- NA
loading.mis <- simMatrix(loading.trivial, "u1")

LCA.Mis <- simMisspecCFA(LY = loading.mis)

Data.Mis <- simData(300, LCA.Model, misspec = LCA.Mis)

Output.Mis <- simResult(Data.Mis, SimModel, 1000)
getCutoff(Output.Mis, 0.05)
plotCutoff(Output.Mis, 0.05)
summaryParam(Output.Mis)
```

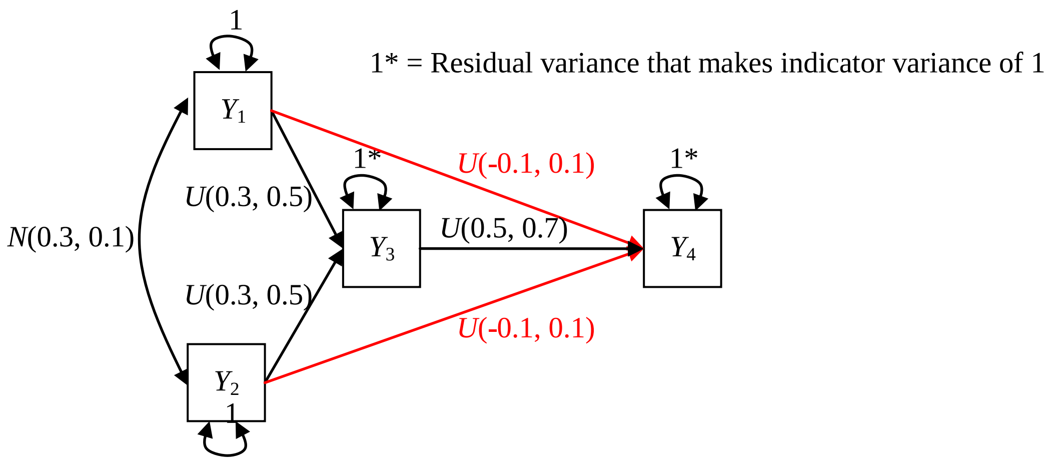
Functions Recap

Functions	Usage
simUnif	Create parameters distributed as uniform distribution
simNorm	Create parameters distributed as normal distribution
simMisspecCFA	Create set of matrices for misspecification in CFA

Example 3

This example changes analysis model to path analysis model. We will show another example of adding trivially model misspecification. In this example, population model is also random parameters.

This path analysis shows full mediation model (all black paths). However, researchers might not sure about their population parameters. Therefore, they specify parameters in range. The effects from Y_1 to Y_3 and Y_2 to Y_3 range from 0.3 to 0.5 in uniform distribution. The effect from Y_3 to Y_4 ranges from 0.5 to 0.7. The correlation between two exogenous variables ranges in normal distribution with mean of 0.3 and standard deviation of 0.1. We need all direct effect in standardized scale. Therefore, all error variances are computed such that its indicator variances equal 1. The trivially misspecification in this model is the potential direct effect from Y_1 and Y_2 to Y_4. These effects are specified in uniform distribution with lower and upper bounds of -0.1 and 0.1.



First, we need to identify distribution objects corresponding to population and misspecified models.

```
u35 <- simUnif(0.3, 0.5)
u57 <- simUnif(0.5, 0.7)
u1 <- simUnif(-0.1, 0.1)
n31 <- simNorm(0.3, 0.1)
```

The `simUnif` function is used to make uniform distribution object. The `simNorm` function is used for making normal distribution object. The first and second attributes of `simNorm` function are mean and standard deviation, respectively.

We will need only two matrices here: path matrix and indicator covariance matrix. The path matrix can be specified as

```
path.BE <- matrix(0, 4, 4)
path.BE[3, 1:2] <- NA
path.BE[4, 3] <- NA
starting.BE <- matrix("", 4, 4)
starting.BE[3, 1:2] <- "u35"
starting.BE[4, 3] <- "u57"
BE <- simMatrix(path.BE, starting.BE)
```

Similar to LISREL, the row number represents response variables and the column number represents predictors. For example, freeing (3, 2) element is to allow regression coefficient predicting Y_3 by Y_2 to be estimated. The parameter/starting values are specified in matrix and put appropriate names of objects to represent desired distributions. Note that recursive (no feedback loop) model is allowed in this program only.

The indicator covariance matrix is separate to indicator variance vector and indicator correlation matrix. First, indicator correlation can be specified as

```
residual.error <- diag(4)
residual.error[1,2] <- residual.error[2,1] <- NA
PS <- symMatrix(residual.error, "n31")
```

Only indicator correlation between Y_1 and Y_2 is estimated. For indicator variances, the default of this program is to make overall indicator variances equal to 1 and all indicator variances are estimated in path analysis.

The matrix set of path analysis object can be specified by `simSetPath` function as

```
Path.Model <- simSetPath(PS = PS, BE = BE)
```

where `PS` is indicator correlation and `BE` is matrix of regression coefficient.

Misspecification model is in regression coefficient object only. This can be specified by `simMisspecPath` function as

```
mis.path.BE <- matrix(0, 4, 4)
mis.path.BE[4, 1:2] <- NA
mis.BE <- simMatrix(mis.path.BE, "u1")
Path.Mis.Model <- simMisspecPath(BE = mis.BE)
```

Data object with trivially misspecification, model object, and result object can be created by

```
Data.Mis <- simData(500, Path.Model, misspec = Path.Mis.Model)
SimModel <- simModel(Path.Model)
Output <- simResult(Data.Mis, SimModel, 1000)
getCutoff(Output, 0.05)
plotCutoff(Output, 0.05)
summary(Output)
summaryParam(Output)
```

This example uses sample size of 500 and 1000 replications.

Note that the printout from `summary` and `summaryParam` functions provides minor different output. For the parameter estimates and standard errors, there are nine columns. The first four columns are the same meanings as previous examples. The last five columns meanings are

5) Average of random parameter values underlying simulated data across all replications
6) Standard Deviation of the random parameter values
7) Average bias of the parameter estimates to the underlying parameters of each replication.
8) Standard deviation of the bias of the parameter estimates. This average of standard errors across all replications is expected to be equal this value if random parameters are specified.
9) Proportion of confidence interval covered the parameter values underlying data in each replication.

This type of output is shown only when random parameters are specified.

Script

```
1   library(simsem)
2
3   u35 <- simUnif(0.3, 0.5)
4   u57 <- simUnif(0.5, 0.7)
5   u1 <- simUnif(-0.1, 0.1)
6   n31 <- simNorm(0.3, 0.1)
7
8   path.BE <- matrix(0, 4, 4)
9   path.BE[3, 1:2] <- NA
10  path.BE[4, 3] <- NA
11  starting.BE <- matrix("", 4, 4)
12  starting.BE[3, 1:2] <- "u35"
13  starting.BE[4, 3] <- "u57"
14  BE <- simMatrix(path.BE, starting.BE)
15
16  residual.error <- diag(4)
17  residual.error[1,2] <- residual.error[2,1] <- NA
18  PS <- symMatrix(residual.error, "n31")
19
20  Path.Model <- simSetPath(PS = PS, BE = BE)
21
22  mis.path.BE <- matrix(0, 4, 4)
23  mis.path.BE[4, 1:2] <- NA
24  mis.BE <- simMatrix(mis.path.BE, "u1")
25  Path.Mis.Model <- simMisspecPath(BE = mis.BE)
26
27  Data.Mis <- simData(500, Path.Model, misspec = Path.Mis.Model)
28  SimModel <- simModel(Path.Model)
29  Output <- simResult(Data.Mis, SimModel, 1000)
30  getCutoff(Output, 0.05)
31  plotCutoff(Output, 0.05)
32  summaryParam(Output)
```

Remark

1) This program can directly specify indicator variances by changing line 19-20 to

```
VE <- simVector(rep(NA, 4), 1)

Path.Model <- simSetPath(PS = PS, BE = BE, VE = VE)
```

where VE is the vector of indicator variances (for SEM model, it means the overall variances of factors). You cannot specify error variances (VPS) of indicators and overall indicators variances at the same time.

2) This program can directly specify indicator means (instead of measurement intercepts) by

```
ME <- simVector(rep(NA, 4), 0)

Path.Model <- simSetPath(PS = PS, BE = BE, ME = ME)
```

where ME is the vector of indicator means (for SEM model, it means the overall means of factors). You cannot specify indicator intercepts (AL) and overall indicators means at the same time.

3) This program can analyze both *X* and *Y* sides at the same time. The script in line 8-25 can be changed to

```
path.GA <- matrix(0, 2, 2)
path.GA[1, 1:2] <- NA
GA <- simMatrix(path.GA, "u35")
```

```
path.BE <- matrix(0, 2, 2)
path.BE[2, 1] <- NA
BE <- simMatrix(path.BE, "u57")

exo.cor <- matrix(NA, 2, 2)
diag(exo.cor) <- 1
PH <- symMatrix(exo.cor, "n31")

PS <- symMatrix(diag(2))

Path.Model <- simSetPath(PS = PS, BE = BE, PH = PH, GA = GA, exo=TRUE)

mis.path.GA <- matrix(0, 2, 2)
mis.path.GA[2, 1:2] <- NA
mis.GA <- simMatrix(mis.path.GA, "u1")
Path.Mis.Model <- simMisspecPath(GA = mis.GA, exo=TRUE)
```

Similar to LISREL notation, we use `GA` for the effects from exogenous indicator to endogenous indicator, `PH` for the correlations (instead of covariance) among exogenous indicators, `BE` for the directional effects among endogenous indicators, and `PS` for the correlations among endogenous residuals.

4) Users might wish to create dataset and would like to see the population values underlying the specific dataset. Then, the data output object can be created instead. This could be created by setting the `dataOnly` argument equal `FALSE`, as

```
dat <- run(Data.Mis, dataOnly = FALSE)
```

The data can be analyzed as usual by `run` command specifying the model object as the first argument and the data output object as

```
out <- run(SimModel, dat)
summary(out)
summaryParam(out)
```

Notice that the output having three additional columns: parameters underlying the data (`Param`), difference between parameters values and parameter estimates (`Bias`), and whether the confidence interval covers the parameter value (`Coverage`). Note that this will work only the parameter set using for data simulation and the parameter set of analysis model are the same.

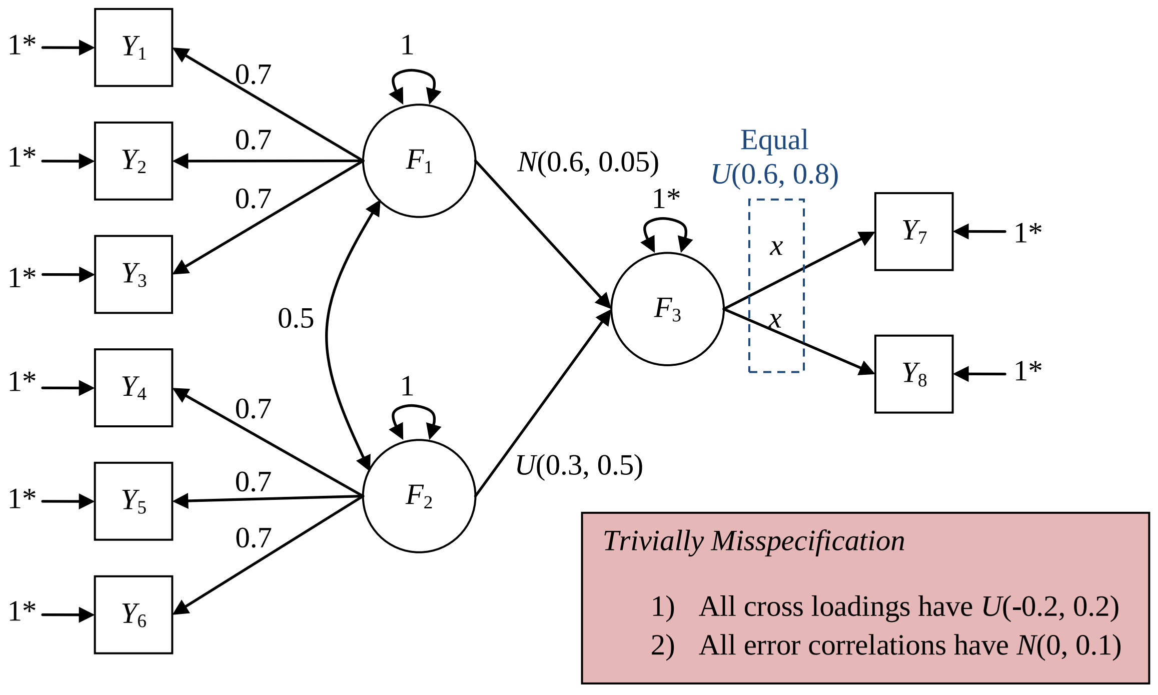
Functions Recap

Functions	Usage
`simSetPath`	Create set of matrices for path analysis
`simMisspecPath`	Create set of matrices for misspecification in path analysis

Example 4

This example will show how to analyze full SEM model with random parameters and trivially model misspecification. Furthermore, this example will illustrate how to set equality constraints. Exogenous factor specification is similar to those in Example 1. However, we will add some cross-loading trivial misspecification so we will make sure that indicator variances are still equal to 1. These two exogenous factors predict one endogenous factor. The effect from the first factor is normally distributed with mean of 0.6 and *SD* of 0.05. The effect from the second factor is uniformly distributed with lower and upper bounds of 0.3 and 0.5. The endogenous factor has two indicators. The factor loadings are constrained to be equal. The parameter of these factor loadings is random in uniform distribution with lower and upper bounds of 0.6 to 0.8. The F_3 error variance in data object is the value leading to F_3 overall variance of 1. This will make regression coefficients and factor loadings interpreted as standardized coefficients. However, the model object will fix the F_3 error variance to be 1. The analysis result will not provide standardized coefficients at first but it does not matter because we are interested only fit indices. Error variances values are computed such that indicator variances are 1.

We will have two trivially misspecification in this model. First, all possible cross loadings range in uniform distribution from -0.2 to 0.2. Second, all possible error correlations parameters range in normal distribution with mean of 0 and *SD* of 0.1.


1*
Y1
Y2
Y3
Y4
Y5
Y6
Y7
Y8
0.7
1
F1
F2
F3
0.5
N(0.6, 0.05)
U(0.3, 0.5)
Equal
U(0.6, 0.8)
x
Trivially Misspecification
1) All cross loadings have U(-0.2, 0.2)
2) All error correlations have N(0, 0.1)

1* = Residual variance that makes indicator variance of 1

First, distribution objects in this model are created as

```
n65 <- simNorm(0.6, 0.05)
u35 <- simUnif(0.3, 0.5)
u68 <- simUnif(0.6, 0.8)
u2 <- simUnif(-0.2, 0.2)
n1 <- simNorm(0, 0.1)
```

For full SEM model, if we consider only *Y* sides, four matrices are required: factor loading matrix, error covariance matrix, factor regression coefficient matrix, and factor residual covariance matrix. Factor loading matrix can be specified as

```
loading <- matrix(0, 8, 3)
loading[1:3, 1] <- NA
loading[4:6, 2] <- NA
loading[7:8, 3] <- NA
loading.start <- matrix("", 8, 3)
loading.start[1:3, 1] <- 0.7
loading.start[4:6, 2] <- 0.7
loading.start[7:8, 3] <- "u68"
```

```
LY <- simMatrix(loading, loading.start)
```

Equality constraints will be specified in making data object and model object step. At this point, the elements (7, 3) and (8, 3) are still independent. We will leave error variances to be set by the program by default (overall indicator variances = 1). Error correlation matrix is specified as

```
TE <- symMatrix(diag(8))
```

We will also leave factor error variances set by the program by default (overall factor variances = 1). Factor correlation matrix is specified as

```
factor.cor <- diag(3)
factor.cor[1, 2] <- factor.cor[2, 1] <- NA
PS <- symMatrix(factor.cor, 0.5)
```

Factor regression coefficient matrix is specified as

```
path <- matrix(0, 3, 3)
path[3, 1:2] <- NA
path.start <- matrix(0, 3, 3)
path.start[3, 1] <- "n65"
path.start[3, 2] <- "u35"
BE <- simMatrix(path, path.start)
```

All matrices are set up. The `simSetSEM` function is used to create set of matrices in SEM model as

```
SEM.model <- simSetSEM(BE=BE, LY=LY, PS=PS, TE=TE)
```

`LY` is the factor loading matrix, `TE` is the error correlation matrix, `BE` is the regression coefficient matrix, and `PS` is the factor (residual) correlation matrix. The next step is to set matrices of trivially misspecification. In this example, factor loading and error correlation matrices are needed. The set of misspecification matrices can be created by `simMisspecSEM` function as

```
loading.trivial <- matrix(NA, 8, 3)
loading.trivial[is.na(loading)] <- 0
LY.trivial <- simMatrix(loading.trivial, "u2")

error.cor.trivial <- matrix(NA, 8, 8)
diag(error.cor.trivial) <- 0
TE.trivial <- symMatrix(error.cor.trivial, "n1")

SEM.Mis.Model <- simMisspecSEM(LY = LY.trivial TE = TE.trivial)
```

Now, we will create the constraint object on two factor loadings. In single group model as in this example, a matrix is needed for each equality constraint. Number of rows in this matrix is number of parameters equated. Number of columns is two representing row and column of target matrices. Row name represent the name of target matrices. In this example, the equality constraint matrix in R should be

$$\begin{array}{c} LY \\ LY \end{array}\begin{bmatrix} 7 & 3 \\ 8 & 3 \end{bmatrix}$$

This means the element (7, 3) in LY matrix equals the element (8, 3) in LY matrix. This can be scripted in R as

```
constraint <- matrix(0, 2, 2)
constraint[1,] <- c(7, 3)
```

```
constraint[2,] <- c(8, 3)
rownames(constraint) <- rep("LY", 2)
```

Now, constraint object can be create from this object by `simEqualCon` function as

```
equal.loading <- simEqualCon(constraint, modelType="SEM")
```

The attribute in this function is to list all equality constraints first and put type of analysis in `modelType` attribute. The possible values of the `modelType` attribute are `"CFA"`, `"Path"`, `"Path.exo"`, `"SEM"`, and `"SEM.exo"`, for each type of analysis.

The next step is to create data object.

```
Data.Original <- simData(300, SEM.model)
Data.Mis <- simData(300, SEM.model, misspec=SEM.Mis.Model)
Data.Con <- simData(300, SEM.model, equalCon=equal.loading)
Data.Mis.Con <- simData(300, SEM.model, misspec=SEM.Mis.Model,
        equalCon=equal.loading)
```

Here, I list four possible combinations in data object making. Put the constraint object in the `equalCon` attribute. Sample size is specified as 300. Model object with and without equality constraints are

```
Model.Original <- simModel(SEM.model)
Model.Con <- simModel(SEM.model, equalCon=equal.loading)
```

Finally, result object can be created by any possible combinations of data and model object. I will show only the most complex combination as

```
Output <- simResult(Data.Mis.Con, Model.Con, 1000)
getCutoff(Output, 0.05)
plotCutoff(Output, 0.05)
summaryParam(Output)
```

Script

```
library(simsem)

n65 <- simNorm(0.6, 0.05)
u35 <- simUnif(0.3, 0.5)
u68 <- simUnif(0.6, 0.8)
u2 <- simUnif(-0.2, 0.2)
n1 <- simNorm(0, 0.1)

loading <- matrix(0, 8, 3)
loading[1:3, 1] <- NA
loading[4:6, 2] <- NA
loading[7:8, 3] <- NA
loading.start <- matrix("", 8, 3)
loading.start[1:3, 1] <- 0.7
loading.start[4:6, 2] <- 0.7
loading.start[7:8, 3] <- "u68"
LY <- simMatrix(loading, loading.start)

TE <- symMatrix(diag(8))

factor.cor <- diag(3)
factor.cor[1, 2] <- factor.cor[2, 1] <- NA
PS <- symMatrix(factor.cor, 0.5)

path <- matrix(0, 3, 3)
path[3, 1:2] <- NA
path.start <- matrix(0, 3, 3)
path.start[3, 1] <- "n65"
path.start[3, 2] <- "u35"
BE <- simMatrix(path, path.start)
```

```
31
32  SEM.model <- simSetSEM(BE=BE, LY=LY, PS=PS, TE=TE)
33
34  loading.trivial <- matrix(NA, 8, 3)
35  loading.trivial[is.na(loading)] <- 0
36  LY.trivial <- simMatrix(loading.trivial, "u2")
37
38  error.cor.trivial <- matrix(NA, 8, 8)
39  diag(error.cor.trivial) <- 0
40  TE.trivial <- symMatrix(error.cor.trivial, "n1")
41
42  SEM.Mis.Model <- simMisspecSEM(LY = LY.trivial, TE = TE.trivial)
43
44  constraint <- matrix(0, 2, 2)
45  constraint[1,] <- c(7, 3)
46  constraint[2,] <- c(8, 3)
47  rownames(constraint) <- rep("LY", 2)
48  equal.loading <- simEqualCon(constraint, modelType="SEM")
49
50  Data.Original <- simData(300, SEM.model)
51  Data.Mis <- simData(300, SEM.model, misspec=SEM.Mis.Model)
52  Data.Con <- simData(300, SEM.model, equalCon=equal.loading)
53  Data.Mis.Con <- simData(300, SEM.model, misspec=SEM.Mis.Model,
            equalCon=equal.loading)
54
55  Model.Original <- simModel(SEM.model)
56  Model.Con <- simModel(SEM.model, equalCon=equal.loading)
57
58  Output <- simResult(Data.Mis.Con, Model.Con, 1000)
59  getCutoff(Output, 0.05)
60  plotCutoff(Output, 0.05)
61  summaryParam(Output)
```

Remark

1) If analysts wish to constrain all factor loadings within the same factor to be equal, we need multiple constraints. The three constraints are

$$\begin{array}{c}LY\\LY\\LY\end{array}\begin{bmatrix}1&1\\2&1\\3&1\end{bmatrix}\qquad\begin{array}{c}LY\\LY\\LY\end{array}\begin{bmatrix}4&2\\5&2\\6&2\end{bmatrix}\qquad\begin{array}{c}LY\\LY\end{array}\begin{bmatrix}7&3\\8&3\end{bmatrix}$$

This can be scripted by changing lines 44-48 to be

```
constraint1 <- matrix(1, 3, 2)
constraint1[,1] <- 1:3
rownames(constraint1) <- rep("LY", 3)
constraint2 <- matrix(2, 3, 2)
constraint2[,1] <- 4:6
rownames(constraint2) <- rep("LY", 3)
constraint3 <- matrix(3, 2, 2)
constraint3[,1] <- 7:8
rownames(constraint3) <- rep("LY", 2)
equal.loading <- simEqualCon(constraint1, constraint2, constraint3, modelType="SEM")
```

The first three attributes of `simEqualCon` function is each equality constraint.

2) Analysts may wish to use both *X* and *Y* side of equations. These can be scripted by changing lines 9-48 to be

```
loading.X <- matrix(0, 6, 2)
loading.X[1:3, 1] <- NA
loading.X[4:6, 2] <- NA
LX <- simMatrix(loading.X, 0.7)

loading.Y <- matrix(NA, 2, 1)
LY <- simMatrix(loading.Y, "u68")

TD <- symMatrix(diag(6))
```

```
TE <- symMatrix(diag(2))

factor.K.cor <- matrix(NA, 2, 2)
diag(factor.K.cor) <- 1
PH <- symMatrix(factor.K.cor, 0.5)

PS <- symMatrix(as.matrix(1))

path.GA <- matrix(NA, 1, 2)
path.GA.start <- matrix(c("n65", "u35"), ncol=2)
GA <- simMatrix(path.GA, path.GA.start)

BE <- simMatrix(as.matrix(0))

SEM.model <- simSetSEM(GA=GA, BE=BE, LX=LX, LY=LY, PH=PH, PS=PS, TD=TD, TE=TE, exo=TRUE)

loading.X.trivial <- matrix(NA, 6, 2)
loading.X.trivial[is.na(loading.X)] <- 0
LX.trivial <- simMatrix(loading.X.trivial, "u2")

error.cor.X.trivial <- matrix(NA, 6, 6)
diag(error.cor.X.trivial) <- 0
TD.trivial <- symMatrix(error.cor.X.trivial, "n1")

error.cor.Y.trivial <- matrix(NA, 2, 2)
diag(error.cor.Y.trivial) <- 0
TE.trivial <- symMatrix(error.cor.Y.trivial, "n1")

TH.trivial <- simMatrix(matrix(NA, 6, 2), "n1")

SEM.Mis.Model <- simMisspecSEM(LX = LX.trivial, TE = TE.trivial, TD = TD.trivial, TH =
        TH.trivial, exo=TRUE)

constraint <- matrix(0, 2, 2)
constraint[1,] <- c(1, 1)
constraint[2,] <- c(2, 1)
rownames(constraint) <- rep("LY", 2)
equal.loading <- simEqualCon(constraint, modelType="SEM.exo")
```

`LX` is factor loading matrix of exogenous factors. `LY` is factor loading matrix of endogenous factors. `TD` is correlation matrix of measurement errors among exogenous indicators. `TE` is correlation matrix of measurement errors among endogenous indicators. `TH` is correlation matrix across measurement errors of indicators in exogenous side (representing rows) and endogenous side (representing columns). `PH` is correlation matrix among exogenous factors. `PS` is correlation matrix among residuals of endogenous factors. `GA` is regression coefficient matrix from exogenous factors to endogenous factors. `BE` is regression coefficient matrix among endogenous factors. If there is only one element in matrix (1 x 1 dimension), make sure to put `as.matrix` function on that element so that program recognize the element as matrix.

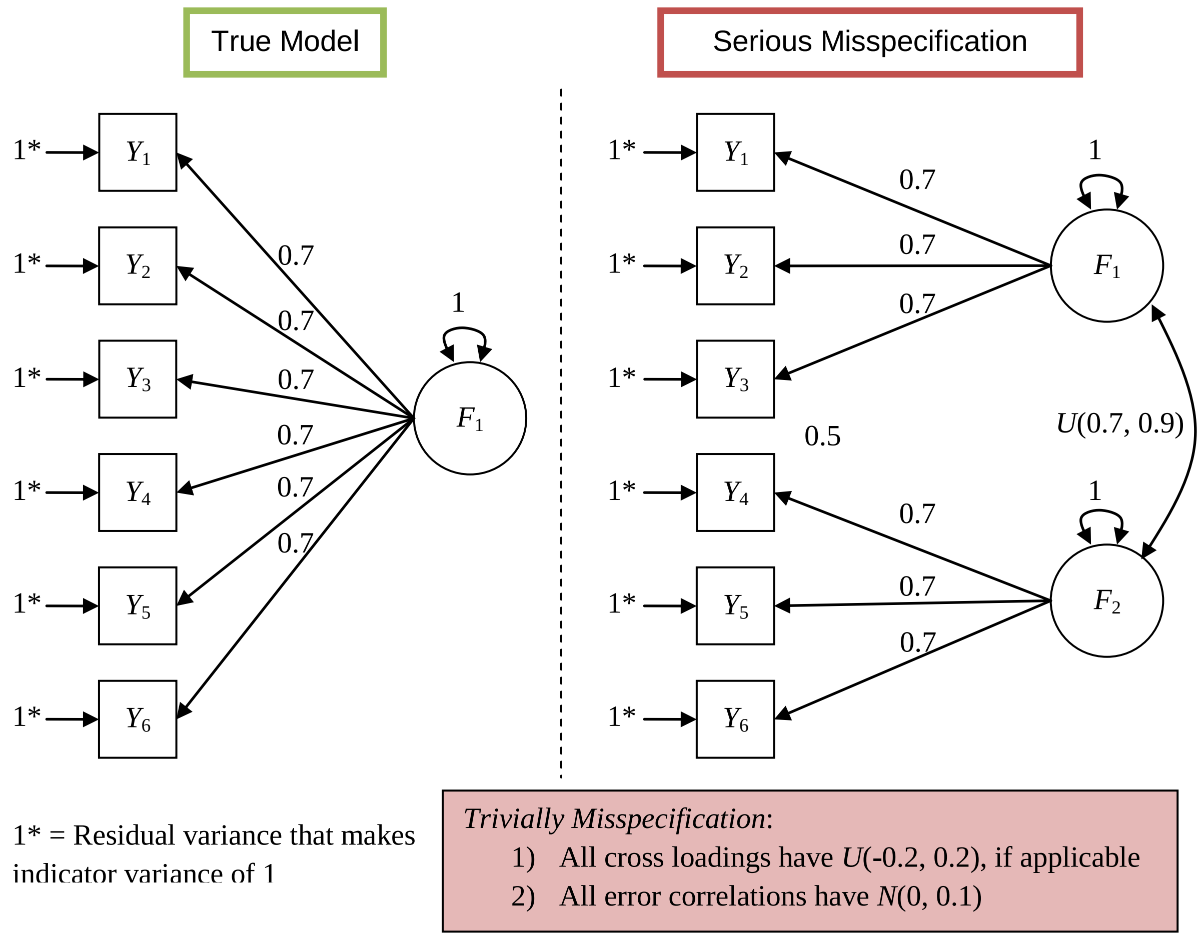
Functions Recap

Functions	Usage
`simSetSEM`	Create set of matrices for SEM
`simMisspecSEM`	Create set of matrices for misspecification in SEM
`simEqualCon`	Create list of equality constraints in the model

Example 5

The aim of tailored cutoffs is to not reject model with minor misspecification but reject the model with serious misspecification. This example will show how to build two models: true model and model with serious misspecification. Then, this example will find the proportion of data simulated from model with serious misspecification rejected by cutoffs, which are derived from sampling distribution of fit indices from true model.

The true model is one-factor model with six indicators. The factor loadings are 0.7. The error variances are calculated so that the indicator variances are 1. The serious misspecification is two-factor model with three indicators each. The factor correlation ranges from 0.7 to 0.8 in uniform distribution. We assume that the two factors are not close enough to be considered as one factor. Thus, we wish that two-factor model was rejected in high proportion (high power).


True Model
Serious Misspecification
1*
Y1
Y2
Y3
Y4
Y5
Y6
0.7
1
F1
F2
0.5
U(0.7, 0.9)
1* = Residual variance that makes indicator variance of 1
Trivially Misspecification:
1) All cross loadings have U(-0.2, 0.2), if applicable
2) All error correlations have N(0, 0.1)

Similar to Example 1, all relevant distribution objects can be specified as

```
u2 <- simUnif(-0.2, 0.2)
n1 <- simNorm(0, 0.1)
u79 <- simUnif(0.7, 0.9)
```

The true model can be specified as

```
loading.null <- matrix(0, 6, 1)
loading.null[1:6, 1] <- NA
LX.NULL <- simMatrix(loading.null, 0.7)
PH.NULL <- symMatrix(diag(1))
TD <- symMatrix(diag(6))
CFA.Model.NULL <- simSetCFA(LY = LX.NULL, PS = PH.NULL, TE = TD)
```

The misspecification of true model can be specified as

```
error.cor.mis <- matrix(NA, 6, 6)
diag(error.cor.mis) <- 1
TD.Mis <- symMatrix(error.cor.mis, "n1")
CFA.Model.NULL.Mis <- simMisspecCFA(TD.Mis)
```

The result object from the true model with trivial misspecification can be created by

```
SimData.NULL <- simData(500, CFA.Model.NULL, misspec = CFA.Model.NULL.Mis)
SimModel <- simModel(CFA.Model.NULL)
Output.NULL <- simResult(SimData.NULL, SimModel, 1000)
```

From here, you can find cutoffs or plot cutoffs. You will take a further step to create the serious misspecification model as

```
loading.alt <- matrix(0, 6, 2)
loading.alt[1:3, 1] <- NA
loading.alt[4:6, 2] <- NA
LX.ALT <- simMatrix(loading.alt, 0.7)
latent.cor.alt <- matrix(NA, 2, 2)
diag(latent.cor.alt) <- 1
PH.ALT <- symMatrix(latent.cor.alt, "u79")
CFA.Model.ALT <- simSetCFA(LY = LX.ALT, PS = PH.ALT, TE = TD)
```

All models with serious misspecification mean the models researchers wish to reject. With trivially misspecification in the serious misspecification model, this model is still seriously misspecified and we need to reject them. Thus, we will add trivially misspecification on top of the serious misspecification model to broaden the range of models we wish to reject. The misspecification part can be specified as,

```
loading.alt.mis <- matrix(NA, 6, 2)
loading.alt.mis[is.na(loading.alt)] <- 0
LX.alt.mis <- simMatrix(loading.alt.mis, "u2")
CFA.Model.alt.mis <- simMisspecCFA(LY = LX.alt.mis, TE=TD.Mis)
```

The result object from the serious misspecification on top with trivial misspecification can be created by

```
SimData.ALT <- simData(500, CFA.Model.ALT, misspec = CFA.Model.alt.mis)
Output.ALT <- simResult(SimData.ALT, SimModel, 1000)
```

Note that the same model object is used for the serious misspecification output object. Also, we expect fit indices indicating worse fit than the output object from the trivial misspecification. Then, as previous examples, we can find the fit indices cutoffs by

```
cutoff <- getCutoff(Output.NULL, 0.05)
```

Now, we save the cutoff in order to use in finding power.

We can find the proportion of samples from the serious misspecification model that was rejected by the cutoffs by `getPower` function as

```
getPower(Output.ALT, cutoff)
```

The first argument is the alternative model, the model we wish to reject. The second argument is the cutoffs.

The cutoffs can be plot as overlapping histograms by the `plotPower` function as

```
plotPower(Output.ALT, Output.NULL, 0.05)
```

The first argument is the alternative model, the model we wish to reject. The second argument is the null model, the model we wish to not reject and find the cutoffs from. The third argument is the alpha level. We may have a priori cutoffs, such as RMSEA < .05, CFI > .95, TLI > .95, and SRMR < .06. We can use these cutoffs and find the power by

```
cutoff2 <- c(RMSEA = 0.05, CFI = 0.95, TLI = 0.95, SRMR = 0.06)
getPower(Output.ALT, cutoff2)
plotPower(Output.ALT, cutoff2)
```

The `plotPower` function will plot all fit indices. If you wish to plot only some fit indices, you can use `usedFit` argument as

```
plotPower(Output.ALT, cutoff2, usedFit=c("RMSEA", "CFI"))
```

Script

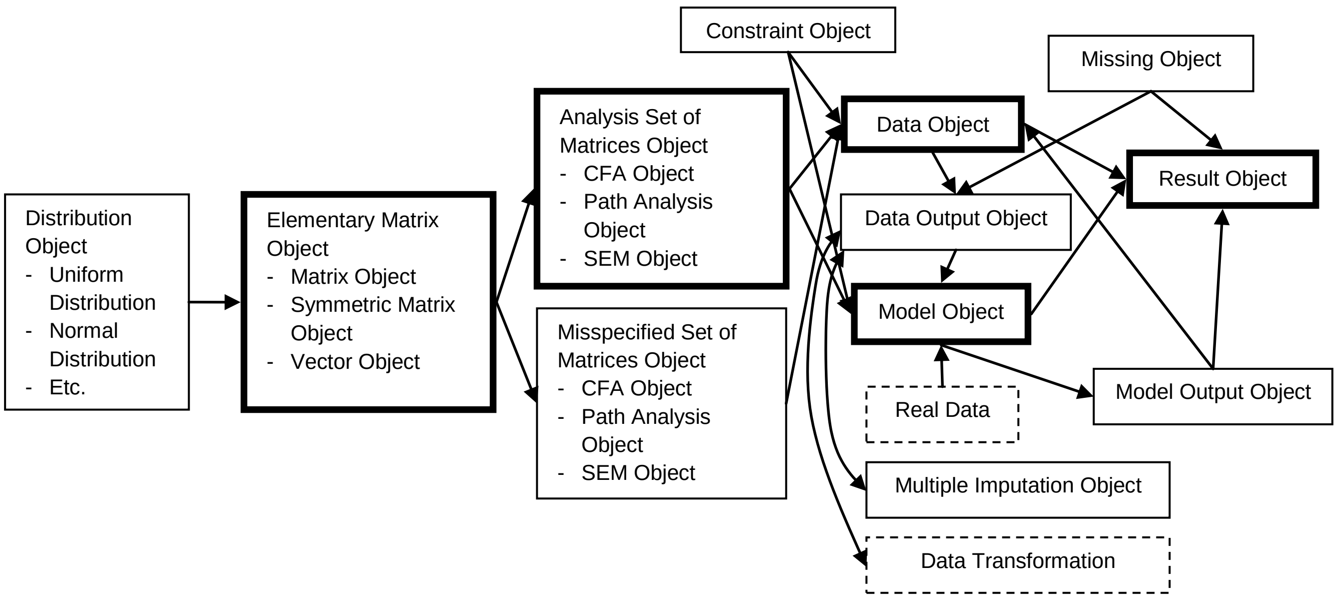
```
library(simsem)

u2 <- simUnif(-0.2, 0.2)
n1 <- simNorm(0, 0.1)
u79 <- simUnif(0.7, 0.9)

loading.null <- matrix(0, 6, 1)
loading.null[1:6, 1] <- NA
LX.NULL <- simMatrix(loading.null, 0.7)
PH.NULL <- symMatrix(diag(1))
TD <- symMatrix(diag(6))
CFA.Model.NULL <- simSetCFA(LY = LX.NULL, PS = PH.NULL, TE = TD)

error.cor.mis <- matrix(NA, 6, 6)
diag(error.cor.mis) <- 1
TD.Mis <- symMatrix(error.cor.mis, "n1")
CFA.Model.NULL.Mis <- simMisspecCFA(TE = TD.Mis)

SimData.NULL <- simData(500, CFA.Model.NULL, misspec = CFA.Model.NULL.Mis)
SimModel <- simModel(CFA.Model.NULL)
Output.NULL <- simResult(SimData.NULL, SimModel, 1000)

loading.alt <- matrix(0, 6, 2)
loading.alt[1:3, 1] <- NA
loading.alt[4:6, 2] <- NA
LX.ALT <- simMatrix(loading.alt, 0.7)
latent.cor.alt <- matrix(NA, 2, 2)
diag(latent.cor.alt) <- 1
PH.ALT <- symMatrix(latent.cor.alt, "u79")
CFA.Model.ALT <- simSetCFA(LY = LX.ALT, PS = PH.ALT, TE = TD)

loading.alt.mis <- matrix(NA, 6, 2)
loading.alt.mis[is.na(loading.alt)] <- 0
LX.alt.mis <- simMatrix(loading.alt.mis, "u2")
CFA.Model.alt.mis <- simMisspecCFA(LY = LX.alt.mis, TE=TD.Mis)

SimData.ALT <- simData(500, CFA.Model.ALT, misspec = CFA.Model.alt.mis)
Output.ALT <- simResult(SimData.ALT, SimModel, 1000)

cutoff <- getCutoff(Output.NULL, 0.05)
getPower(Output.ALT, cutoff)
plotPower(Output.ALT, Output.NULL, 0.05)

cutoff2 <- c(RMSEA = 0.05, CFI = 0.95, TLI = 0.95, SRMR = 0.06)
getPower(Output.ALT, cutoff2)
plotPower(Output.ALT, cutoff2)
```

Functions Recap

Functions	Usage
`getPower`	Get the power given cutoffs
`plotPower`	Visualize the power of rejection in sampling distribution

Summary of Model Specification

This picture shows the map of all objects and their relationships in the package. All solid border boxes indicate objects in `simsem` package. The bold border shows all objects used in the Example 1, which are required objects for simulation. This is the minimal requirement to run data simulation if you do not have real data. The dashed boxes indicate things that are not the object in this package but can interact with the package. Some objects are implemented in the near future.



Accessing Help Files

Function

You can access help file in each function by

```
?run
```

Class

You can access help file in each class by

```
class?SimMatrix
```

Methods in each class

You can access help file for a method that uses in multiple classes by

```
method?run
```

If you would like to see help file of a method that uses in a specific class by

```
method?run("SimMatrix")
```

Public Objects

Classes

Public Classes	Getting Documentation by
SimNorm	class?SimNorm
SimUnif	class?SimUnif
VirtualDist	class?VirtualDist
SimMatrix	class?SimMatrix
SymMatrix	class?SymMatrix
SimVector	class?SimVector
SimSet	class?SimSet
SimEqualCon	class?SimEqualCon
SimData	class?SimData
SimModel	class?SimModel
SimResult	class?SimResult
SimMisspec	class?SimMisspec
SimModelOut	class?SimModelOut

S4 Functions

Public Functions	Getting Documentation by	Available Classes
summary	method?summary	All classes
run	method?run	SimNorm, SimUnif, SimData, SimMatrix, SimSet, SimMisspec, SimModel, SimVector, SymMatrix
summaryShort	method?summaryShort	All classes
adjust	method?adjust	SimMatrix, SymMatrix, SimVector
simModel	method?simModel	SimSet
getCutoff	method?getCutoff	simResult
getPower	method?getPower	simResult
plotCutoff	method?plotCutoff	simResult
plotPower	method?plotPower	simResult
summaryParam	method?summaryParam	simResult

S3 Functions

Public Functions	Getting Documentation by
loadingFromAlpha	?loadingFromAlpha
simUnif	?simUnif
simNorm	?simNorm
simMatrix	?simMatrix
symMatrix	?symMatrix
simVector	?simVector
simSetCFA	?simSetCFA
simSetPath	?simSetPath
simSetSEM	?simSetSEM
simEqualCon	?simEqualCon
simData	?simData
simResult	?simResult

simMisspecCFA	?simMisspecCFA
simMisspecPath	?simMisspecPath
simMisspecSEM	?simMisspecSEM

Give Us Feedback

If you found any bugs or had any suggestions, please let us know at

Sunthud Pornprasertmanit
Center for Research Methods and Data Analysis
University of Kansas
Email: psunthud@ku.edu