How the R package `simsem` Can Be Useful in Simulation Studies of Large-Scale Assessments

Vignette to Accompany a Symposium at NCME 2023

Terrence D. Jorgensen and Alexander M. Schoemann

Friday 31 March 2023

Table of contents

Introduction

This vignette demonstrates how the R package `simsem` can facilitate simulation of large-scale assessment (LSA) data that originate from a structural equation model (SEM). It was written as a support document for a presentation given as part of a symposium:

- Schoemann, A. M., & Jorgensen, T. D. (2023, April). Simulating large-scale assessment data using the R package simsem. In T. Zhang (Chair), *Simulating large-scale assessment data: Tools and practice*. Symposium conducted at the annual meeting of the National Council on Measurement in Education (NCME), Chicago, IL.

All our citations of published research are hyperlinks to their DOI, so this version of the document does not include an explicit **References** section. However, a later (more final) version of the document will include a list of cited **References**.

Abstract from Symposium

Latent variable models (e.g., IRT or SEM) are popular analytic techniques utilized with Large-Scale Assessment (LSA) data, and Monte-Carlo simulations are often used to plan such studies (e.g., determining sample size, evaluating robustness of test statistics). The free, open-source R package `simsem` was developed to aid users in conducting and analyzing Monte Carlo simulations in a latent variable framework. The `simsem` package can generate data and fit models with either the `lavaan` or `OpenMx` packages, or custom functions may be provided by the user (e.g., using M*plus* via the `MplusAutomation` package, or using the `mirt` package for IRT analyses). In this presentation we will demonstrate the basic framework of a Monte Carlo simulation in `simsem`, highlighting features relevant for LSA data. Features will be highlighted in the context of detailed examples, provided in a vignette available at simsem.org. We will summarize the examples to discuss how `simsem` can facilitate implementing Monte Carlo simulations for LSA data with any combination of several features: using discrete item responses via factor analysis with a threshold model, weights and survey attributes, plausible values, and missing values handled via full-information maximum likelihood (FIML) or multiple imputation.

Organization of Content

This vignette is organized into 2 "chapters":

1. Simulating (and Analyzing) LSA Data
2. Conducting a Monte Carlo Study with LSA Data

The `simsem` package is part of the `lavaan` ecosystem, so this vignette primarily demonstrates how (simulated or real) LSA data can be analyzed using `lavaan` (Rosseel, 2012) in a Monte

Carlo simulation study. As of this writing, `lavaan` can analyze LSA data with the following features:

- probability weights, specified by passing a variable name (in the `data=` argument) to the `sampling.weights=` argument
- cluster-robust *SE*s and test statistics, specified by passing a variable name (in the `data=` argument) to the `cluster=` argument
- incomplete data using full-information maximum likelihood (FIML)

Strata are not accommodated directly by `lavaan`, but can be accommodated using the `lavaan.survey` package (Oberski, 2014), which will be demonstrated in Ch. 1. However, `lavaan.survey` does not offer FIML estimation for incomplete data. Multiple imputations of incomplete data can be analyzed by `lavaan.survey` or by the `lavaan.mi()` function in the `semTools` package (soon to be deprecated, when `lavaan.mi` becomes its own package).

Whereas `simsem` will automatically use `lavaan` for estimation of simulated data, the `sim()` function's `model=` argument also accepts a custom data-analysis function that is expected to return output in a predetermined format. This will be demonstrated in Ch. 2 by calling `lavaan.mi()` to analyze plausible values of a latent trait. Other LSA software can also be used to analyze simulated data in `simsem`, such as `Dire` or M*plus* via the `MplusAutomation` package (Hallquist & Wiley, 2018). Ch. 2 will also demonstrate `simsem` features related to imposing missing data on the simulated LSA data.

Ch. 1 Simulating (and Analyzing) LSA Data

The `sim()` function's `generate=` argument also accepts a custom data-generating function, which can utilize other R packages for LSA data, such as `lsasim` (also presented during this symposium). However, we will focus on data-generation using the `lavaan` package itself, which `simsem` accesses via `lavaan::simulateData()`. That function internally calls `MASS::mvrnorm()`, which `simsem` also calls directly when data-generating models are specified using LISREL-style matrices rather than `lavaan::model.syntax`.

The examples begin by showing how to generate data with:

1. multiple strata, represented via multiple-group SEM (MG-SEM)
2. multiple clusters, represented via multilevel SEM (ML-SEM)

In the first case, strata can vary in both mean and covariance structures, but our example will only demonstrate differences in intercepts. Interested readers can see Mang et al. (2021), who simulated German PISA data in which all parameters varied across strata (and they also simulated sampling weights).

In the second case, clusters can vary only in mean structure, consistent with "random intercepts" that can covary across variables (as modeled at Level 2 of a ML-SEM). In all examples, we will use the path diagram below as a template for the data-generating model.

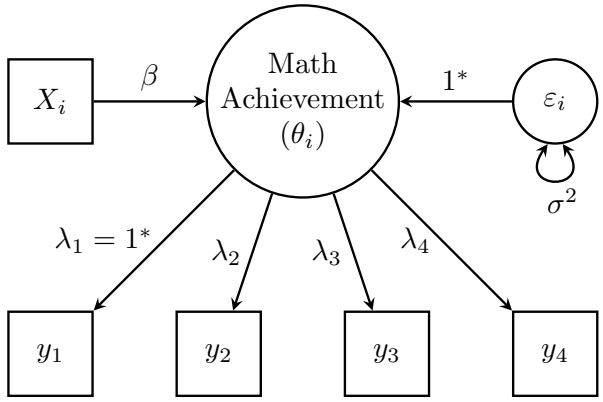

Figure 1: This path diagram depicts 1 latent variable (θ: mathematics achievement) with 4 indicators (test items y_1–y_4) that could be binary, ordinal, or approximately continuous, so mean structure is not depicted. The latent variable is regressed on an observed exogenous predictor X, which could represent a continuous or binary variable. When fitting this model to data, the latent scale can be identified by fixing its residual variance to 1.

1.1 Simulating Stratified Data as Multiple Groups

We will simplify the introduction by beginning with normally distributed data, in which 5 strata differ only in their intercepts. The following example will impose a threshold model for binary data. All examples we provide can easily be scaled up to more strata and more thresholds/categories.

We first specify some population factor loadings, which (if preferred) can be determined by transforming 2-parameter IRT model parameters (Kamata & Bauer, 2008). We will specify population parameters using `lavaan::model.syntax` in order to facilitate adding a threshold model or item responses in a later example. Thresholds cannot be specified directly using LISREL-style matrix specification in `simsem`, but readers may consult `simsem` documentation and vignettes on simsem.org for matrix-specification examples to simulate continuous data. In that case, it is still possible to write a custom function that applies thresholds to the generated data, by passing the function to the `sim(datafun=)` argument.

Simulate Normal Stratified Data

We can specify population parameters in `lavaan::model.syntax` the way we fix parameters (using the `*` operator) or set starting values (using the `?` operator). The latter can be useful when we want to use the same model syntax both for data generation and for analysis (i.e., when many parameters should be estimated rather than fixed).

```
pop.loadings <- ' Math =~ 1*y1 + 0.7*y2 + 1.2*y3 + 0.9*y4 '
```

The syntax above sets loadings explicitly, but perhaps we simply want to draw parameters from a distribution. We could save the random population loadings in a vector and use `paste()` to write the model syntax. Advantageously, `lavaan` accepts a `character` vector of model syntax, so users can specific one or multiple parameters per character string in the vector rather than all parameters in a single quoted string.

```
set.seed(12345)
(ran.lambda <- c(1, # keep first loading = 1 to correspond with analysis model
                 rnorm(n = 3, mean = 1, sd = .25)))
```

```
[1] 1.0000000 1.1463822 1.1773665 0.9726742
```

```
(pop.loadings <- paste0('Math =~ ', round(ran.lambda, 3), '?y', 1:4))
```

```
[1] "Math =~ 1?y1"     "Math =~ 1.146?y2" "Math =~ 1.177?y3" "Math =~ 0.973?y4"
```

In a Monte Carlo study, this would still keep the (one sample of) random population loadings fixed across samples drawn from that population. If instead the loadings should be randomly sampled for each sample/replication of the study, `simsem` matrix-style specification is necessary (see Vignette 4 for an example). But in this section, we only demonstrate how to generate data, not run a full Monte Carlo study.

Next, we specify the population slope β for the regression of Math on X. To interpret it on a standardized metric, we will specify the variance of $X = 1$ and the residual variance of Math to $1 - \beta^2 = 0.91$.

```
pop.slope <- ' Math ~ 0.3?X
  X ~~ 1*X
  Math ~~ 0.91*Math
'
```

We must also specify residual variances for the indicators.

```
(pop.resvar <- paste0('y', 1:4, ' ~~ ', c(1, .9, 1.1, 1), '?y', 1:4))
```

```
[1] "y1 ~~ 1?y1"   "y2 ~~ 0.9?y2" "y3 ~~ 1.1?y3" "y4 ~~ 1?y4"
```

These are all the covariance-structure parameters of our SEM, which we did not set to vary across strata. To vary parameters across strata (groups), simply specify a vector (one value per group) rather than a single value for a parameter. We demonstrate varying intercepts across 5 strata, which are specified by regressing a variable on the number 1 (e.g., `y ~ 1`). The parameter value(s) must be specified before the `*` operator in front of the constant (e.g., `y ~ parameter*1`).

As for loadings, we draw random intercepts below, but they may be predetermined to reflect a particular population (e.g., German PISA data; Mang et al., 2021).

```
(ran.int <- round(runif(n = 5, min = -1, max = 1), 2))
```

```
[1] -0.35  0.02  0.46  0.98 -0.93
```

```
## the collapse= argument is useful to "write" vector syntax
paste(ran.int, collapse = ", ")
```

```
[1] "-0.35, 0.02, 0.46, 0.98, -0.93"
```

```
## This gets pasted into "c()" like this:
paste0('Math ~ c(',
       paste(ran.int, collapse = ", "),
       ")*1")
```

```
[1] "Math ~ c(-0.35, 0.02, 0.46, 0.98, -0.93)*1"
```

```
## now apply this idea to 4 indicators (leave X and Math means = 0)
(pop.int <- sapply(1:4, function(i) {
  ran.int <- round(runif(n = 5, min = -1, max = 1), 2)
  paste0('y', i, ' ~ c(',
```

```
        paste(ran.int, collapse = ", "),
        ")*1")
}))
```

```
[1] "y1 ~ c(-0.7, 0.47, -1, -0.22, -0.08)*1"
[2] "y2 ~ c(-0.22, -0.2, -0.64, 0.9, -0.09)*1"
[3] "y3 ~ c(-0.35, 0.93, 0.41, 0.29, -0.22)*1"
[4] "y4 ~ c(0.4, 0.09, -0.55, -0.03, 0.59)*1"
```

Other parameters (e.g., the slope β) can be specified to vary across strata the same way (i.e., specifying a vector of values: 1 per stratum).

Finally, we concatenate all the parameters into a single character vector and pass it to `simsem::generate()`, which will return a `data.frame` that includes each modeled variable, as well as a column of `group` identifiers. The sample size argument `n=` for a MG-SEM must be a vector of sample sizes (one n per group/strata), which we keep small for simplicity.

```
pop.mod.norm <- c(pop.loadings, pop.slope, pop.resvar, pop.int)
(dat.strat   <- generate(model = pop.mod.norm, n = c(4, 3, 6, 3, 5)))
```

```
            y1         y2          y3          y4          X group
1   0.43955489  2.4723746  3.91564221  3.89733597  1.3872509     1
2  -0.57095215 -1.6598357 -0.59873390  0.07898357 -1.2463084     1
3   0.89054026  0.1240826 -0.40490076  0.63109065 -0.9825452     1
4  -2.59820137 -2.8853460 -1.55319345 -0.33637441  0.8345130     1
5  -0.25272843 -1.3210797  0.84036122  2.27589419 -0.3753162     2
6  -0.14522259  0.5894647  2.22336665  0.20294309  0.9323760     2
7   0.05426169 -0.2973422  0.45180980 -0.32271353 -0.2607622     2
8  -0.54213738 -1.6407673  0.42014742 -1.27992807 -0.6515276     3
9  -1.73496564 -0.1629395 -1.23822032 -0.02289582  0.3405086     3
10 -2.00764494 -1.7056573 -1.58280656 -2.58100892 -0.6142322     3
11 -2.19088494 -2.0687227 -0.69854658 -2.32991961  0.4238001     3
12 -0.47156112  1.7015192  1.12411198 -0.62337840  0.5768838     3
13  0.33194282 -0.2714295  0.40685566 -0.85799086  0.3514253     3
14 -1.54307717  0.8841614  0.61217205 -0.61945872  0.9334666     4
15 -0.71223253 -0.8819495 -0.60701477 -1.45339094  1.0294567     4
16  0.63617330  0.5791814  1.15347829 -1.16295411 -1.1258513     4
17 -0.08124195 -2.3420916 -2.64393184 -0.12490141 -0.5020378     5
18  0.01067828  0.4359390  0.39676065  1.57172529  0.6505319     5
19  2.59707173  3.4841967  2.33358527  3.43771662  1.0222139     5
```

```
20 -1.44899295 -1.9276444 -0.02143799 -0.86427335  0.8694220     5
21 -0.57075860  0.3877689 -0.45488013  1.44146834 -0.1534830     5
```

Analyze Normal Data

We can analyze stratified samples using the `lavaan.survey` package. First, we must fit an initial lavaan model that *ignores* stratification. Because we specified our population parameters as starting values (of which there was only 1, recycled across strata), we can use the same model syntax for analysis, **except for the intercepts**. However, we can simply set `meanstructure=TRUE` for the `sem()` function to automatically estimate indicator intercepts. The first factor loading is fixed to 1 by default. We should also leave out the `pop.slopes` syntax, which fixes the variances of X and Math—neither are necessary in the analysis model, so we will just specify the slope to be estimated.

```
library(lavaan.survey)
mod <- c(pop.loadings, 'Math ~ X', pop.resvar)
fit.naive <- sem(mod, data = dat.strat, meanstructure=TRUE)
# summary(fit) # CANNOT TRUST RESULTS
```

Then we create a survey-design object. Without actual sampling weights or clusters in the data set, we must indicate their absence by `~0` or `~1` (see `?survey::svydesign` for details).

```
myDesign <- svydesign(strata = ~group, ids = ~1, weights = ~1, data = dat.strat)
```

We obtain results that account for strata (or any other design features we have) by passing both the initial fitted SEM and the design object to `lavaan.survey()`.

```
fit.svy <- lavaan.survey(fit.naive, survey.design = myDesign)
summary(fit.svy) # robust SEs, robust test in "Scaled" column
```

```
lavaan 0.6.16 ended normally after 27 iterations

  Estimator                                         ML
  Optimization method                           NLMINB
  Number of model parameters                        13

  Number of observations                            21

Model Test User Model:
                                              Standard      Scaled
  Test Statistic                                10.649      10.394
```

```
  Degrees of freedom                                 5           5
  P-value (Chi-square)                           0.059       0.065
  Scaling correction factor                                  1.025
    Satorra-Bentler correction

Parameter Estimates:

  Standard errors                             Robust.sem
  Information                                   Expected
  Information saturated (h1) model          Structured

Latent Variables:
                   Estimate  Std.Err  z-value  P(>|z|)
  Math =~
    y1                1.000
    y2                1.775    0.513    3.463    0.001
    y3                1.462    0.648    2.255    0.024
    y4                1.363    0.533    2.557    0.011

Regressions:
                   Estimate  Std.Err  z-value  P(>|z|)
  Math ~
    X                 0.389    0.284    1.370    0.171

Intercepts:
                   Estimate  Std.Err  z-value  P(>|z|)
   .y1               -0.536    0.269   -1.992    0.046
   .y2               -0.423    0.396   -1.067    0.286
   .y3                0.101    0.351    0.287    0.774
   .y4               -0.041    0.316   -0.131    0.896
   .Math              0.000

Variances:
                   Estimate  Std.Err  z-value  P(>|z|)
   .y1                0.657    0.188    3.494    0.000
   .y2                0.370    0.303    1.220    0.223
   .y3                0.623    0.220    2.827    0.005
   .y4                1.413    0.358    3.953    0.000
   .Math              0.603    0.482    1.252    0.210
```

Simulate Discrete Stratified Data

We simulated normal/Gaussian data above, but we can also specify population thresholds to discretize those data when we call `generate()`. The `|t1` operator in `lavaan` is used to specify any `ordered=` variable's first threshold. Further thresholds can be specified (per variable) with `|t2`, `|t3`, etc., or all at once (e.g., `variable | t1 + t2 + t3`). Parameter values must be specified after the pipe `|` and before `t1` (or whichever threshold).

In this example, we simply set all thresholds to 0 to generate binary data. For binary data, this is equivalent to setting all intercepts to 0 and setting thresholds to the **same absolute values** of (stratum-specific) intercepts we specified above, but with **opposite sign**. Different approaches may be taken in practice.

```
(pop.th <- paste0('y', 1:4, ' | 0*t1'))
```

```
[1] "y1 | 0*t1" "y2 | 0*t1" "y3 | 0*t1" "y4 | 0*t1"
```

We concatenate these parameters to the rest of our population parameters, specified in the previous section. When we `generate()` data again, we must use the argument `parameterization="theta"` (see `?lavOptions`) so that residual variances are allowed to be model parameters, rather than determined from latent scaling factors under the default `parameterization="delta"` (see `?lavOptions` for details). This argument gets passed to `lavaan::simulateData()` via `...`, as described on the `?generate` help page.

```
pop.mod.ord <- c(pop.loadings, pop.slope, pop.resvar, pop.int, pop.th)
(dat.strat  <- generate(model = pop.mod.ord, n = c(4, 3, 6, 3, 5),
                        parameterization = "theta"))
```

```
   y1 y2 y3 y4           X group
1   1  1  1  1  0.09089568     1
2   1  1  1  1  1.27486391     1
3   1  2  1  1 -0.78368765     1
4   1  1  2  1  0.88994734     1
5   1  1  2  1  0.40520544     2
6   2  2  2  2  0.88930703     2
7   2  1  2  2  0.97314764     2
8   1  1  2  2 -0.12397580     3
9   1  2  2  1  1.93499383     3
10  1  1  1  1 -2.45022742     3
11  1  1  2  1 -0.75927539     3
12  1  1  1  1  0.48355099     3
```

```
13  1  1  1  1  0.23291703     3
14  1  2  1  1  1.44074666     4
15  2  2  1  1 -0.21857750     4
16  1  2  2  2  0.85721746     4
17  1  1  2  2  0.53955761     5
18  2  2  2  2 -1.47883756     5
19  2  1  2  2  1.08928801     5
20  2  2  1  2  0.03095851     5
21  2  2  2  2 -0.48643498     5
```

The `lavaan.survey` package does not accommodate polychoric correlations via threshold models, so these data would have to be treated as continuous to account for strata (not recommended for binary data; Robitzsch, 2020). Although `lavaan` does allow for threshold models, it does not (as of version 0.6-15) implement corrections for stratification, nor other survey-design features (sampling weights or cluster-robust *SE*s) for models with thresholds. Thus, discrete stratified data generated this way should be analyzed with other SEM software (e.g., M*plus*) or IRT packages that can account for stratification. At the end of this tutorial, we demonstrate an example of using a customized data-analyzing function to pass to the `sim(model=)` argument, in case other software is needed for analysis in a Monte Carlo study.

1.2 Simulating Clustered Data

Whereas strata are systematically defined, clusters are random. Unfortunately, the `sim::generate()` function simply passes a character string of model syntax to `lavaan::simulateData()`, which does not generate data from ML-SEM syntax (as of version 0.6-15). So we take this as an opportunity to show how users can specify a customized data-generation function, which could be passed to the `sim(generate=)` argument.

Simulate Normal Clustered Data

For the Level-1 model, we will use the same parameters from the previous section, except for the intercepts, which will all be 0 at Level 1.

```
pop.L1 <- c(pop.loadings, pop.slope, pop.resvar)
```

For design-based inference, only Level-1 parameters are estimated. But we specify Level-2 parameters to simulate this aspect of survey designs (e.g., nonzero intraclass correlation coefficient (ICC) or design effect, which is a function of ICC and N per cluster). We can use `lavaan` to calculate the population-model-implied covariance matrix, which can inform us how large to specify Level-2 variances if we have a target ICC.

```
fitted(lavaan(pop.L1))
```

```
$cov
      y1    y2    y3    y4     X
y1 2.000
y2 1.146 2.213
y3 1.177 1.349 2.485
y4 0.973 1.115 1.145 1.947
X  0.300 0.344 0.353 0.292 1.000
```

The variances are around 2 or more, so Level-2 variances of 1 would yield substantial ICCs around 30%. These (and correlations among random intercepts) can be specified more systematically to answer particular research questions or mimic particular populations (e.g., Mang et al., 2021). Because the Level-2 model is not of substantive interest, we specify arbitrary small-to-medium covariance parameters. Here, we only specify Level-2 (co)variances among the Math indicators, implying X varies only within clusters, which may only be realistic in certain scenarios (e.g., X based on a study-design factor).

```
pop.L2 <- ' y1 ~~ 1*y1 + 0.2*y2 + -0.15*y3 + 0.1*y4
  y2 ~~ 1*y2 + -0.1*y3 + -0.1*y4
  y3 ~~ 1*y3 + -0.15*y4
  y4 ~~ 1*y4

  X ~~ 0*X  # no Level-2 (co)variance
'
```

We can calculate the population ICCs from the model-implied variances:

```
v1 <- diag(fitted(lavaan(pop.L1))$cov)
v2 <- diag(fitted(lavaan(pop.L2))$cov)
v2 / (v1 + v2)
```

```
       y1        y2        y3        y4         X
0.3333333 0.3112050 0.2869170 0.3393593 0.0000000
```

We must generate data from each level separately, then combine them. For example, if we had $N_c = 5$ students from one school, we would generate their 5 Level-1 components:

```
(L1comp <- MASS::mvrnorm(n = 5, mu = rep(0, 5),
                         Sigma = fitted(lavaan(pop.L1))$cov))
```

```
             y1         y2         y3         y4          X
[1,]  0.97328626 -0.1696548 -0.7508086  1.2394340  0.79563878
[2,] -0.35118606 -0.9498305 -2.1471787 -0.8437548 -0.01636357
[3,] -1.15969543 -0.2726120 -0.5304355 -0.4471380 -0.47424276
[4,]  0.72982045  1.3784007  0.9542165 -0.3469542  1.13007893
[5,] -0.03469399  2.1657340  2.0172264  1.6295519  1.77579301
```

Then we would generate a single vector of Level-2 components for that school:

```
(L2comp <- MASS::mvrnorm(n = 1, mu = rep(0, 5),
                         Sigma = fitted(lavaan(pop.L2))$cov))
```

```
       y1         y2         y3         y4          X
-1.1374758 -1.4994889 -0.7330258  0.4625673  0.0000000
```

But this vector of random intercepts needs to be added to each row of the Level-1 components. We can do this with matrix multiplication: premultiply the Level-2 components by a vector of 1s.

```
matrix(1, nrow = 5) %*% L2comp
```

```
          [,1]      [,2]       [,3]      [,4] [,5]
[1,] -1.137476 -1.499489 -0.7330258 0.4625673    0
[2,] -1.137476 -1.499489 -0.7330258 0.4625673    0
[3,] -1.137476 -1.499489 -0.7330258 0.4625673    0
[4,] -1.137476 -1.499489 -0.7330258 0.4625673    0
[5,] -1.137476 -1.499489 -0.7330258 0.4625673    0
```

Then its dimensions match the Level-1 components, so the 2 matrices can be added together and converted to a `data.frame`.

```
data.frame(L1comp  +  matrix(1, nrow = 5) %*% L2comp)
```

```
          y1         y2         y3          y4           X
1 -0.1641896 -1.6691437 -1.4838344  1.70200129  0.79563878
2 -1.4886619 -2.4493194 -2.8802046 -0.38118748 -0.01636357
3 -2.2971712 -1.7721009 -1.2634613  0.01542934 -0.47424276
4 -0.4076554 -0.1210883  0.2211907  0.11561313  1.13007893
5 -1.1721698  0.6662451  1.2842006  2.09211920  1.77579301
```

We can write a function that repeats this process by looping over Level-2 units:

- generate a vector of random intercepts (Level-2 components)
- generate the Level-1 components for that Level-2 unit
- combine them the components into a single set of variables

Then stack each Level-2 data set into a single `data.frame`.

The function below simulates a random cluster size of $N_c = 10$–20 students sampled per school.

```
gen2L <- function(N2) {
  NperC <- sample(10:20, size = N2, replace = TRUE) # N1 == sum(NperC)

  ## specify population models
  popL1 <- ' Math ~ 0.3*X
    Math =~     1*y1
    Math =~ 1.146*y2
    Math =~ 1.177*y3
    Math =~ 0.973*y4

    X    ~~    1*X
    Math ~~ 0.91*Math
    y1   ~~    1*y1
    y2   ~~  0.9*y2
    y3   ~~  1.1*y3
    y4   ~~    1*y4
  '
  popL2 <- ' y1 ~~ 1*y1 + 0.2*y2 + -0.15*y3 + 0.1*y4
    y2 ~~ 1*y2 + -0.1*y3 + -0.1*y4
    y3 ~~ 1*y3 + -0.15*y4
    y4 ~~ 1*y4

    X ~~ 0*X  # no Level-2 (co)variance
  '
  ## model-implied covariance matrices
```

```
  Sigma1 <- fitted(lavaan::lavaan(popL1))$cov
  Sigma2 <- fitted(lavaan::lavaan(popL2))$cov
  ## make sure order of names is consistent
  Sigma2 <- Sigma2[rownames(Sigma1), colnames(Sigma1)]

  clusterList <- lapply(1:N2, function(cl) {
    L2comp <- MASS::mvrnorm(n = 1, mu = rep(0, 5), Sigma = Sigma2)
    dat2 <- t(L2comp) %x% rep(1, NperC[cl]) # apply to each L1 component
    dat1 <- MASS::mvrnorm(n = NperC[cl], mu = rep(0, 5), Sigma = Sigma1)
    ## combine components
    dat <- data.frame(dat1 + dat2, schoolID = rep(cl, NperC[cl]))
    dat
  })
  ## stack all Level-2 data sets
  do.call(rbind, clusterList)
}
```

Then we can call that function with a Level-2 sample size (e.g., 100 schools).

```
set.seed(12345)
dat.clus <- gen2L(100)
head(dat.clus)
```

```
         y1         y2          y3         y4          X schoolID
1  0.5138769 -3.4954219 -0.23904622 -1.5045827  0.7029619        1
2  0.6174791 -0.6694169 -1.26295141 -2.6852780 -0.2777064        1
3  1.5333204  1.6694645  0.64065110  0.1314912 -0.3839743        1
4  0.9896491 -1.0933640  0.66782558 -2.6001046 -1.8942508        1
5 -0.5476870 -2.9638183 -0.02803752 -1.6581844  0.5351057        1
6  1.7490903 -0.6803488 -1.96049128 -1.4686362 -2.0486780        1
```

We wrote the `gen2L()` function with only 1 argument (the level-2 sample size) because in `simsem`, passing a custom function to the `sim(generate=)` argument is only possible when that function has (only) a sample-size argument.

Analyze Normal Clustered Data

We can analyze the clustered data using `lavaan`, even with `missing="FIML"` and `sampling.weights=` (not simulated here). Simply specify the (single-level) hypothesized model, then request cluster-robust *SE*s and test statistics by passing the cluster-ID variable's name (`school`) to the `cluster=` argument.

```r
mod.clus <- ' Math ~ X
  Math =~ y1 + y2 + y3 + y4
'
fit.clus <- sem(mod.clus, data = dat.clus, cluster = "schoolID")
summary(fit.clus, standardized = TRUE) # use "Scaled" test
```

```
lavaan 0.6.16 ended normally after 29 iterations

  Estimator                                         ML
  Optimization method                           NLMINB
  Number of model parameters                        13

  Number of observations                          1555
  Number of clusters [schoolID]                    100

Model Test User Model:
                                              Standard      Scaled
  Test Statistic                                21.048       8.064
  Degrees of freedom                                 5           5
  P-value (Chi-square)                           0.001       0.153
  Scaling correction factor                                  2.610
    Yuan-Bentler correction (Mplus variant)
  Observed information based on                     H1

Parameter Estimates:

  Standard errors                         Robust.cluster
  Information                                   Observed
  Observed information based on                 Hessian

Latent Variables:
                   Estimate  Std.Err  z-value  P(>|z|)   Std.lv  Std.all
  Math =~
    y1                1.000                               1.099    0.654
    y2                1.058    0.082   12.862    0.000    1.163    0.670
    y3                0.983    0.101    9.746    0.000    1.080    0.573
    y4                0.804    0.084    9.567    0.000    0.884    0.518

Regressions:
                   Estimate  Std.Err  z-value  P(>|z|)   Std.lv  Std.all
  Math ~
```

```
    X                 0.334    0.037    9.076    0.000    0.304    0.301

Intercepts:
                   Estimate  Std.Err  z-value  P(>|z|)   Std.lv  Std.all
   .y1                0.053    0.098    0.545    0.585    0.053    0.032
   .y2                0.277    0.095    2.919    0.004    0.277    0.160
   .y3               -0.003    0.116   -0.030    0.976   -0.003   -0.002
   .y4               -0.205    0.109   -1.878    0.060   -0.205   -0.120
   .Math              0.000                             0.000    0.000

Variances:
                   Estimate  Std.Err  z-value  P(>|z|)   Std.lv  Std.all
   .y1                1.614    0.125   12.874    0.000    1.614    0.572
   .y2                1.661    0.184    9.021    0.000    1.661    0.551
   .y3                2.389    0.281    8.492    0.000    2.389    0.672
   .y4                2.133    0.222    9.594    0.000    2.133    0.732
   .Math              1.098    0.124    8.855    0.000    0.910    0.910
```

Simulate Discrete Clustered Data

The threshold model links observed discrete responses to latent continuous (e.g., normal) responses. In the case of clustered data, the observed and latent responses are linked *prior to the latent responses being decomposed* into Level-1 and Level-2 components. Thus, to generate discrete data, we simply apply population thresholds (e.g., `c(-1, 0, 1)` to generate 4 categories) to the data generated by `gen2L()`.

```
ord.clus <- dat.clus # copy data
## loop over indicators (leave X alone)
for (i in 1:4) {
  ord.clus[,paste0("y", i)] <- cut(ord.clus[,paste0("y", i)],
                                   # include +/- infinity thresholds
                                   breaks = c(-Inf, -1:1, Inf),
                                   labels = FALSE)
}
head(ord.clus)
```

```
  y1 y2 y3 y4          X schoolID
1  3  1  2  1  0.7029619        1
2  3  2  1  1 -0.2777064        1
3  4  4  3  3 -0.3839743        1
4  3  1  3  1 -1.8942508        1
```

```
5  2  1  2  1  0.5351057        1
6  4  2  1  1 -2.0486780        1
```

In a Monte Carlo study using `sim()`, we can write a separate function to do this (passed to the `datafun=` argument), or we can simply add this step to the end of the `gen2L()` function before returning the data. But neither `lavaan` (as of 0.6-15) nor `lavaan.survey` offer cluster-robust *SE*s/tests for ordinal outcomes modeled with thresholds. So either ordinal data would be treated as numeric (to use `lavaan`), or other software for discrete data can be used for analysis.

Ch. 2 Conducting a Monte Carlo Study with LSA Data

In this section, we go beyond data generation to highlight `simsem` functionality that is useful specifically for conducting Monte Carlo studies. The primary interface of `simsem` is the `sim()` function, although many of its arguments have their own interfaces (e.g., constructing a SEM with LISREL-style matrices via `bind()` and `model()` functions; see their help-page documentation for descriptions and examples).

We begin by extending the previous clustered-data-generation example to a Monte Carlo simulation (i.e., multiple samples are drawn from the specified population, each is analyzed, and results are summarized across replications/samples). This example demonstrates the `sim()` function, whose critical components are:

- a data-generating model for the `sim(generate=)` argument. This is the `gen2L()` function defined above.
- a data-analyzing model for the `sim(model=)` argument. This is the object `mod.clus`, which contains model syntax defined above.
- the size of each sample (in our case, the Level-2 sample size) via the `n=` argument
- the number of replications/samples to draw (i.e., the "Monte Carlo sample size"), which we set as `nRep=10` only to keep our examples short)

Because we are passing `lavaan::model.syntax` to the `model=` argument, we should specify which `lavaan` function we want to pass that model to: the `sem()` function. Further arguments can be passed to `sem()` via `...`, such as the `cluster=` argument.

```
out2L <- sim(seed = 12345, nRep = 10, n = 100, # Level-2 N
             generate = gen2L,     # function to generate
             model = mod.clus,     # lavaan script to analyze
             lavaanfun = "sem",    # use sem() function
             cluster = "schoolID", # argument passed to sem()
             silent = TRUE)#, # FALSE prints when each replication completes
             ## optional parallel processing to speed it up:
```

```
                #multicore = TRUE, numProc = 10)
```

The `sim()` function returns a `SimResult` object, whose contents are described on the `class?SimResult` help page. Because the generating model `mod2g` is altogether different than the analysis model `mod2L`, the population parameters in `out2L@paramValue` do not match the format of estimates in `out2L@coef`. In order to take full advantage of the `summaryParam()` function, the `SimResult`'s `out2L@paramValue` slot must be updated. Below, we save the first row of `out2L@coef` to an object `dummyPar` that we use as a template to set our population parameters.

```
out2L@paramValue
```

```
  V1
1 NA
```

```
(dummyPar <- out2L@coef[1,]) # initialize population parameters
```

```
     Math~X Math=~y2  Math=~y3  Math=~y4   y1~~y1   y2~~y2   y3~~y3  y4~~y4
1 0.3642161 1.079274 0.9236164 0.8435905 1.848704 1.644475 2.459207 2.17303
  Math~~Math    y1~1      y2~1       y3~1       y4~1
1   1.055578 0.12645 -0.161802 0.01707955 -0.0859721
```

```
## Update @paramValue slot:
dummyPar[1] <- .3                          # slope
dummyPar[2:4] <- round(ran.lambda[-1], 3) # loadings 2:4
dummyPar[5:9] <- c(1, .9, 1.1, 1, .91)    # residual variances
dummyPar[9:12] <- 0                        # intercepts
## save the parameters in the @paramValue slot:
out2L@paramValue <- dummyPar
```

This is not absolutely necessary, but it does enable the full advantages of the `summaryParam()` function, which automatically calculates bias of point and *SE* estimates, CI-coverage rates, and rejection rates (labeled `Power (Not equal 0)`, but really Type I error rates when H_0 is true). If this is not the focus of the Monte Carlo study, then the `out2L@paramValue` slot does not need to be updated. Also bear in mind that the cluster-level covariances (survey-design effects) are the source of bias seen below, and that the bias estimates have a lot of sampling error because our Monte Carlo sample sizes was only `nRep=10`.

```r
summaryParam(out2L, detail = TRUE, digits = 3)
```

```
          Estimate Average Estimate SD Average SE Power (Not equal 0) Std Est
Math~X                0.333       0.040      0.037                 1.0   0.306
Math=~y2              1.075       0.111      0.100                 1.0   0.649
Math=~y3              0.927       0.139      0.115                 1.0   0.538
Math=~y4              0.855       0.093      0.086                 1.0   0.540
y1~~y1                1.760       0.171      0.180                 1.0   0.599
y2~~y2                1.838       0.224      0.198                 1.0   0.578
y3~~y3                2.416       0.137      0.228                 1.0   0.710
y4~~y4                2.047       0.136      0.196                 1.0   0.708
Math~~Math            1.081       0.245      0.138                 1.0   0.906
y1~1                  0.015       0.128      0.107                 0.0   0.008
y2~1                 -0.041       0.111      0.108                 0.1  -0.023
y3~1                  0.026       0.066      0.105                 0.0   0.014
y4~1                  0.024       0.090      0.105                 0.1   0.014
          Std Est SD Std Ave SE Average Param Average Bias Coverage Rel Bias
Math~X         0.023      0.029         0.300        0.033      0.8    0.110
Math=~y2       0.044      0.040         1.146       -0.071      0.9   -0.062
Math=~y3       0.030      0.045         1.177       -0.250      0.3   -0.212
Math=~y4       0.026      0.044         0.973       -0.118      0.7   -0.121
y1~~y1         0.074      0.047         1.000        0.760      0.0    0.760
y2~~y2         0.057      0.052         0.900        0.938      0.0    1.042
y3~~y3         0.032      0.049         1.100        1.316      0.0    1.196
y4~~y4         0.028      0.047         1.000        1.047      0.0    1.047
Math~~Math     0.014      0.017         0.000        1.081      0.0      Inf
y1~1           0.074      0.062         0.000        0.015      1.0       NA
y2~1           0.063      0.061         0.000       -0.041      0.9       NA
y3~1           0.036      0.057         0.000        0.026      1.0       NA
y4~1           0.053      0.062        -0.086        0.110      0.9   -1.281
          Std Bias Rel SE Bias Not Cover Below Not Cover Above
Math~X       0.831     -0.081            0.2            0.0
Math=~y2    -0.640     -0.099            0.0            0.1
Math=~y3    -1.800     -0.171            0.0            0.7
Math=~y4    -1.268     -0.069            0.0            0.3
y1~~y1       4.446      0.052            1.0            0.0
y2~~y2       4.194     -0.115            1.0            0.0
y3~~y3       9.632      0.670            1.0            0.0
y4~~y4       7.686      0.437            1.0            0.0
Math~~Math   4.408     -0.436            1.0            0.0
y1~1         0.114     -0.169            0.0            0.0
```

```
y2~1          -0.368      -0.028             0.0             0.1
y3~1           0.388       0.588             0.0             0.0
y4~1           1.223       0.169             0.1             0.0
           Average CI Width SD CI Width
Math~X                 0.143       0.017
Math=~y2               0.392       0.117
Math=~y3               0.451       0.147
Math=~y4               0.339       0.094
y1~~y1                 0.705       0.069
y2~~y2                 0.776       0.126
y3~~y3                 0.894       0.129
y4~~y4                 0.767       0.136
Math~~Math             0.542       0.064
y1~1                   0.418       0.030
y2~1                   0.424       0.029
y3~1                   0.413       0.021
y4~1                   0.412       0.023
```

A variety of other outcomes could be of interest, such as statistics and indices of data–model fit. We encourage readers to explore possibilities demonstrated in the vignettes available at simsem.org.

Now, we extend this basic example of a Monte Carlo simulation to 2 more examples:

- In the first example, we show how to specify missing-data mechanisms with the `simsem::miss()` function, then choose whether to analyze the incomplete simulated data using FIML or multiple imputation.
- In the second example, we show how to use the `semTools` package to generate `plausibleValues()` from a `lavaan` model and fit a subsequent path analysis using the `lavaan.mi()` function (which can also be used to analyze multiple imputations of missing data). This example demonstrates the flexibility of `simsem` to utilize custom functions for data **generation**, **transformation**, and **analysis** via the `generate=`, `datafun=` and `model=` arguments, respectively.

2.1 Impose Missing Mechanisms

LSA data are regularly incomplete. To simulate data that are missing *completely* at random (MCAR), there is a global approach and a variable-specific approach:

- **Global**: An MCAR mechanism can be imposed on all simulated variables by passing a number between 0 and 1 to the argument `sim(pmMCAR=)`, which will impose that same probability of missingness on every variable. For example, we can set `sim(..., pmMCAR = 0.25)` to randomly set 25% of each variable's observations as missing (i.e., the

missing-data patterns will differ across variables, so it is *not* 25% of the sample missing *all* variables).

- **Variable-specific**: Different missing-data rates can be specified for each variable by writing `lavaan`-like model syntax. This option requires creating a `SimMissing` object via the `miss()` function, which can be passed to the `sim(miss=)` argument.

The latter approach is also how we can specify more complicated missing-data mechanisms that depend on other variables. When analyzing/imputing the incomplete data, conditioning on variables that explain missingness corresponds to the less restrictive missing at random (MAR) assumption, but failing to condition on such a variable would make the mechanism missing *not* at random (MNAR). When the variables we (must) condition on are not part of our substantive-hypothesized data-generating model, the conditioning variables are called *auxiliary variables*, which can be included in the imputation model only or (when using FIML or two-stage MLE) in the hypothesized model as saturated correlates (Enders, 2008).

To specify missingness mechanisms separately for each variable, a logistic regression model can be specified in a character string. Any simulated variable on the left-hand side indicates there is dummy-coded variable corresponding to whether that variable is missing (1) or observed (0). The right-hand side of each variable's equation must include an intercept or (if it is a value between 0 and 1 in parentheses) the overall probability of missing values. Optionally, any predictors on the right-hand side of each variable's equation indicate a MAR mechanism, and the (fixed) slope quantifies the log-odds change in the probability that variable is missing. This character string is then passed to the `miss(logit=)` argument.

```
missMech <- '## MCAR for first 2 indicators
  y1 ~ p(.1)  # 10% chance of missing data
  y2 ~ -2     # logit-scale intercept implies 12% chance
## MAR for third indicator, explained by fourth indicator
  y3 ~ p(.1) + -0.2*y4    # 22% decrease in odds
'
myMissingnessModel <- miss(logit = missMech)
```

Many other missing-data mechanisms (e.g., longitudinal attrition, planned-missing designs) are automated by `simsem::miss()`. Some demonstrations can be found in Vignettes 8, 9, and 10 on simsem.org. Note that combinations of mechanisms are possible.

We can test the missingness mechanism by imposing it on a `data.frame`, such as the `dat.clus` object we generated with `gen2L()`. We can either use the `imposeMissing()` function, which accepts the same arguments we pass to `miss()`:

```
set.seed(654321)
head(imposeMissing(dat.clus, logit = missMech))
```

```
          y1         y2         y3         y4          X schoolID
1  0.5138769 -3.4954219         NA -1.5045827  0.7029619        1
2  0.6174791 -0.6694169 -1.2629514 -2.6852780 -0.2777064        1
3  1.5333204  1.6694645  0.6406511  0.1314912 -0.3839743        1
4         NA -1.0933640  0.6678256 -2.6001046 -1.8942508        1
5 -0.5476870 -2.9638183         NA -1.6581844  0.5351057        1
6  1.7490903 -0.6803488         NA -1.4686362 -2.0486780        1
```

Or because we have already specified our mechanism(s) via `miss()`, we can pass that `SimMissing-class` object to the `impose()` function:

```
set.seed(654321)
head(miss.clus <- impose(myMissingnessModel, dat.clus))
```

```
          y1         y2         y3         y4          X schoolID
1  0.5138769 -3.4954219         NA -1.5045827  0.7029619        1
2  0.6174791 -0.6694169 -1.2629514 -2.6852780 -0.2777064        1
3  1.5333204  1.6694645  0.6406511  0.1314912 -0.3839743        1
4         NA -1.0933640  0.6678256 -2.6001046 -1.8942508        1
5 -0.5476870 -2.9638183         NA -1.6581844  0.5351057        1
6  1.7490903 -0.6803488         NA -1.4686362 -2.0486780        1
```

```
## match expectations?
mean(is.na(miss.clus$y1)) # about 10%?
```

```
[1] 0.09710611
```

```
mean(is.na(miss.clus$y2)) # about 12%?
```

```
[1] 0.1382637
```

Analyze Incomplete Data with FIML

Now that we have verified the mechanism works, we can apply it to every generated sample during our Monte Carlo simulation. All other `sim()` arguments are the same as when we created the `out2L` object above, but we additionally specify the `miss=` argument now.

```
mis2L <- sim(seed = 12345, nRep = 10, n = 100, # Level-2 N
             generate = gen2L,     # function to generate
           miss = myMissingnessModel, # NEW: impose missing mechanism(s)
             model = mod.clus,     # lavaan script to analyze
             lavaanfun = "sem",    # use sem() function
             cluster = "schoolID", # argument passed to sem()
             silent = TRUE, # FALSE prints when each replication completes
             ## optional parallel processing to speed it up:
             multicore = TRUE, numProc = 10)
```

We can then inspect the output as needed. See previous section about using `summaryParam()`, if that is of interest.

Note that because the `model=` argument was a character vector of `lavaan::model.syntax`, the incomplete data were analyzed using the argument `lavaan(missing = "FIML")`. This is because the default `miss(m=)` argument is 0, indicating no imputations of missing data.

Analyze Multiple Imputations of Incomplete Data

To use multiple imputations, simply specify a positive integer to indicate how many times to impute missing values:

```
miss(logit = missMech, m = 5, package = "mice")
```

In this case, `simsem` will use the `semTools::lavaan.mi()` function to fit the model and pool results across multiple imputations. However, this option might disappear in a future version, after `lavaan.mi()` is deprecated from `semTools` and moved to its own package. It will always be possible to write a custom data-analysis function (for the `sim(model=)` argument) that calls `lavaan.mi()`. Likewise, imputing LSA data requires careful consideration of the design features, so relying on the automated `simsem::miss()` feature is not advisable. Instead, one should write a custom data-transformation function for the `sim(datafun=)` argument, which performs an appropriate imputation using any software (not merely the `Amelia` or `mice` packages automatically called by `simsem`) and returns a `list` of imputed data sets. The next example demonstrates both features, albeit by generating plausible values of latent variables rather than imputing incomplete observed variables.

2.2 Draw Plausible Values of Latent Variables

The full versatility of `simsem` means that it can be useful for Monte Carlo simulations that have nothing to do with SEM at all! For instance, we can write functions that `generate=` multilevel data that we analyze with multilevel regression (e.g., using `lme4:lmer()`, passed to

the `model=` argument). Details are provided below, but we first discuss data-generation and data-transformation functions in more detail.

Data-Generation Functions

A data-generating function passed to `sim(generate=)` can only have 1 argument that specifies the sample size(s). For MG-SEMs, this can be a vector of integers (one sample size per group). In our `gen2L()` example above, our sample-size argument `N2=` was the number of clusters, and the overall Level-1 sample size was determined within the function by randomly sampling `NperC` from 10–20. This means there is a different overall number of Level-1 units in every sample drawn from that population model. If we wanted to keep that constant, we could more systematically set Level-1 sample sizes, e.g., using an equal number of `NperC = c(10, 13, 15, 17, 20)`. That would guarantee the average `NperC=` is always 15, yielding a constant Level-1 $N = 15 * 100 = 1500$. The top of the `gen2L()` function could be amended like this:

```
gen2L <- function(N2) {
  L1.sizes <- c(10, 13, 15, 17, 20)
  NperC <- rep(L1.sizes, each = N2 / length(L1.sizes))
  ...
}
```

The data-generating function should return data in whatever format is expected by the data-analyzing function. For example, `lavaan()` typically expects a `data.frame`, but the custom function calling `lavaan.mi()` below will expect a list of them. If missing data are to be imputed, the imputation could already happen in a custom data-generating function. Alternatively (as demonstrated below), the incomplete data returned by the data-generating function can be "transformed" into a list of multiple imputations via a data-transformation function, the latter being passed to `sim(datafun=)`.

Data-Transformation Functions

A data-transformation function must have a single argument to accept a `data.frame` (or whatever format is returned by the custom data-generating function). The data-transformation function must ultimately return data in whatever format is expected by the data-analyzing function. In this example, we need a `list` of data to be analyzed by `lavaan.mi()`, but these values are not standard imputations in incomplete observed variables. Instead, they are imputations of latent variables, which (by definition) are missing for everyone. Factor scores can be requested for fitted `lavaan` models, with an argument `acov=TRUE` to additionally request their uncertainty (an asymptotic covariance matrix of each subject's vector of factor scores). These will be cluster-robust *SE*s when requested from `lavaan`, as our example above did when fitting the `fit.clus` model.

```
FS <- lavPredict(fit.clus, acov = TRUE)
head(FS)         # point estimate of first 6 subjects' factor score
```

```
           Math
[1,] -0.8558380
[2,] -0.6152169
[3,]  0.7139834
[4,] -0.5135800
[5,] -0.9729193
[6,] -0.5026752
```

```
attr(FS, "se")   # SE of each estimated factor score
```

```
[[1]]
          Math
[1,] 0.5860306
```

```
attr(FS, "acov") # sampling (co)variance of factor scores
```

```
[[1]]
          Math
Math 0.3434319
```

The `semTools::plausibleValues()` function generates a random sample of factor scores from a multivariate normal distribution with `MASS::mvrnorm(n, mu = FS, Sigma = attr(FS, "acov"))` per group. In this case, it is a univariate normal distribution because there is only 1 latent variable (Math). By default, `nDraws=20` plausible values will be sampled, but here we use only 3 to keep the demonstration brief.

```
library(semTools)
PVs <- plausibleValues(fit.clus, nDraws = 3, seed = 123)
## print the head() of each "imputation"
lapply(PVs, head)
```

```
[[1]]
  case.idx        Math
1        1 -0.50802934
2        2  0.12580921
3        3  0.12344473
4        4 -0.04284748
5        5 -1.38276745
6        6 -0.25454002

[[2]]
  case.idx       Math
1        1 -0.6468181
2        2  0.6140808
3        3  0.9894020
4        4 -1.0308674
5        5 -0.1585456
6        6 -0.6880608

[[3]]
  case.idx        Math
1        1 -0.79555098
2        2  0.02965808
3        3  0.90278097
4        4 -0.08084251
5        5 -1.72787336
6        6 -0.03775671
```

The example above generated plausible values from a SEM that already regressed latent Math scores on X, but in practice we would use an unrestricted measurement model (e.g., IRT or CFA, conditioning on background variables) to estimate factor scores / person parameters; then we would use plausible values in a subsequent regression or path analysis. Regardless, we need each `data.frame` to include `X` (or any other modeled variables) to be modeled in a subsequent path analysis.

```r
## add row index as variable in original data
dat.clus$case.idx <- lavInspect(fit.clus, "case.idx")
## use it to merge with each set of PVs
for (i in seq_along(PVs)) {
  PVs[[i]] <- merge(PVs[[i]], dat.clus, by = "case.idx")
  PVs[[i]]$case.idx <- NULL # no more need for this index
}
lapply(PVs, head)
```

```
[[1]]
         Math         y1         y2          y3         y4          X schoolID
1 -0.50802934  0.5138769 -3.4954219 -0.23904622 -1.5045827  0.7029619        1
2  0.12580921  0.6174791 -0.6694169 -1.26295141 -2.6852780 -0.2777064        1
3  0.12344473  1.5333204  1.6694645  0.64065110  0.1314912 -0.3839743        1
4 -0.04284748  0.9896491 -1.0933640  0.66782558 -2.6001046 -1.8942508        1
5 -1.38276745 -0.5476870 -2.9638183 -0.02803752 -1.6581844  0.5351057        1
6 -0.25454002  1.7490903 -0.6803488 -1.96049128 -1.4686362 -2.0486780        1

[[2]]
        Math         y1         y2          y3         y4          X schoolID
1 -0.6468181  0.5138769 -3.4954219 -0.23904622 -1.5045827  0.7029619        1
2  0.6140808  0.6174791 -0.6694169 -1.26295141 -2.6852780 -0.2777064        1
3  0.9894020  1.5333204  1.6694645  0.64065110  0.1314912 -0.3839743        1
4 -1.0308674  0.9896491 -1.0933640  0.66782558 -2.6001046 -1.8942508        1
5 -0.1585456 -0.5476870 -2.9638183 -0.02803752 -1.6581844  0.5351057        1
6 -0.6880608  1.7490903 -0.6803488 -1.96049128 -1.4686362 -2.0486780        1

[[3]]
         Math         y1         y2          y3         y4          X schoolID
1 -0.79555098  0.5138769 -3.4954219 -0.23904622 -1.5045827  0.7029619        1
2  0.02965808  0.6174791 -0.6694169 -1.26295141 -2.6852780 -0.2777064        1
3  0.90278097  1.5333204  1.6694645  0.64065110  0.1314912 -0.3839743        1
4 -0.08084251  0.9896491 -1.0933640  0.66782558 -2.6001046 -1.8942508        1
5 -1.72787336 -0.5476870 -2.9638183 -0.02803752 -1.6581844  0.5351057        1
6 -0.03775671  1.7490903 -0.6803488 -1.96049128 -1.4686362 -2.0486780        1
```

Notice the merged data now includes `X` as well as `"schoolID"`, which was not part a modeled variable but **would be needed for cluster-robust results** of a subsequent path model.

Now we need to write a function that will fit the cluster-robust CFA to generated data and return the list of data with plausible values.

```r
getPVs <- function(data) {
  library(semTools)
  ## specify CFA
  mod <- ' Math =~ y1 + y2 + y3 + y4 ' # also Math ~ any background variables
  ## fit CFA to data, requesting cluster-robust results
  fit <- sem(mod, data = data, cluster = "schoolID", std.lv = TRUE)
  ## estimate plausible values
  PV <- plausibleValues(fit, nDraws = 3)
  ## merge with original data
```

```r
  data$case.idx <- lavInspect(fit, "case.idx")
  for (i in seq_along(PV)) {
    PV[[i]] <- merge(PV[[i]], data, by = "case.idx")
    PV[[i]]$case.idx <- NULL # no more need for this index
  }
  PV
}
## test it once
PVs <- getPVs(dat.clus)
lapply(PVs, head)
```

```
[[1]]
        Math         y1         y2          y3         y4          X schoolID
1 -0.8310734  0.5138769 -3.4954219 -0.23904622 -1.5045827  0.7029619        1
2 -0.1708188  0.6174791 -0.6694169 -1.26295141 -2.6852780 -0.2777064        1
3  0.5820924  1.5333204  1.6694645  0.64065110  0.1314912 -0.3839743        1
4 -1.1384291  0.9896491 -1.0933640  0.66782558 -2.6001046 -1.8942508        1
5 -1.0428836 -0.5476870 -2.9638183 -0.02803752 -1.6581844  0.5351057        1
6 -0.8328979  1.7490903 -0.6803488 -1.96049128 -1.4686362 -2.0486780        1

[[2]]
         Math         y1         y2          y3         y4          X schoolID
1 -1.13021062  0.5138769 -3.4954219 -0.23904622 -1.5045827  0.7029619        1
2 -0.42502871  0.6174791 -0.6694169 -1.26295141 -2.6852780 -0.2777064        1
3 -0.07355671  1.5333204  1.6694645  0.64065110  0.1314912 -0.3839743        1
4 -0.59225623  0.9896491 -1.0933640  0.66782558 -2.6001046 -1.8942508        1
5 -1.08642759 -0.5476870 -2.9638183 -0.02803752 -1.6581844  0.5351057        1
6 -0.09314458  1.7490903 -0.6803488 -1.96049128 -1.4686362 -2.0486780        1

[[3]]
         Math         y1         y2          y3         y4          X schoolID
1 -1.00974468  0.5138769 -3.4954219 -0.23904622 -1.5045827  0.7029619        1
2 -0.46902358  0.6174791 -0.6694169 -1.26295141 -2.6852780 -0.2777064        1
3 -0.71157559  1.5333204  1.6694645  0.64065110  0.1314912 -0.3839743        1
4 -0.17868661  0.9896491 -1.0933640  0.66782558 -2.6001046 -1.8942508        1
5 -1.90122504 -0.5476870 -2.9638183 -0.02803752 -1.6581844  0.5351057        1
6  0.09670342  1.7490903 -0.6803488 -1.96049128 -1.4686362 -2.0486780        1
```

Now this function can be passed to `sim(datafun=)`, which will be applied **after** data are generated via `generate=gen2L` but **before** data are analyzed via the custom function defined next.

Data-Analysis Functions

Now that we have obtained plausible values, we can now analyze them using multiple-imputation methods to pool results across the samples/imputations of plausible values. This will appropriately incorporate their uncertainty/indeterminacy into the pooled *SE*s and test statistics. The `lavaan.mi()` function, and its wrappers `cfa.mi()` or `sem.mi()`, automate the fitting of a SEM to multiple imputations (or in this example, samples of plausible values). Because the path model in this example is a simple regression model, we could also use other packages (e.g., `mitml`) to pool cluster-robust regression results, e.g., from `survey::svyglm()`.

```
# library(lavaan.mi) # eventually its own package
fit.mi <- sem.mi('Math ~ X', data = PVs, cluster = "schoolID")
( pooledResults <- summary(fit.mi, output = "table") )
```

```
  lhs op  rhs    est    se      t      df pvalue
1 Math  ~    X  0.207 0.027  7.571 158.573  0.000
2 Math ~~ Math  0.965 0.038 25.462  92.183  0.000
3    X ~~    X  0.981 0.000     NA      NA     NA
4 Math ~1       0.003 0.041  0.074 775.417  0.941
5    X ~1      -0.033 0.000     NA      NA     NA
```

Now we need to write a function for the `sim(model=)` argument that performs such an analysis and returns a `list` of the results expected by `simsem`. The only argument must be for the data (in this case, a list of `data.frame`s), and it must return (at a minimum) a `list` containing estimated coefficients (`$coef`), their *SE*s (`$se`), and a logical vector indicating whether the SEM `$converged` for each replication. Other elements of interest can also be returned (see **Details** in the `?sim` documentation). These required arguments can be easily obtained from a `class?lavaan.mi` object.

```
coef(fit.mi, type = "user") # same as pooledResults$est, but with names()
```

```
  Math~X Math~~Math       X~~X     Math~1        X~1
   0.207      0.965      0.981      0.003     -0.033
```

```
pooledResults$se # borrow names(coef)
```

```
[1] 0.02735839 0.03789406 0.00000000 0.04119545 0.00000000
```

```
fit.mi@convergence # model converged for all 3 "imputations"
```

```
[[1]]
 converged         SE Heywood.lv Heywood.ov
      TRUE       TRUE      FALSE      FALSE

[[2]]
 converged         SE Heywood.lv Heywood.ov
      TRUE       TRUE      FALSE      FALSE

[[3]]
 converged         SE Heywood.lv Heywood.ov
      TRUE       TRUE      FALSE      FALSE
```

There are multiple indicators in `fit.mi@convergence`, indicating not only whether the optimizer found a solution for point estimates (`$converged`) but also whether `$SE` estimates could be calculated or whether Heywood cases were detected (i.e., an "improper solution" that yields a model-implied observed/latent-variable covariance matrix that is not positive definite).

```
coef(fit.mi, type = "user") # same as pooledResults$est, but with names()
```

```
   Math~X Math~~Math       X~~X     Math~1        X~1
    0.207      0.965      0.981      0.003     -0.033
```

```
pooledResults$se # borrow names(coef)
```

```
[1] 0.02735839 0.03789406 0.00000000 0.04119545 0.00000000
```

```
fit.mi@convergence # model converged for all 3 "imputations"
```

```
[[1]]
 converged         SE Heywood.lv Heywood.ov
      TRUE       TRUE      FALSE      FALSE

[[2]]
 converged         SE Heywood.lv Heywood.ov
```

```
      TRUE       TRUE      FALSE      FALSE

[[3]]
 converged          SE Heywood.lv Heywood.ov
      TRUE       TRUE      FALSE      FALSE
```

Because `simsem` expects only a single `logical` value for whether each replication converged, the 3 values in `fit.mi@convergence` will lead to an error. We must decide on a rule for convergence:

- **Strict**: Point and *SE* estimates must be found for `all()` samples of plausible values per replication. This is probably safe for our example (a simple regression model), but it could be problematic in more complex models.
- **Minimum possible**: Rubin's rules can be applied to pool results of as few as 2 imputations, so we could accept that few per replication.
- **Minimum advisable**: Rubin (1987) recommended 5 imputations for point estimates to be unbiased estimates of population parameters.

We will employ the strict rule here, by checking (for each sample of plausible values) whether both `$converged` and `$SE` are `TRUE`, and whether that is the case for all samples of plausible values.

```
all(fit.mi@convergence[[1]][c("converged","SE")]) # first PV
```

```
[1] TRUE
```

```
all(sapply(fit.mi@convergence, function(pv) { # loop over all PVs
  all(pv[c("converged","SE")])
}))
```

```
[1] TRUE
```

The custom function below formats them more fully for `sim()` to store in its results.

```
fitPVs <- function(dataList) {
  library(semTools) # or library(lavaan.mi)
  fit <- sem.mi('Math ~ X', data = dataList, cluster = "schoolID")
  EST <- summary(fit, output = "table", ci = TRUE)
  ## save the 3 necessary elements
```

```
  COEF <- coef(fit, type = "user")
  SE <- setNames(EST$se, nm = names(COEF))
  CONV <- all(sapply(fit@convergence, function(pv) {
                    all(pv[c("converged","SE")])  }))
  ## return them in a list, with some other info
  list(coef = COEF, se = SE, converged = CONV,
       ## confidence limits too:
       cilower = setNames(EST$ci.lower, nm = names(COEF)),
       ciupper = setNames(EST$ci.upper, nm = names(COEF)) )
}
## test it once
fitPVs(PVs)
```

```
$coef
    Math~X Math~~Math       X~~X     Math~1        X~1
     0.207      0.965      0.981      0.003     -0.033

$se
    Math~X Math~~Math       X~~X     Math~1        X~1
0.02735839 0.03789406 0.00000000 0.04119545 0.00000000

$converged
[1] TRUE

$cilower
    Math~X Math~~Math       X~~X     Math~1        X~1
     0.153      0.890      0.981     -0.078     -0.033

$ciupper
    Math~X Math~~Math       X~~X     Math~1        X~1
     0.261      1.040      0.981      0.084     -0.033
```

Now this function can be passed to `sim(model=)` in a Monte Carlo study. Again, the same custom function could be applied to analyze multiple imputations of missing data.

Run Monte Carlo Simulation

This time we call `sim()` with similar arguments as when we created the `out2L` object, but instead of using `lavaan::model.syntax` as our analysis `model=`, we use the custom function defined above. We also extract plausible values using the `datafun=` argument. We no longer

pass the `cluster=` argument to `lavaan()`, but it is specified in the `fitPVs()` function that calls `sem.mi()` with the `cluster=` argument.

```
simPV <- sim(seed = 12345, nRep = 10, n = 100, # Level-2 N
             generate = gen2L, # function to generate clustered data
             datafun = getPVs, # extract plausible values, then
             model = fitPVs,   # analyze them with sem.mi() function
             silent = TRUE)#,    # FALSE prints when each replication completes
             ## optional parallel processing to speed it up:
             #multicore = TRUE, numProc = 10)
```

Again, we can set the `@paramValue` slot manually if we want to use `summaryParam()`.

```
(dummyPar <- simPV@coef[1,])  # initialize population parameters
```

```
    Math~X Math~~Math      X~~X      Math~1        X~1
1 0.2407136  0.9722308 0.9185464 -0.01462541 0.04262128
```

```
dummyPar[1] <- .3             # slope
dummyPar[2:3] <- c(.91, 1)    # (residual) variances
dummyPar[4:5] <- c(0, 0)      # intercepts
## save the parameters in the @paramValue slot:
simPV@paramValue    <- dummyPar

## check bias of standardized estimates
summaryParam(simPV, detail = TRUE, digits = 3, matchParam = TRUE)
```

```
          Estimate Average Estimate SD Average SE Power (Not equal 0)
Math~X                    0.215       0.020      0.029                   1
Math~~Math                0.966       0.012      0.043                   1
X~~X                      0.987       0.031         NA                   1
Math~1                   -0.008       0.006      0.047                   0
X~1                       0.021       0.023         NA                   1
          Average Param Average Bias Coverage Rel Bias Std Bias Rel SE Bias
Math~X             0.30       -0.085      0.3   -0.282   -4.259       0.464
Math~~Math         0.91        0.056      0.9    0.062    4.868       2.723
X~~X               1.00       -0.013      0.0   -0.013   -0.417          NA
Math~1             0.00       -0.008      1.0       NA   -1.353       7.003
X~1                0.00        0.021      0.0       NA    0.900          NA
          Not Cover Below Not Cover Above Average CI Width SD CI Width
```

Math~X	0.0	0.7	0.124	0.019
Math~~Math	0.1	0.0	0.183	0.024
X~~X	0.4	0.6	0.000	0.000
Math~1	0.0	0.0	0.185	0.021
X~1	0.7	0.3	0.000	0.000