# Introduction to 'simSEM' package

*Beta version 0.0.6*

Sunthud Pornprasertmanit
Patrick Miller
Alex Schoemann

*University of Kansas*

This R package has been developing for facilitating analysts to simulate and analyze data within the structural equation modeling (SEM) framework. This package aims to 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 fit 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 and on power analysis based on missing data.

1. Building simulated sampling distribution (SSD) for fit indices. This will help researchers tailor their fit indices cutoff based on a priori alpha level. We will show how to find SSD for absolute model fit with various model specification.
2. Power analysis. This package will help analysts find power in their model in both parameter estimates and fit indices. They can find the power by accounting for possible missing data. In addition, this package will allow us to estimate power based on planned missing data.

## Installing simsem package

Find the appropriate compressed file (`simsem_0.0-6.tar.gz` for both Linux and Windows).

Next, install the package by typing this line in R.

```
install.packages("simsem_0.0-6.tar.gz", repos=NULL, type="source")
```

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

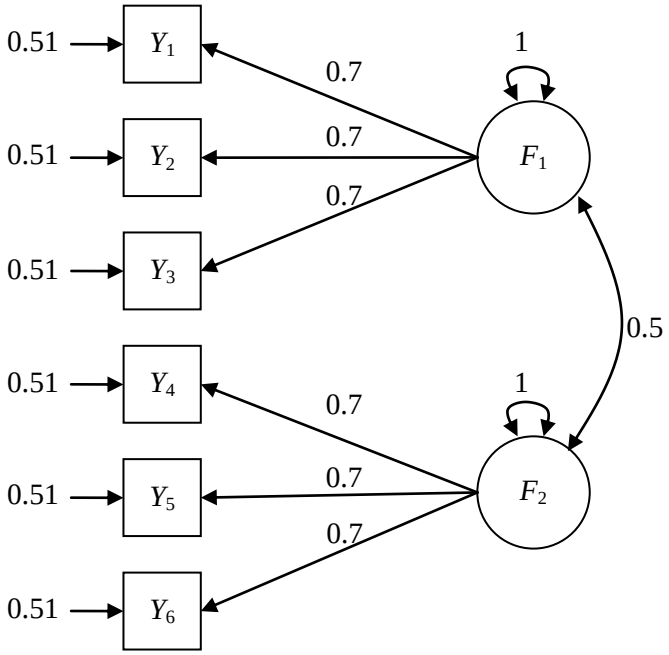
After installing the package, when you open `R` session, you can use the package by

```
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.




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

In creating a matrix, this program will call the matrix as matrix object. Matrix object has two components: parameters and starting values. In the parameters part, the 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 the free parameters. In data simulation, these parameter/starting values will be used as data generation model. The elements can be a number for a fixed parameter or a distribution object for a random parameter (which will be clarified in the 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 the 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, the factor loading can be interpreted as the standardized factor loading. The error variances by default are free parameters. From this example, the error variances are .51, which implies that indicator variances are 1 (i.e., .7×1×.7 + .51). Therefore, we will not set any error variances (or any indicator variances) and use the program default by skipping the error-variances specification and set only error correlations. There is no error correlation in this example; therefore, the error correlation is set to be identity matrix without any free parameters.

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

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

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

The symMatrix structure is similar to simMatrix. The main difference is that the symMatrix have more control on free parameters and constants such that the elements above and below the diagonal line are the same (i.e., symmetric). The parameter/starting values are not required in the symMatrix (as well as the simMatrix)command so there is only one attribute of the free parameters in this function.

The last matrix is the factor covariance matrix. Again, the 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 (i.e., fixed factor method of scale identification). Therefore, the only thing we need to specify is the factor correlation. For all correlation matrices, the diagonal elements are 1. In this model, we allow the only one element of factor correlation to be freely estimated and have the 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 an object containing the set of matrices (i.e., the factor loading matrix, the factor correlation matrix, and the error

correlation matrix). This example uses CFA; therefore, the simSetCFA function will be used. The R script will be

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

Similar to the LISREL notation, LX means the factor loading matrix, PH means the factor correlation matrix, and TD means the error correlation matrix. You may notice that PH and TD means the covariance matrices in LISREL. We use them as the correlation matrices here. This step will apply all default set-ups of this package that is to free the error variances and to fix the factor variances. This default can be seen by the summary function as

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

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

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

This set of all CFA matrices will be used to create a data-generation object (a data object) and an analysis-model object (a model object) in order to create a set of simulated data and analyze the set of simulated data later. The data and model objects do not need to have the same set of matrices (e.g., CFA). However, in this example, I will use the same set of matrices, which is the CFA model with two factors with three indicators each without any additional constraints, in both data and model objects.

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

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

The first argument is a 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 the summary function as well. From this, you are ready to simulate data by using the run command as

```
run(SimData)
```

You may save this data by

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

Next, the model object can be specified by the 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 the lavaan package, which is the default of this program. You may see the specification of this model object by the summary function also. You may run the saved data by this model object by the run function as

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

The result can be summarized by

```
summary(out)
```

The simulated data was analyzed by the specified CFA model. All fit indices, parameter estimates, standard errors, and Wald statistics will be provided in the screen. Finally, we need to use the

data object and the model object to create simulated sampling distribution. That leads to the result object. We can create the result object by the simResult function:

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

The first attribute is the number of replications. The second attribute is the desired data object. The third attribute is the desired model object. After submitting this command, the program will simulate 1000 datasets and analyze all of the datasets by the specified model.

The result object contains all fit indices values that are ready for creating the SSD. You can find a fit indices cutoff based on the percentile point of the SSD. For example, we wish to find the 95$^{th}$ 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 the cutoffs in a set of 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 seed number could be changed by adding the seed argument in the simResult function as

```
Output <- simResult(1000, SimData, SimModel, seed=751785)
```

If users who have a computer with multiple processors, this package can ask R to run with multiple processors by setting the multicore argument as TRUE:

```
 Output <- simResult(1000, SimData, SimModel, multicore=TRUE)
```

The default is to use the maximum numbers of the processors in the machine. The users can specify their desired number of processors by adding the numProc argument as

```
Output <- simResult(1000, SimData, SimModel, multicore=TRUE, numProc=2)
```

The summary of the result object can be asked by

```
summary(Output)
```

The summary on the screen has mainly two sections: the fit indices cutoffs based on each alpha level and the 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 the standard errors, there are seven columns provided:

1) Estimate.Average: Average of parameter estimates
2) Esitimate.SD: Standard deviation of parameter estimates
3) Average.SE: Average of standard errors of each parameter estimate
4) Power: The proportion of significant parameter estimates
5) Average.Param: Parameter values underlying simulated data
6) Average.Bias: Average bias of parameter estimates
7) Coverage: Proportion of confidence interval covered the parameter values.

Note that the columns 5-7 are not provided if users provide a list of data frame instead of data object in the simResult function (putting in the function by replacing the SimData). 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 the standard errors of all replications only, the 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(1000, SimData, SimModel)
20  getCutoff(Output, 0.05)
21  plotCutoff(Output, 0.05)
22  summaryParam(Output)
```

## Remark

1) Users may want to explicitly specify the error variances and the 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 the error variance and VPH (or VPS) is the vector of the factor variance

2) Users may want to include the indicators intercepts or the 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 the indicator intercepts and KA (or AL) is the vector of the factor intercepts

3) This program can directly specify the indicator variances (instead of the 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 the indicator variances. You cannot specify the error variances of indicators and the 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 the indicator intercepts and the 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
```

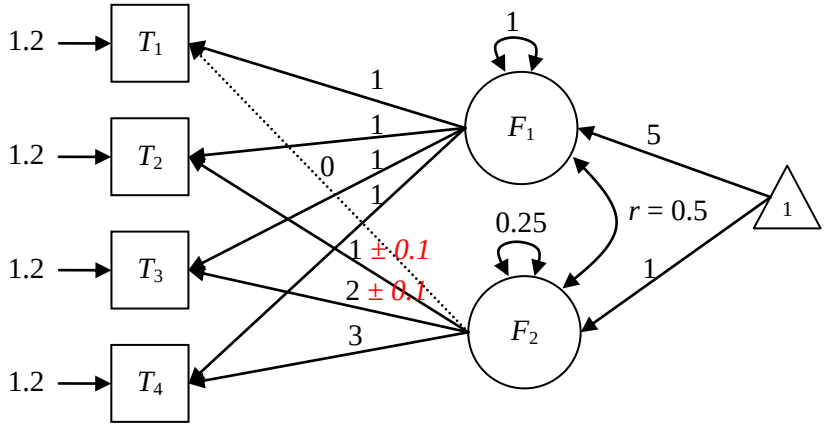
### Functions Recap

| 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: growth curve model. The growth curve model in this example specifies two factors as intercept and slope. The factor loadings of the intercept factor are all 1. The factor loadings of the slope factor are 0, 1, 2, and 3, representing the linear change across time. In the population model, the intercept factor has the mean of 5 and the variance of 1. The slope factor has the mean of 1 and the variance of 0.25. All error variances are 1.2.

In this model, we will add a trivially misspecification in the model. In other words, we will specify a model such that the specified model is still a good approximation of the desired population. For example, the changes in the population model may not exactly follow a linear trend but the specification as a linear trend is still a good approximation of the population model. As shown in the Figure shown below, we will add a minor model misspecification that the factor loadings from the slope factor to the indicators representing 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) are the examples of the population model which is well approximated by (0, 1, 2, 3).




The 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)
```

The 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)
```

The factor correlation matrix can be specified as

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

The factor mean vector can be specified as

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

The error variance vector can be specified as

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

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

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

The diag function creates an identity matrix. The attribute of the diag function means the number of row and columns in the identity matrix. Finally, the indicator intercepts vector can be specified as

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

The CFA object that represents the growth 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 the factor loading matrix, PS is the factor correlation matrix, TE is the error correlation matrix, VPS is the factor variance vector, VTE is the error variance vector, AL is the factor mean vector, and TY is the measurement intercept vector.

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

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

This example uses sample size of 300 and uses 1000 replications. The next step is to add a trivially misspecification. That is, the factor loadings from the slope factor to the indicators representing Times 2 and 3 vary ±0.1. Thus, we need to make an object representing the variation, i.e., ±0.1. This object is called a 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 the simUnif function as

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

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

```
run(u1)
```

You can also use summary function to see specification of this object. Next, we need to put the distribution object into appropriate positions in the model. We need to put the uniform distribution object into the factor loadings from the slope factors to the indicators representing Times 2 and 3. Therefore, we need to create a factor loading matrix and put the distribution object into the factor loading matrix. Thus, the process is similar to building the simMatrix object. The only difference is to put the object name as the parameter/starting values 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 the run function to see how this matrix randomly draws numbers from the specified distribution. Because this example has the trivially misspecification in only factor loadigns, we are ready to create an object with the set of the matrices containing model misspecification, called a misspecified set object. The function name to create the misspecified set object depends on an analysis model. This example uses a simMisspecCFA function to represent the 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)
```

The parameters from the misspecification set will added on top of the real parameters and then data will be created based on the combined parameters. You may use the run function on this object to create data from the population with trivially misspecification. We retain the same analysis model; therefore, we do need to change the model object. Finally, we are ready to create a new result object and examine the results of the simulation by

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

You may notice that the fit indices cutoff from the simulation result without the trivially misspecification is a little more stringent than from the simulation result with the trivially misspecification.

## Script

The summary of the whole script in this example is

```
1   library(simsem)
2
3   factor.loading <- matrix(NA, 4, 2)
4   factor.loading[,1] <- 1
5   factor.loading[,2] <- 0:3
6   LY <- simMatrix(factor.loading)
7
8   factor.mean <- rep(NA, 2)
9   factor.mean.starting <- c(5, 1)
10  AL <- simVector(factor.mean, factor.mean.starting)
11
12  factor.var <- rep(NA, 2)
13  factor.var.starting <- c(1, 0.25)
14  VPS <- simVector(factor.var, factor.var.starting)
15
16  factor.cor <- matrix(NA, 2, 2)
17  diag(factor.cor) <- 1
18  PS <- symMatrix(factor.cor, 0.5)
19
20  VTE <- simVector(rep(NA, 4), 1.2)
21
22  TE <- symMatrix(diag(4))
23
24  TY <- simVector(rep(0, 4))
25
26  LCA.Model <- simSetCFA(LY=LY, PS=PS, VPS=VPS, AL=AL, VTE=VTE, TE=TE, TY=TY)
27
28  SimModel <- simModel(LCA.Model)
29
30  ### Get the number sign out if you wish to run the model without misspecification
31  # Data.True <- simData(300, LCA.Model)
32  # Output <- simResult(1000, Data.True, SimModel)
33  # getCutoff(Output, 0.05)
34  # plotCutoff(Output, 0.05)
35  # summaryParam(Output)
36
37  u1 <- simUnif(-0.1, 0.1)
38
39  loading.trivial <- matrix(0, 4, 2)
40  loading.trivial[2:3, 2] <- NA
41  loading.mis <- simMatrix(loading.trivial, "u1")
```

```
42
43  LCA.Mis <- simMisspecCFA(LY = loading.mis)
44
45  Data.Mis <- simData(300, LCA.Model, misspec = LCA.Mis)
46
47  Output.Mis <- simResult(1000, Data.Mis, SimModel)
48  getCutoff(Output.Mis, 0.05)
49  plotCutoff(Output.Mis, 0.05)
50  summaryParam(Output.Mis)
```

### Functions Recap

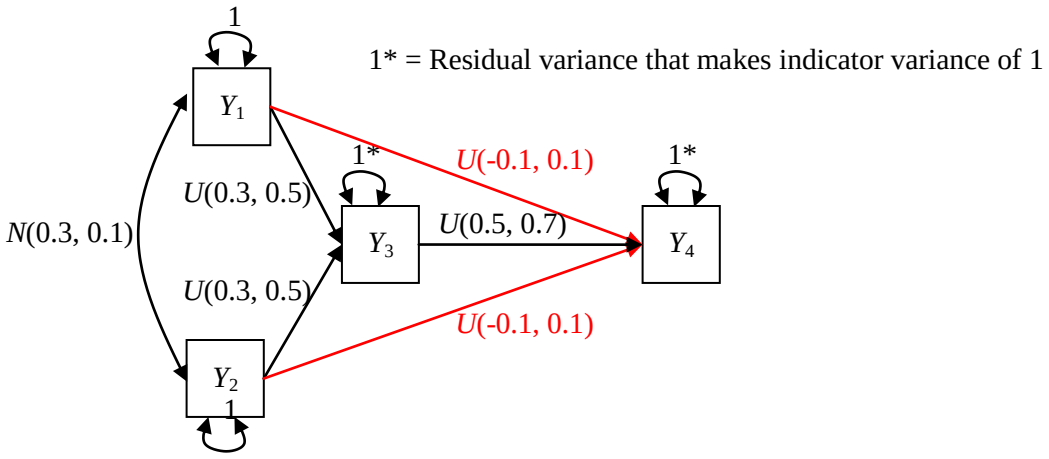
| Functions | Usage |
|---|---|
| simUnif | Create parameters distributed as uniform distribution |
| simNorm | Create parameters distributed as normal distribution |
| simMisspecCFA | Create set of matrices for misspecification in CFA |

## Example 3

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

The hypothesized model is a full mediation model (all black paths). Users might be not sure about the exact values of the parameters in the model. Therefore, they specify parameters in ranges to represent the users' uncertainty. The effects from $Y_1$ to $Y_3$ and $Y_2$ to $Y_3$ range from 0.3 to 0.5 in a 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 a normal distribution with the mean of 0.3 and the standard deviation of 0.1. We need all direct effects in standardized scale. Therefore, all error variances are computed such that their indicator variances equal 1.

In the population model, the mediator does not explain all effects from the independent variables to the dependent variable. The trivially misspecification in this model is the potential direct effects from $Y_1$ and $Y_2$ to $Y_4$. These effects are specified in a uniform distribution with the lower and upper bounds of -0.1 and 0.1.




First, we need to identify the distribution objects corresponding to the population and misspecification 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 a uniform distribution object. The simNorm function is used to make a normal distribution object. The first and second arguments of the simUnif function are the lower and the upper bounds, respectively. The first and second arguments of the simNorm function are the mean and the standard deviation, respectively.

We will need only two matrices in this model: a path matrix and an 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 the (3, 2) element is to estimate the regression coefficient from $Y_2$ to $Y_3$. To put random parameters, the appropriate names of the random parameter object should be set in the appropriate positions in the parameter/starting values matrix. Note that a nonrecursive (with feedback loop) model is not allowed in this program.

The indicator covariance matrix separates into the indicator variance vector and the indicator correlation (or residual correlation) matrix. First, the indicator correlation can be specified as

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

In this example, only indicator correlation between $Y_1$ and $Y_2$ is estimated. For the indicator variances, the default of this program is to make the overall indicator variances equal to 1 and all indicator variances are estimated in a path analysis model.

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

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

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

The misspecification model in this example is in the regression coefficients only. This can be specified by the 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)
```

Notice that the "u1" object (i.e., the uniform distribution object ranging from -0.1 to 0.1) is put in the elements (4, 1) and (4, 2) of the regression coefficient matrix to represent the misspecified direct effects.

The data object with trivially misspecification, the model object, and the result object can be created by

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

This example uses sample size of 500 and uses 1000 replications.

Note that the printout from the `summary` and `summaryParam` functions provides a slightly different output. There are nine columns in the parameter estimates and standard errors section. The first four columns are the same meanings as previous examples. The last five columns meanings are

5) `Average.Param`: The average of random parameter values underlying the simulated data across all replications
6) `SD.Param`: The standard deviation of the random parameter values
7) `Average.Bias`: The average bias of the parameter estimates from the random parameters of each replication.
8) `SD.Bias`: The standard deviation of the bias of all parameter estimates. This value is expected to be equal to the average of standard errors across all replications if random parameters are specified.
9) `Coverage`: Proportion of confidence interval covered the random parameter values underlying data in each replication.

This printout 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(1000, Data.Mis, SimModel)
30  getCutoff(Output, 0.05)
31  plotCutoff(Output, 0.05)
32  summaryParam(Output)
```

### Remark

1) We can directly specify the indicator variances by changing Lines 19-20 to

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

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

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

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

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

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

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

3) This program can analyze both *X* and *Y* sides at the same time. The script in Lines 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 indicators to endogenous indicators, 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 a 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 the run command with the model object as the first argument and the data output object as the second argument

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

```
summaryParam(out)
```

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

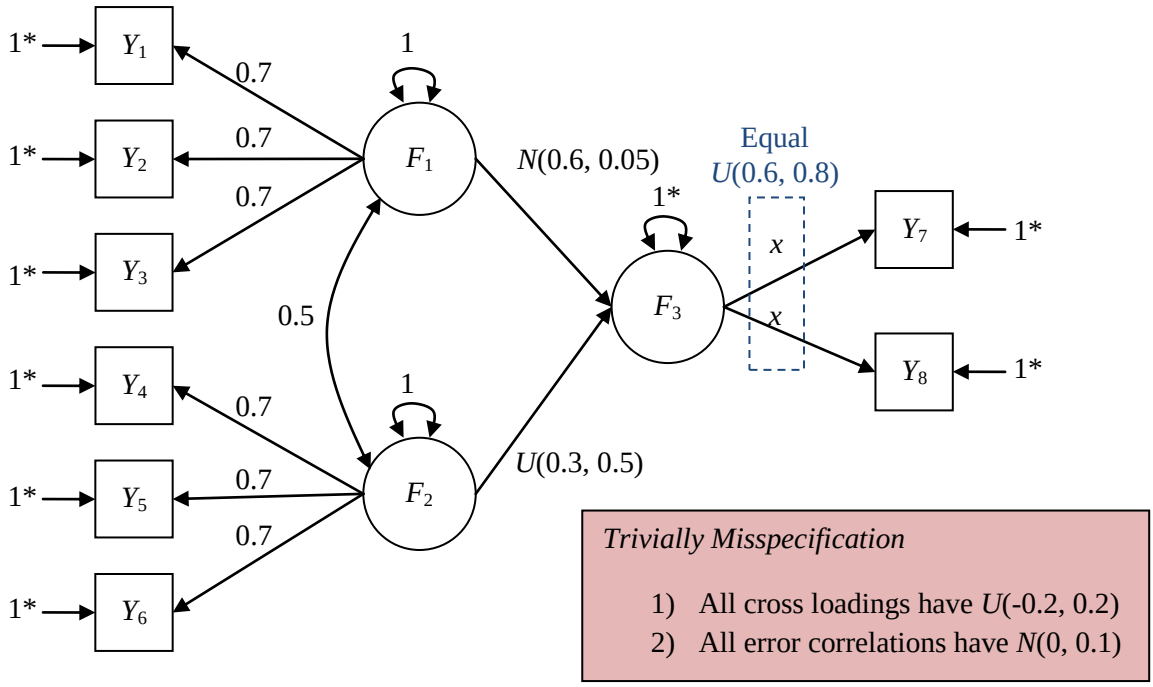
### Functions Recap

| 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 specify full SEM model with random parameters and a trivially model misspecification. Furthermore, this example will illustrate how to set an equality constraint. The specification of the factors and indicators in the exogenous side is similar to the syntax provided in the Example 1. In this example, we will add trivial cross-loadings as a trivial misspecification. We still need to make sure that the 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 the mean of 0.6 and the standard deviation of 0.05. The effect from the second factor is uniformly distributed with the lower and the upper bounds of 0.3 and 0.5. The endogenous factor has two indicators. The factor loadings are equally constrained. The parameter of the endogenous factor loadings is uniformly distributed with the lower and the upper bounds of 0.6 to 0.8. The $F_3$ error variance is the value that makes the $F_3$ overall variance equal to 1. As a result, the regression coefficients and the factor loadings can be interpreted as standardized coefficients. However, the fixing overall variances as 1 can be done only in data generation. The analysis model will fix the $F_3$ error variance as 1 instead (not overall variance). Therefore, the analysis result will not provide the standardized coefficients.

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




1* = Residual variance that makes indicator variance of 1

First, the 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 a full SEM model, if we consider only *Y* side, four matrices are required: the factor loading matrix, the error covariance matrix, the factor regression coefficient matrix, and the factor residual covariance matrix. The 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)
```

If we run the `LY` object, we will see that the loadings of the endogenous indicators are not equal. We will make them equal leter. We will leave the error variances by default (overall indicator variances = 1). The error correlation matrix is specified as

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

We will also leave the factor error variances set by default (overall factor variances = 1). The 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)
```

The 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)
```

Now, all matrices are set up. The simSetSEM function will be used to create the set of matrices in the 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 the matrices in the trivial model misspecification. In this example, the factor loading and the error correlation matrices are needed. The set of misspecification matrices can be created by the 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) <- 1
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 a single group model as in this example, a matrix is needed for each equality constraint. The number of rows in this matrix is the number of constrained parameters in each set of equality constraint. The number of columns is two representing the row and the column of the target matrices. The row name represents the name of the target matrices. In this example, the equality constraint matrix should be

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

This means that the element (7, 3) in LY matrix equals the element (8, 3) in LY matrix. The syntax will be

```
constraint <- matrix(0, 2, 2)
constraint[1,] <- c(7, 3)
constraint[2,] <- c(8, 3)
rownames(constraint) <- rep("LY", 2)
```

Now, the constraint object can be created from this matrix by the simEqualCon function as

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

The argument in this function is to list all equality constraints first and put the type of analysis in the modelType argument as the last argument. 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 a 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 is the list of four possible combinations to make a data object. We can put the constraint object in the equalCon argument. In this example, the sample size is specified as 300. The model objects with and without equality constraints are

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

Finally, the result object can be created by any possible combinations of the data and the model objects. I will show only the most complex combination (the data object with the trivial misspecification and the equality constraint combined with the model object with the equality constraint) as

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

## Script

```
1   library(simsem)
2
3   n65 <- simNorm(0.6, 0.05)
4   u35 <- simUnif(0.3, 0.5)
5   u68 <- simUnif(0.6, 0.8)
6   u2 <- simUnif(-0.2, 0.2)
7   n1 <- simNorm(0, 0.1)
8
9   loading <- matrix(0, 8, 3)
10  loading[1:3, 1] <- NA
11  loading[4:6, 2] <- NA
12  loading[7:8, 3] <- NA
13  loading.start <- matrix("", 8, 3)
14  loading.start[1:3, 1] <- 0.7
15  loading.start[4:6, 2] <- 0.7
16  loading.start[7:8, 3] <- "u68"
17  LY <- simMatrix(loading, loading.start)
18
19  TE <- symMatrix(diag(8))
20
21  factor.cor <- diag(3)
22  factor.cor[1, 2] <- factor.cor[2, 1] <- NA
23  PS <- symMatrix(factor.cor, 0.5)
24
25  path <- matrix(0, 3, 3)
26  path[3, 1:2] <- NA
27  path.start <- matrix(0, 3, 3)
28  path.start[3, 1] <- "n65"
29  path.start[3, 2] <- "u35"
30  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) <- 1
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(1000, Data.Mis.Con, Model.Con)
59  getCutoff(Output, 0.05)
60  plotCutoff(Output, 0.05)
61  summaryParam(Output)
```

### Remark

1) If users wish to constrain all factor loadings within a same factor to be equal, we need multiple constraints. All three constraints are

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

To make the syntax, Lines 44-48 can be changed as

```
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 arguments of the `simEqualCon` function is each equality constraint.

2) Users may wish to use both *X* and *Y* sides by changing Lines 9-48 as

```
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) <- 1
TD.trivial <- symMatrix(error.cor.X.trivial, "n1")

error.cor.Y.trivial <- matrix(NA, 2, 2)
diag(error.cor.Y.trivial) <- 1
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 the factor loading matrix of the exogenous factors. LY is the factor loading matrix of the endogenous factors. TD is the correlation matrix of the measurement errors among exogenous indicators. TE is the correlation matrix of the measurement errors among endogenous indicators. TH is the correlation matrix across the measurement errors of indicators in both exogenous side (representing rows) and endogenous side (representing columns). PH is the correlation matrix among the exogenous factors. PS is correlation matrix among residuals of endogenous factors. GA is the regression coefficient matrix from exogenous factors to endogenous factors. BE is the regression coefficient matrix among endogenous factors. If there is only one element in a matrix (1 x 1 dimension), make sure to put the as.matrix function on that element so that the program recognizes the element as a matrix.

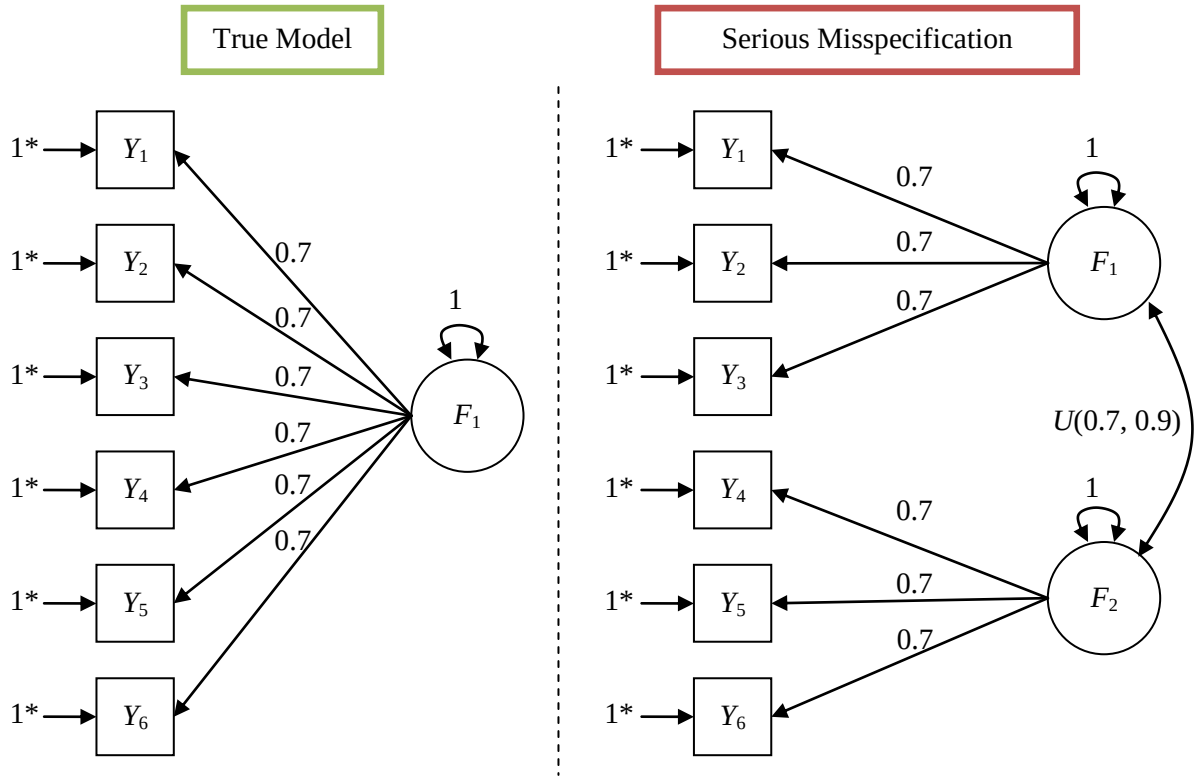
### Functions Recap

| 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

All previous examples have shown how to find a cutoff in order to discriminate between trivial misspecification and severe misspecification using SSD. This example will show how to build two models: a correct population that users do not wish to reject and another population that users wish to reject. A cutoff is created from a correct population with a trivial model misspecification. Then, the data is created from the other population that users wish to reject. Then, we will find the proportion of data simulated from model with serious misspecification rejected by the cutoffs (i.e., statistical power).

In this example, the correct population is the one-factor model with six indicators. All factor loadings are 0.7. All error variances are calculated so that all indicator variances are 1. The trivial misspecification of the correct model includes all possible small cross-loadings and all possible small error correlations. The other population model is a two-factor model with three indicators each. The factor correlation of the other model ranges from 0.7 to 0.8 in a uniform distribution. We assume that the two factors are not close enough to be considered as one factor that we wish to reject. Thus, we hope that the data from the two-factor model were rejected in a high proportion (high power).




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)

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 correct population 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 the correct population 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 correct population model with trivial misspecification can be specified as

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

From here, we can find cutoffs or plot cutoffs of the correct population model. You will take a further step to create the other 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)
```

We wish to reject this model. We can add a trivial misspecification in this model; however, we still wish to reject this model. We will add trivial misspecification on top of this 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 other model with trivial misspecification can be created by

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

Note that the same model object is used and we wish that the result of the analysis will provide a bad fit index. We expect the fit indices obtained from the data from the other model indicating worse fit than the fit indices from the correct model with the trivial misspecification. Then, as previous examples, we can find the fit indices cutoffs from the correct model by

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

Now, we save the cutoff in order to find power.

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

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

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

The cutoffs can be plot on a figure of overlapping histograms from the samples from both populations by the `plotPower` function as

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

The first argument is the alternative model or the model we wish to reject. The second argument is the null model or the model we wish to not reject and find the cutoffs from. The third argument is the alpha level. We may set a priori cutoffs, such as RMSEA < .05, CFI > .95, TLI > .95, and SRMR < .06, and use these cutoffs to 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 of fit indices, you can use a `usedFit` argument as

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

### Script

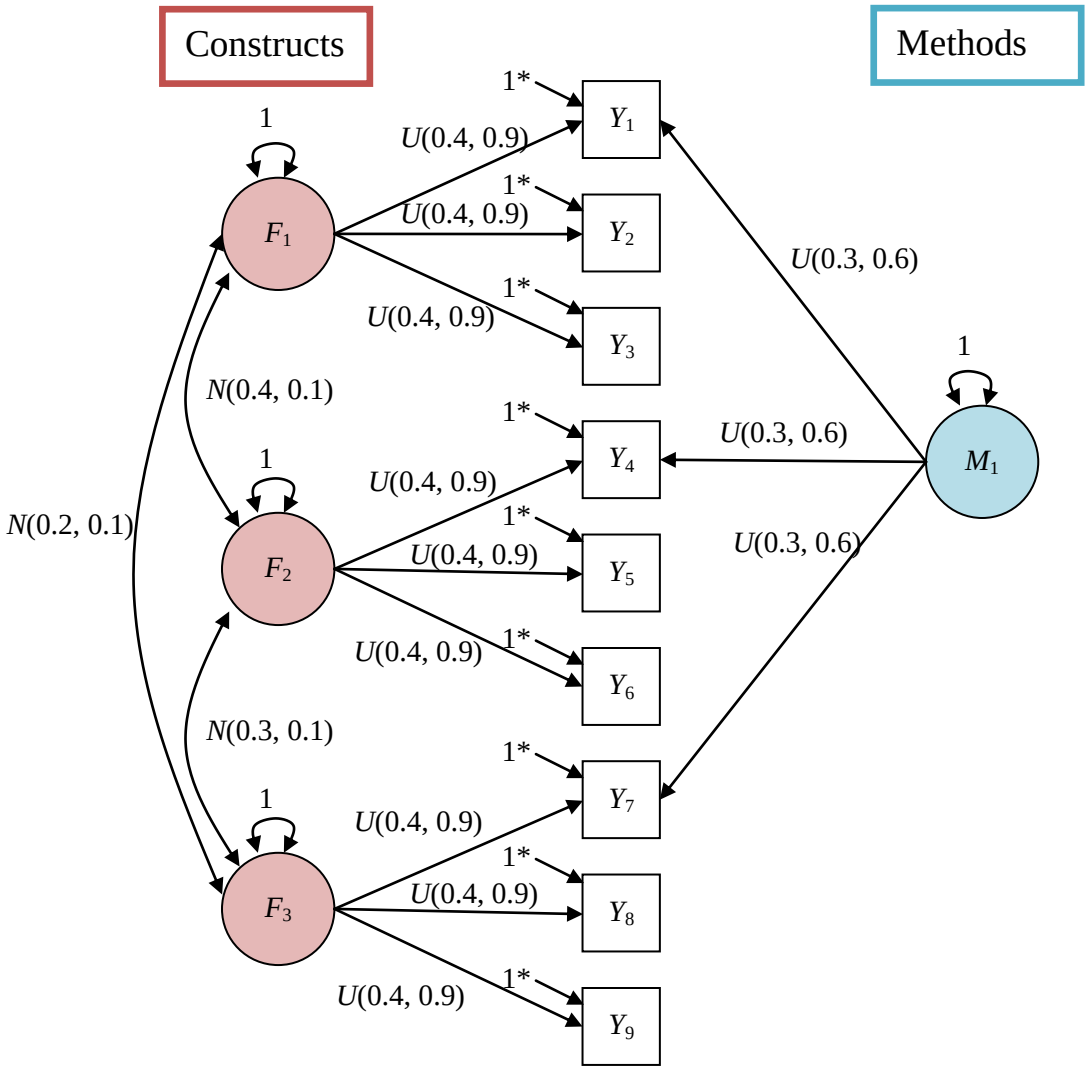
```
1   library(simsem)
2
3   u2 <- simUnif(-0.2, 0.2)
4   n1 <- simNorm(0, 0.1)
5   u79 <- simUnif(0.7, 0.9)
6
7   loading.null <- matrix(0, 6, 1)
8   loading.null[1:6, 1] <- NA
9   LX.NULL <- simMatrix(loading.null, 0.7)
10  PH.NULL <- symMatrix(diag(1))
11  TD <- symMatrix(diag(6))
12  CFA.Model.NULL <- simSetCFA(LY = LX.NULL, PS = PH.NULL, TE = TD)
13
14  error.cor.mis <- matrix(NA, 6, 6)
15  diag(error.cor.mis) <- 1
16  TD.Mis <- symMatrix(error.cor.mis, "n1")
17  CFA.Model.NULL.Mis <- simMisspecCFA(TE = TD.Mis)
18
19  SimData.NULL <- simData(500, CFA.Model.NULL, misspec = CFA.Model.NULL.Mis)
20  SimModel <- simModel(CFA.Model.NULL)
21  Output.NULL <- simResult(1000, SimData.NULL, SimModel)
22
23  loading.alt <- matrix(0, 6, 2)
24  loading.alt[1:3, 1] <- NA
25  loading.alt[4:6, 2] <- NA
26  LX.ALT <- simMatrix(loading.alt, 0.7)
27  latent.cor.alt <- matrix(NA, 2, 2)
28  diag(latent.cor.alt) <- 1
29  PH.ALT <- symMatrix(latent.cor.alt, "u79")
30  CFA.Model.ALT <- simSetCFA(LY = LX.ALT, PS = PH.ALT, TE = TD)
31
32  loading.alt.mis <- matrix(NA, 6, 2)
33  loading.alt.mis[is.na(loading.alt)] <- 0
34  LX.alt.mis <- simMatrix(loading.alt.mis, "u2")
35  CFA.Model.alt.mis <- simMisspecCFA(LY = LX.alt.mis, TE=TD.Mis)
36
37  SimData.ALT <- simData(500, CFA.Model.ALT, misspec = CFA.Model.alt.mis)
38  Output.ALT <- simResult(1000, SimData.ALT, SimModel)
39
40  cutoff <- getCutoff(Output.NULL, 0.05)
41  getPower(Output.ALT, cutoff)
42  plotPower(Output.ALT, Output.NULL, 0.05)
43
44  cutoff2 <- c(RMSEA = 0.05, CFI = 0.95, TLI = 0.95, SRMR = 0.06)
45  getPower(Output.ALT, cutoff2)
46  plotPower(Output.ALT, cutoff2)
```

### Functions Recap

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

## Example 6

This example will show how to impose missing values into datasets. The model of this example is the Multi-Trait, Multi-Method (MTMM) model. There are three traits in this model. $Y_1$, $Y_4$, and $Y_7$ are measured by a common method. The parameter models are shown below. The trivial model misspecification is specified in cross-loadings and error correlations. Note that the cross-loadings in the construct side are only made because the cross-loadings in the method side do not make sense. We are expected that the percentage of missing data will be approximately 20% in all variables.




1* = Residual variance that makes indicator variance of 1

*Trivially Misspecification*:

1) All cross loadings have $U(-0.2, 0.2)$ only in the construct side.
2) All error correlations have $N(0, 0.1)$

All relevant distribution objects can be specified as

```
u2 <- simUnif(-0.2, 0.2)
u49 <- simUnif(0.4, 0.9)
u36 <- simUnif(0.3, 0.6)
n1 <- simNorm(0, 0.1)
n21 <- simNorm(0.2, 0.1)
n31 <- simNorm(0.3, 0.1)
n41 <- simNorm(0.4, 0.1)
```

The factor loading matrix can be specified as

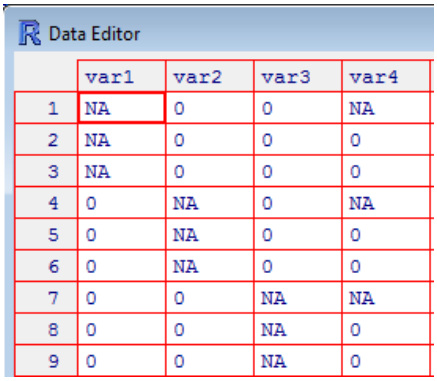
```
loading <- matrix(0, 9, 4)
loading[1:3, 1] <- NA
loading[4:6, 2] <- NA
```

```
loading[7:9, 3] <- NA
loading[c(1, 4, 7), 4] <- NA
loading.v <- matrix(0, 9, 4)
loading.v[1:3, 1] <- "u49"
loading.v[4:6, 2] <- "u49"
loading.v[7:9, 3] <- "u49"
loading.v[c(1, 4, 7), 4] <- "u36"
LY <- simMatrix(loading, loading.v)
```

For some users, type in values in a matrix might be easier. You might consider the data.entry function.

```
loading <- matrix(0, 9, 4)
data.entry(loading)
```

Then, users can edit each element of the loading matrix. The picture of the loading matrix should be




| | var1 | var2 | var3 | var4 |
|---|---|---|---|---|
| 1 | NA | 0 | 0 | NA |
| 2 | NA | 0 | 0 | 0 |
| 3 | NA | 0 | 0 | 0 |
| 4 | 0 | NA | 0 | NA |
| 5 | 0 | NA | 0 | 0 |
| 6 | 0 | NA | 0 | 0 |
| 7 | 0 | 0 | NA | NA |
| 8 | 0 | 0 | NA | 0 |
| 9 | 0 | 0 | NA | 0 |

The syntax of the factor correlation matrix is

```
faccor <- diag(4)
faccor[1, 2] <- faccor[2, 1] <- NA
faccor[1, 3] <- faccor[3, 1] <- NA
faccor[2, 3] <- faccor[3, 2] <- NA
faccor.v <- diag(4)
faccor.v[1, 2] <- faccor.v[2, 1] <- "n41"
faccor.v[1, 3] <- faccor.v[3, 1] <- "n21"
faccor.v[2, 3] <- faccor.v[3, 2] <- "n31"
PS <- symMatrix(faccor, faccor.v)
```

In this example, the transpose function is used to not put the values twice. The semi-colon is used to save space. Users may use separate lines instead of the semi-colons. The factor variances are set as 1 by the program default. There is no correlation among measurement errors. The error correlation matrix can be specified as

```
TE <- symMatrix(diag(9))
```

Thus, the MTMM model can be set up as

```
mtmm.model <- simSetCFA(LY=LY, PS=PS, TE=TE)
```

The trivial model misspecification can be specified as

```
error.cor.mis <- matrix(NA, 9, 9)
diag(error.cor.mis) <- 1
TE.mis <- symMatrix(error.cor.mis, "n1")
loading.mis <- matrix(NA, 9, 4)
loading.mis[is.na(loading)] <- 0
loading.mis[,4] <- 0
LY.mis <- simMatrix(loading.mis, "u2")
```

```
mtmm.model.mis <- simMisspecCFA(TE = TE.mis, LY=LY.mis)
```

Next, we need to specify a missing object. This object will indicate both the amount of missingness imposed in the simulated data and the method to handle missing data. The missing object can be made by the simMissing function as

```
SimMissing <- simMissing(pmMCAR=0.2, numImps=5)
```

The pmMCAR argument means the proportion of values in each variable that will be imposed by missing values. The numImps argument is the number of imputations, which implies using the multiple imputation method in missing data handling. If the numImps argument is not specified, the missing data handling method will be full information maximum likelihood.

The data object and the model object can be made by

```
SimData <- simData(500, mtmm.model, misspec = mtmm.model.mis)
SimModel <- simModel(mtmm.model)
```

We can create only one dataset, impose missing values, and analyze the data by

```
data <- run(SimData)
data <- run(SimMissing, data)
result <- run(SimModel, data, SimMissing)
summary(result)
```

The run function on the missing object with a dataset as the second argument will impose missing values on the data. Also, we can add the missing object on the third argument of the run function of the model object to specify the missing data handling method. In this example, we use multiple imputation with 5 imputations. If users do not specify the missing object in the run function, the analysis will use full information maximum likelihood by default. The summary of the result will provide two new columns: FMI1 and FMI2. These are the fraction missing information using two different methods.

The result object can be specified and investigated by

```
Output <- simResult(1000, SimData, SimModel, SimMissing)
getCutoff(Output, 0.05)
plotCutoff(Output, 0.05)
summary(Output)
```

Note that the simulation could be slow because we use five copies of a dataset (i.e., multiple imputation) in each replication. Therefore, we need to run the MTMM model for five times in each replication.

The summary of the simResult object will provide four new columns: the means and the standard deviations of FMI1 and FMI2 across replications.

## Script

```
1   library(simsem)
2
3   u2 <- simUnif(-0.2, 0.2)
4   u49 <- simUnif(0.4, 0.9)
5   u36 <- simUnif(0.3, 0.6)
6   n1 <- simNorm(0, 0.1)
7   n21 <- simNorm(0.2, 0.1)
8   n31 <- simNorm(0.3, 0.1)
9   n41 <- simNorm(0.4, 0.1)
10
11  loading <- matrix(0, 9, 4)
12  loading[1:3, 1] <- NA
```

```
13  loading[4:6, 2] <- NA
14  loading[7:9, 3] <- NA
15  loading[c(1, 4, 7), 4] <- NA
16  loading.v <- matrix(0, 9, 4)
17  loading.v[1:3, 1] <- "u49"
18  loading.v[4:6, 2] <- "u49"
19  loading.v[7:9, 3] <- "u49"
20  loading.v[c(1, 4, 7), 4] <- "u36"
21  LY <- simMatrix(loading, loading.v)
22
23  faccor <- diag(4)
24  faccor[1, 2] <- faccor[2, 1] <- NA
25  faccor[1, 3] <- faccor[3, 1] <- NA
26  faccor[2, 3] <- faccor[3, 2] <- NA
27  faccor.v <- diag(4)
28  faccor.v[1, 2] <- faccor.v[2, 1] <- "n41"
29  faccor.v[1, 3] <- faccor.v[3, 1] <- "n21"
30  faccor.v[2, 3] <- faccor.v[3, 2] <- "n31"
31  PS <- symMatrix(faccor, faccor.v)
32
33  TE <- symMatrix(diag(9))
34
35  mtmm.model <- simSetCFA(LY=LY, PS=PS, TE=TE)
36
37  error.cor.mis <- matrix(NA, 9, 9)
38  diag(error.cor.mis) <- 1
39  TE.mis <- symMatrix(error.cor.mis, "n1")
40  loading.mis <- matrix(NA, 9, 4)
41  loading.mis[is.na(loading)] <- 0
42  loading.mis[,4] <- 0
43  LY.mis <- simMatrix(loading.mis, "u2")
44  mtmm.model.mis <- simMisspecCFA(TE = TE.mis, LY=LY.mis)
45
46  SimMissing <- simMissing(pmMCAR=0.2, numImps=5)
47
48  SimData <- simData(500, mtmm.model, misspec = mtmm.model.mis)
49  SimModel <- simModel(mtmm.model)
50
51  Output <- simResult(1000, SimData, SimModel, SimMissing)
52  getCutoff(Output, 0.05)
53  plotCutoff(Output, 0.05)
54  summary(Output)
```

### Functions Recap

| Functions | Usage |
|---|---|
| simMissing | Create a missing object |

### Remark

1) If users wish to not impose missing values on a set of variables, they can specify the covs argument in the missing object in the Line 46 as

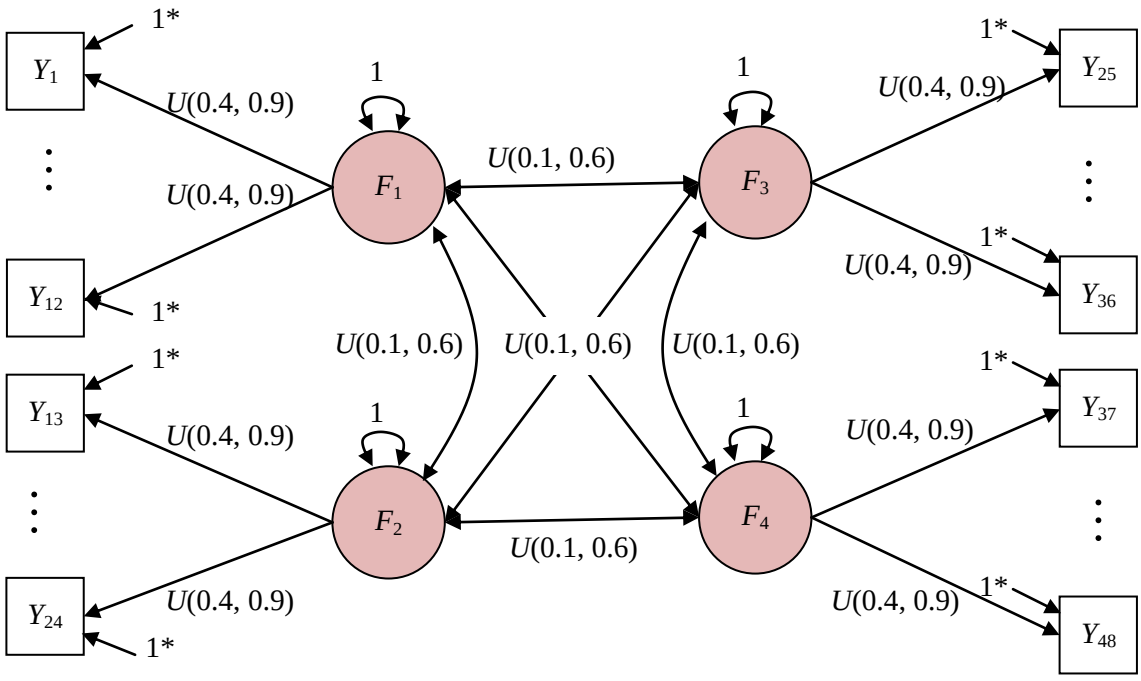
```
SimMissing <- simMissing(pmMCAR=0.2, numImps=5, covs=c(1, 4, 7))
```

Variables 1, 4, and 7 will not have any missing values.

## Example 7

This example will show how to implement a planned missing data. The model of this example is confirmatory factor analysis model with 4 factors and 12 indicators in each factor. We will make a three-form design such that the Indicators 1-3 in each factor are observed in all form, Indicators 4-6 are missing in Form 1, Indicators 7-9 are missing in Form 2, and Indicators 10-12 are missing in Form 3. The factor loading of all indicators are uniformly distributed from 0.4 to 0.9. The factor correlations are uniformly distributed from 0.1 to 0.6. The error variances are constrained such that the indicators variances will be

equal to 1. The trivial model misspecification is specified in cross-loadings only, which is uniformly distributed from -0.2 to 0.2.




1* = Residual variance that makes indicator variance of 1

*Trivially Misspecification*:
All cross loadings have $U(-0.2, 0.2)$.

All relevant distribution objects can be specified as

```
u2 <- simUnif(-0.2, 0.2)
u49 <- simUnif(0.4, 0.9)
u16 <- simUnif(0.1, 0.6)
```

The parameter model can be specified as

```
loading <- matrix(0, 48, 4)
loading[1:12, 1] <- NA
loading[13:24, 2] <- NA
loading[25:36, 3] <- NA
loading[37:48, 4] <- NA
LY <- simMatrix(loading, "u49")
faccor <- matrix(NA, 4, 4)
diag(faccor) <- 1
PS <- symMatrix(faccor, "u16")
TE <- symMatrix(diag(48))
CFA.model <- simSetCFA(LY=LY, PS=PS, TE=TE)
```

The trivial model misspecification can be specified as

```
loading.mis <- matrix(NA, 48, 4)
loading.mis[is.na(loading)] <- 0
LY.mis <- simMatrix(loading.mis, "u2")
CFA.model.mis <- simMisspecCFA(LY=LY.mis)
```

Next, we need to specify a missing object specifying the three-form design. We need to make a group of variables in Set X (variables without any missing values), Set 1, Set 2, and Set 3. Subjects with Form 1 will answer the variables in Set X and Set 1. Subjects with Form 2 will answer the variables in Set X and Set 2. Subjects with Form 3 will answer the variables in Set X and Set 3. After we specify the sets of variables, we group them together in a list and make a missing object as

```
setx <- c(1:3, 13:15, 25:27, 37:39)
set1 <- setx + 3
set2 <- set1 + 3
set3 <- set2 + 3
itemGroups <- list(setx, set1, set2, set3)

SimMissing <- simMissing(nforms=3, itemGroups=itemGroups, numImps=5)
```

The `nforms` argument means the number of forms in the planned missing data design. The `itemGroups` argument means the sets of variables. The number of set must be greater than the number of forms by 1. Then, the `numImps` argument is the number of imputations.

The data, model, and result objects can be made and investigated by

```
SimData <- simData(1000, CFA.model, misspec = CFA.model.mis)
SimModel <- simModel(CFA.model)
Output <- simResult(1000, SimData, SimModel, SimMissing)
getCutoff(Output, 0.05)
plotCutoff(Output, 0.05)
summary(Output)
```

Again, the simulation could be slow because we use five copies of a dataset (i.e., multiple imputation) in each replication.

## Script

```
1   library(simsem)
2
3   u2 <- simUnif(-0.2, 0.2)
4   u49 <- simUnif(0.4, 0.9)
5   u16 <- simUnif(0.1, 0.6)
6
7   loading <- matrix(0, 48, 4)
8   loading[1:12, 1] <- NA
9   loading[13:24, 2] <- NA
10  loading[25:36, 3] <- NA
11  loading[37:48, 4] <- NA
12  LY <- simMatrix(loading, "u49")
13
14  faccor <- matrix(NA, 4, 4)
15  diag(faccor) <- 1
16  PS <- symMatrix(faccor, "u16")
17
18  TE <- symMatrix(diag(48))
19
20  CFA.model <- simSetCFA(LY=LY, PS=PS, TE=TE)
21
22  loading.mis <- matrix(NA, 48, 4)
23  loading.mis[is.na(loading)] <- 0
24  LY.mis <- simMatrix(loading.mis, "u2")
25  CFA.model.mis <- simMisspecCFA(LY=LY.mis)
26
27  setx <- c(1:3, 13:15, 25:27, 37:39)
28  set1 <- setx + 3
29  set2 <- set1 + 3
30  set3 <- set2 + 3
31  itemGroups <- list(setx, set1, set2, set3)
32
33  SimMissing <- simMissing(nforms=3, itemGroups=itemGroups, numImps=5)
```

```
34
35  SimData <- simData(1000, CFA.model, misspec = CFA.model.mis)
36  SimModel <- simModel(CFA.model)
37  Output <- simResult(100, SimData, SimModel, SimMissing)
38  getCutoff(Output, 0.05)
39  plotCutoff(Output, 0.05)
40  summary(Output)
```

### Remark

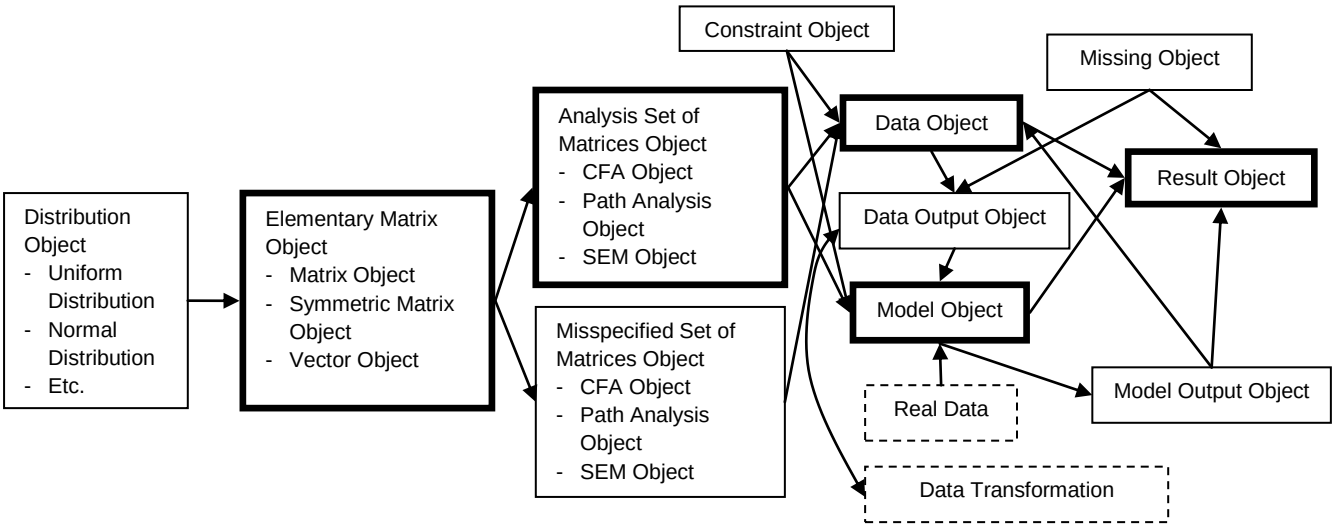
1) Users may implement a two-method design. For example, Indicators 1 is an expensive measurement and users wish to measure it for only 50% of all subjects. The missing object in Line 37 can be changed to

```
SimMissing <- simMissing(twoMethod=c(1, 0.5), numImps=5)
```

where the twoMethod argument is the specification of the two-method design. It takes a vector with two arguments: the index of variable that researchers wish to impose missing values and the proportion of missing values.

## Summary of Model Specification

This picture shows the map of all objects and their relationships in the package. All solid border boxes indicate the objects in the simsem package. The bold-border boxes shows all objects used in the Example 1, which are required objects for simulation. This is the minimal requirement to run a Monte Carlo 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.




## 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 |
| **SimDataOut** | class?SimDataOut |
| **SimModelOut** | class?SimModelOut |
| **SimModelMIOut** | class?SimModelMIOut |
| **SimMissing** | class?SimMissing |

### 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, SimMissing |
| **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 |
| **simMissing** | ?simMissing |
| **runMI** | ?runMI |
| **imposeMissing** | ?imposeMissing |

## 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