

City University of New York (CUNY)

CUNY Academic Works

Open Educational Resources

John Jay College of Criminal Justice

2016

Implementing Some Basic Simulation Designs Using the simsem Package in R

Keith A. Markus
CUNY John Jay College

[How does access to this work benefit you? Let us know!](#)

More information about this work at: https://academicworks.cuny.edu/jj_oers/25

Discover additional works at: <https://academicworks.cuny.edu>

This work is made publicly available by the City University of New York (CUNY).
Contact: AcademicWorks@cuny.edu

Implementing Some Basic Simulation Designs Using the simsem Package in R

Keith A. Markus

John Jay College of Criminal Justice & Graduate Center
The City University of New York

Version 3

Fall 2016

(c) 2016 Keith A. Markus

Implementing Some Basic Simulation Designs Using the `simsem` Package in R

The purpose of this tutorial is to provide a very basic introduction to implementing three simple research designs using the `simsem` package in R. R is an open source statistical computing environment (R Core Team, 2015). For more information about R, see the R Project homepage (<https://www.r-project.org/>) and the Comprehensive R Archive Network (CRAN) web page (<https://cran.r-project.org/>). The `lavaan` package provides functions for fitting and evaluating structural equation models (Rosseel, 2012). For further information about the `lavaan` package including tutorials, see the `lavaan` Project web page (<http://lavaan.ugent.be/>). The `simsem` package (Pornprasertmanit, Miller & Schoemann, 2016) provides functions to facilitate structural equation modeling simulation studies and is compatible with both `lavaan` and `openMx` (Boker, et al., 2014). For more information about the `simsem` package, see the `simsem` web page (<http://simsem.org/>). This document assumes some familiarity with the basics of R and assumes familiarity with `lavaan` model specification. The `simsem` web page provides extensive examples for running specific simulation conditions. The goal of this document is to provide a basic guide to implementing simple research designs by combining multiple conditions into a larger design for comparative analysis. Sample code illustrates the complete process including simulation, basic data management strategies, and graphical display of results. The document will focus on `lavaan` model specification and estimation but the techniques generalize to `openMx`. For more information about `openMx`, see the `openMx` web page (<http://openmx.psyc.virginia.edu/>).

The first section considers a simple two-group between-subjects design. The second section considers a within subjects design with two conditions. The third section combines these into a 2×2 mixed factorial design. The final section discusses the use of functions in programming simulations. This document also assumes basic familiarity with factorial research designs and terms used to describe them. Four self-contained R scripts accompany this tutorial.

Between-Subjects Design

In a between-subjects design, different treatments are applied to different cases. In the context of a simulation study, cases are generally data sets. So, in a between-subjects design, different data sets are used in each condition. Between-subjects designs primarily occur where the conditions of the simulation study differ in a way that impacts the nature of the data analyzed in those conditions. Examples include studies that vary the sample size, the number of variables, the statistical distributions of the variables, or patterns of missing data. Such designs are relatively straightforward to implement because each cell in the design can be run separately and then combined after the fact for further analysis. To illustrate, the example will compare standard errors for an indirect effect with sample sizes of 50 and 500. The complete R code with annotations can be found in the file <Between Subjects Simulation Design.R>.

Before specifying the structural equation model, it is helpful to set the random number seed. This allows you or others to reproduce identical results when re-running the simulation.

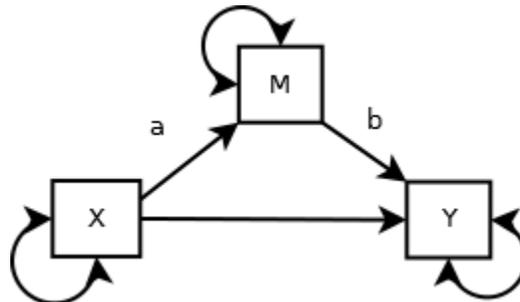
```
set.seed(12345)
```

The next two lines make the `lavaan` and `simsem` packages available for use. The packages must be installed prior to using the `require()` or `library()` function. You only need to install once, aside from updates, but need to load each package each time you begin a new R session.

```
require(lavaan)
require(simsem)
```

The model involves a simple three-variable mediation model (Figure 1). The simulations will focus on the standard error of the indirect effect computed as the product of parameter a , the effect of X on M and the parameter b , the effect of M on Y .

Figure 1.
Mediation Model.



The model can be written in equations as follows.

$$M = aX + e_M$$

$$Y = bM + cX + e_Y$$

Using `lavaan` model specification, one can specify the model as follows. The present specification defines the model used to simulate the data. As such it specifies the values for all six parameters in the model: one variance (s_x), two residual variances (e_m and e_y), and three effect coefficients (a , b , and c).

```
simModel <- '
  # Mediation path model

  # Regressions
```

```

M ~ .5*X
Y ~ .5*M + .5*X

# Variances and residuals
X ~~ 1*X
M ~~ .5*M
Y ~~ .5*Y

```

The above lines create an R object encoding the model as a string. lavaan ignores comments marked off by '#' but such comments make models easier to read for human users. The regressions parallel the above equations. The lines with the double-tilde operator encode the variance of X and the residuals variances for M and Y .

The numbers in the model are fixed parameter values used to simulate the data. In order to use a model for simulation it must be fully specified with no free parameters so that it implies a unique covariance matrix (and possibly mean vector). One typically wants to select realistic values to support generalization of results to empirical applications. When one simulates data for a specific substantive context, existing empirical research can offer the best source for such values. When the simulation abstracts over specific content, then one can still summarize the distribution of effects in a given literature and use that distribution to guide choices of parameter values. A third option involves fitting a specific model to a, preferably large, empirical data set or obtaining parameter estimates from a previously published study. One then takes the parameter estimates as population values in the simulation.

One also needs to specify a model to be fit to the data in the simulation.

```

fitModel <- '
# Mediation path model
# Variance of exogenous variable assumed fixed

# Regressions
M ~ a*X
Y ~ b*M + X

# Residuals
M ~~ M
Y ~~ Y

# Computed parameter for indirect effect

```

```
i := a * b  
,
```

Notice that this specification does not provide all the parameter values because these will be estimated when fitting the model to the data. This specification also adds a computed parameter, i , to represent the indirect effect of X on Y through M . The purpose of adding this parameter to the fitted model is to provide a standard error for statistical inference about the indirect effect. (The code does not label parameter c because it does not use c to compute i , but there is no problem with labeling c if you want to.)

Single analyses, with no bootstrapping, often run very quickly. Repeating similar analyses hundreds or even thousands of times takes longer. So, test your code with a single repetition before launching an entire condition. Although not shown here, the accompanying R scripts illustrate such checks.

One can run the $N=50$ condition as follows.

```
# Run simulation with N = 50  
simOutput_1 <- sim(  
  nRep = 1000,  
  model = fitModel,  
  n = 50,  
  generate = simModel,  
  lavaanfun = 'lavaan',  
  se='standard'  
)
```

simOutput_1 is an object created in your work environment by the code to store the result. You can give this any name consistent with R syntax but it helps to have a naming system to keep things straight.

The code uses six parameters to the *sim()* function. The *nRep* parameter sets the number of replications (cases) in the condition to 1000. The *model* parameter sets the model to be fit to the data using the R object created earlier called *fitModel*. The *n* parameter sets the sample size for each data set at 50, corresponding to the independent variable in the design. The *generate* parameter specifies the data generating model and the *lavaanfun* parameter specifies the function from the lavaan package used to fit the model, here *lavaan()*.

The accompanying script illustrates the *summary()* and *summaryTime()* functions to obtain summary results for just this condition. One can run the second condition simply by changing

the sample size (n) and saving the result into a different R object to avoid overwriting the $n = 50$ data.

Having run both conditions, one can then combine them into a single data set as follows.

```
# Compare results for indirect effect
iSE.50 <- inspect(simOutput_1, "se",
  improper = TRUE,
  nonconverged = TRUE)[,6]
iSE.500 <- inspect(simOutput_2, "se",
  improper = TRUE,
  nonconverged = TRUE)[,6]
NCondition <- rep(c(50,500), each=length(iSE.50))
iSE <- c(iSE.50, iSE.500)
betweenSubjects.df <- data.frame(NCondition=NCondition, iSE=iSE)
str(betweenSubjects.df)
```

The first two assignments (here spanning six lines) assign the standard errors from each condition to an R object which in this case constitutes a vector of data values. The desired values are in the sixth column of the output of the `inspect()` function. So, the `[,6]` serves to select the sixth column from this output. The only difference between the two lines is that one extracts from `simOutput_1` and the other from `simOutput_2`. This simple example examines only one outcome. For multiple outcomes, copy all the desired outcomes into one data set rather than creating separate data sets for each outcome. For example, `[2:6]` would extract columns 2 through 6 and `[c(2,6)]` would extract columns 2 and 6. One can then incorporate the resulting data frame object into the desired data frame for the study using the `data.frame()` function but without naming the data frame object to be incorporated (e.g., `newDataframe <- data.frame(oldDataframe, newVariables)`).

The next line creates an R object identifying the condition for each case by repeating the number 50 as many times as there are values in `iSE.50` and then repeating the number 500 the same number of times. Note that this works because the design is balanced. If you run different numbers of replications in different conditions, then you would need to tweak this line to reflect that. The value assigned to the `times` parameter can be a vector such as `times = c(500, 1000)`.

The next line concatenates the two vectors of data values into a single vector called `iSE`. The second last line then combines `iSE` and `NCondition` as two columns in a single data frame. The structure function, `str()`, offers a useful generic way to check the structure of an object in R. The accompanying R script use this function at each step. This can be helpful for debugging. Once

your code is working, one can comment out the *str()* functions to speed it up. Here is the result of the last line:

```
> str(betweenSubjects.df)
'data.frame':  2000 obs. of  2 variables:
 $ NCondition: num  50 50 50 50 50 50 50 50 50 50 ...
 $ iSE       : num  0.0903 0.0545 0.0597 0.0976 0.0957 ...
>
```

The result shows a data frame with 2000 cases and two variables. The first column contains the independent variable identifying the condition. Note that the first column of the data frame is described by the first row of the output. The second column contains the dependent variable equal to the standard error of the indirect effect. For example, the first case had $n = 50$ and produced an SE of 0.0903.

A simple *t* test confirms the difference between conditions of about 0.06.

```
> t.test(betweenSubjects.df$iSE ~ betweenSubjects.df$NCondition)

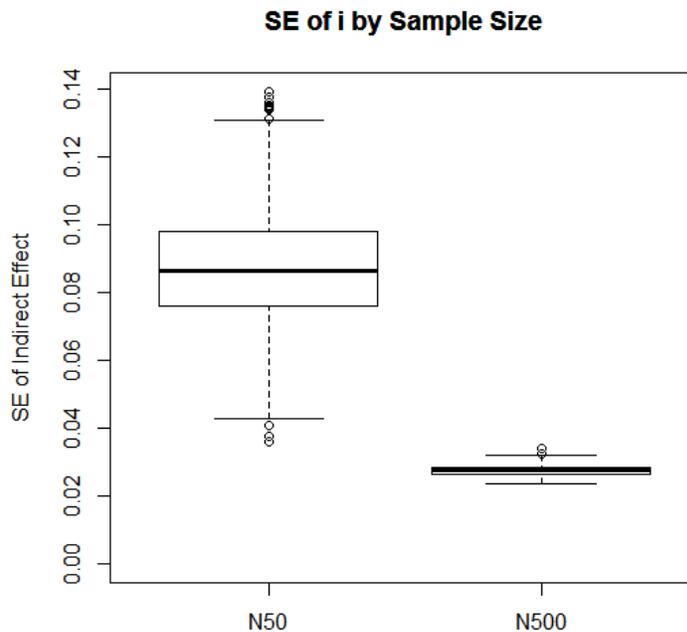
Welch Two Sample t-test

data:  betweenSubjects.df$iSE by betweenSubjects.df$NCondition
t = 113.13, df = 1017.9, p-value < 2.2e-16
alternative hypothesis: true difference in means is not equal to
0
95 percent confidence interval:
 0.05905573 0.06114054
sample estimates:
mean in group 50 mean in group 500
 0.08753205      0.02743391
>
```

A box-and-whisker plot provides a graphical display of the results.

```
boxplot(x=list(N50=betweenSubjects.df$iSE[1:1000],
              N500=betweenSubjects.df$iSE[1001:2000]),
        ylim=c(0, max(betweenSubjects.df$iSE)))
title(main='SE of i by Sample Size',
      ylab='SE of Indirect Effect')
```

Figure 2.
Between-subjects Design



As one might expect, the larger sample size produces both smaller standard errors and also more consistency in standard errors across data sets. (The next example elaborates this basic graph.)

To sum up, for the between-subjects design, the script ran each condition separately and saved the output from each condition as a separate R object in the workspace. Once those were in place, the script then extracted the data of interest into vectors and concatenated these across conditions. The script also added an independent variable to identify which cases came from which conditions by repeating the desired values. The final step then combined these variables as columns in a data frame used for analysis of the simulation results.

Within-Subjects Design

Within-subjects designs occur when one compares different analyses of the same data. This requires a slightly different strategy. First, one generates the data sets. Then one recycles those data sets in each condition. To illustrate this, the next script analyzes the same data sets once using standard (ML) standard errors for the indirect effect and again using robust Huber White (MLR) standard errors. This simulation can reuse the same data because the difference in methods of computing standard errors does not require differences in the data. When the independent variable allows, analyzing the same data different ways is preferable to generating

new data sets because it eliminates sampling variation between conditions. (Unlike people, data sets are not affected by one condition in a way that can impact outcomes in other conditions.) The script for this example appears in the file <Within Subjects Simulation Design.R>.

The script for the within-subjects design begins the same way as the between-subjects design. The initial steps make the required packages available, set the random number seed, and create the models used to generate the data and to analyze the data.

The next step generates the data sets.

```
# Generate data sets and put into list for future use.
Nrep <- 1000
Data50 <- vector('list', Nrep)
for(rep in 1:Nrep){
  Data50[[rep]] <- generate(model=simModel, n=50)
}
length(Data50) # check
```

The *Nrep* object stores the number of replications so that when modifying the code one only has to make one change. The script creates an R object called *Data50* to store the data sets. Rather than the *list()* function, the second line use the *vector()* function to specify the length of the desired list as the number of data sets (*Nrep*). One could simply use *list()* to create an empty list but then R would expand the list each time you add a data set rather than creating a list large enough to hold them all and then inserting the data sets into the list. Imagine building just enough bookshelf for the next book as you place 1000 books on a shelf instead of installing the bookshelf all at once and then filling the empty space with books. Creating a list of the needed length all at once will allow R to complete the task faster because it does not have to keep reallocating memory¹ (Burns, 2011). You will not notice much difference for small simulations, but the difference in speed gets larger as the list gets larger.

The *for()* function creates an explicit loop, repeating the process inside the braces 1000 times. (To speed up code, one generally tries to avoid loops in R but sometimes they come in handy.) The R object named *rep* serves as an index for the loop, increasing in value from 1 to 1000. You can manually confirm this after the loop is finished.

```
> rep
[1] 1000
>
```

¹ This practice runs counter to good programming practice in certain other languages, such as C. So, if you learned another language first, do not let that throw you.

The line inside the braces generates one data set using the *generate()* function from the *simsem* package, specifying the data generating model and the sample size. The loop repeats this line 1000 times. Each time, the resulting data set is added to the existing list named *Data50* in a position identified by the current value of *rep*. The name of *rep* appears in double brackets in order to write the entire data set as one element of the list. If one used a single bracket, R would expect each element of the data set to correspond to a unique element of this list (*Data50*) but that is not the desired result. Note that you cannot use the *?* short cut for help with brackets. You need to use the long form: *help('')*. The last line confirms the expected number of replications in the within-subjects design. (The next example provides a more rigorous check.)

```
> length(Data50) # check
[1] 1000
>
```

The next step runs the first condition (ML) using the saved list of data sets.

```
# Run simulation with ML (standard) standard errors
simOutput_ML <- sim(
  nRep = NULL,
  model = fitModel,
  rawData = Data50,
  lavaanfun = 'lavaan',
  se='standard'
)
```

The R object called *simOutput_ML* will store the results. The call uses five parameters to the *sim()* function. *nRep* must be set to null because the number of data sets stored in *Data50* determines the number of replications. The *model* parameter again specifies the model to be analyzed. The *rawData* parameter specifies the list of data sets. The *lavaanfun* parameter again specifies the function used to fit the models. Finally, the *se* parameter specifies the desired type of standard errors, corresponding to the independent variable in this example.

To run the second condition one need only change the name of the R object and the value of the *se* parameter.

```
# Run simulation with MLR (robust) standard errors
simOutput_MLR <- sim(
  nRep = NULL,
  model = fitModel,
```

```

rawData = Data50,
lavaanfun = 'lavaan',
se='robust.huber.white'
)

```

Whereas the previous example put the between-subjects data into narrow format because all the observations were unique, the next lines of code place the within-subjects data into wide format to reflect the fact that different conditions used the same observations (data sets). (The final example illustrates narrow format with partially within-subjects data.)

```

# Compare results for indirect effect
iSE.ML <- inspect(simOutput_ML, "se", improper = TRUE,
nonconverged = TRUE) [,6]
iSE.MLR <- inspect(simOutput_MLR, "se", improper = TRUE,
nonconverged = TRUE) [,6]
withinSubjects.df <- data.frame(ML=iSE.ML, MLR=iSE.MLR)
str(withinSubjects.df)

```

Analogous to before, the *iSE.ML* and *iSE.MLR* objects store the standard errors for their respective conditions. In this case, the script places these into the data frame as separate columns. As such, the condition is identified by the variable name rather than a separate independent variable in the data set. The result comprises a data frame with 1000 rows and two columns entirely filled with standard errors of the indirect effect.

```

> str(withinSubjects.df)
'data.frame': 1000 obs. of 2 variables:
 $ ML : num 0.0977 0.0736 0.0628 0.0735 0.0766 ...
 $ MLR: num 0.0869 0.0658 0.0624 0.0819 0.0869 ...
>

```

In this case a paired t-test summarizes the difference between conditions.

```

> t.test(withinSubjects.df$ML, withinSubjects.df$MLR,
paired=TRUE)

```

Paired t-test

```

data: withinSubjects.df$ML and withinSubjects.df$MLR
t = 6.1691, df = 999, p-value = 9.962e-10
alternative hypothesis: true difference in means is not equal to
0

```

```
95 percent confidence interval:
```

```
0.001640789 0.003171551
```

```
sample estimates:
```

```
mean of the differences
```

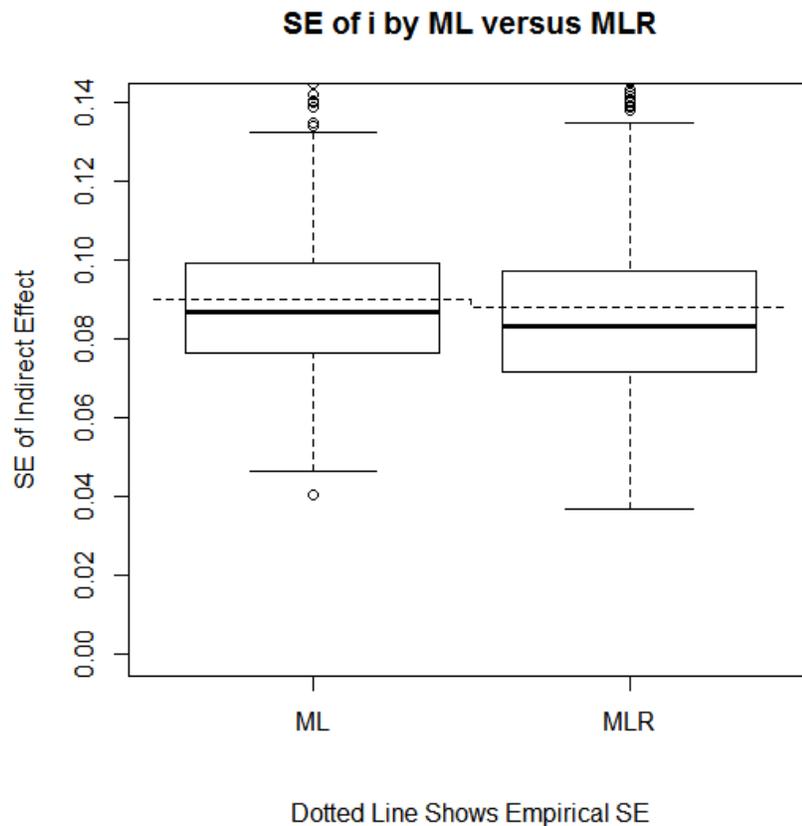
```
0.00240617
```

```
>
```

Finally, a box-and-whisker plot displays the results. For reference, the lines after the *title()* function extract and plot the empirical standard errors equal to the standard deviation of the estimates.

```
boxplot(x=list(ML=withinSubjects.df$ML,
MLR=withinSubjects.df$MLR),
ylim=c(0, max(witinSubjects.df$iSE)))
title(main='SE of i by ML versus MLR',
ylab='SE of Indirect Effect',
sub='Dotted Line Shows Empirical SE')
empirical.se.i.ML <- inspect(simOutput_ML, "coef",
improper = TRUE,
nonconverged = TRUE)[,6]
empirical.se.i.MLR <- inspect(simOutput_MLR, "coef",
improper = TRUE,
nonconverged = TRUE)[,6]
lines(x=c(0.5,1.5,1.5,2.5),
y=rep(c(sd(empirical.se.i.ML), sd(empirical.se.i.MLR)),
each=2),
lty=2)
```

Figure 3.
Within-subjects Design



The standard errors are smaller under MLR than ML but both methods appear to underestimate the true standard errors based on the dotted line showing the standard deviation of indirect effects across data sets.

Some simulations may involve a within-subjects design but also some transformation or modification of the data between conditions. For example, one might wish to compare an analysis using the original variables to the same analysis with the variables centered around their means. As other examples, one might wish to log-transform one or more variables, or compute sum scores for a scale or for item parcels. The *datafun* parameter to the *sim()* function can be very useful for such purposes. To use this parameter, one first defines a function to carry out the transformations and add the transformed variables to the data set (or replace the data). Here, I use the *return()* function to make this explicit. In the *sim()* function call, one would then specify *datafun = myDataFunction*. The transformed variables will then be available for analysis in the simulation. Different analyses can analyze different subsets of variables.

```
myDataFunction <- function(data, ...){
  Xcent <- data$X - mean(data$X) # Mean-centered X
```

```

Ylog <- ln(data$Y)          # Log transformed Y
XplusM <- data$X + data$M    # Sum of X and M
return(data.frame(data, Xcent, Ylog, XplusM))
}

```

To summarize, the within-subjects design required a somewhat different strategy from the between-subjects design. The first step involved generating the data sets for analysis and saving them in a list object. The second and third steps involved running each condition using these saved data sets. Finally, the last steps combined the results into a wide formatted data frame and used that to analyze the results. Choose wide or narrow format depending upon your intended analysis. The next example illustrates narrow formatted data.

Mixed Factorial Design

This final example combines the above two examples into a single four-condition design. The sample size independent variable (50, 500) remains between-subjects and the standard error independent variable (ML and MLR) remains within-subjects. The result is a 2×2 mixed factorial design with one factor of each type. The initial steps remain the same. So, as with the previous example, the exposition picks up at the step generating the data sets.

```

# Generate data sets and put into lists for future use.
nRep <- 1000
Data50 <- vector('list',nRep)
Data500 <- vector('list',nRep)
for(rep in 1:nRep){
  Data50[[rep]] <- generate(model=simModel, n=50)
  Data500[[rep]] <- generate(model=simModel, n=500)
}
table(unlist(lapply(Data50, dim))) # Check
table(unlist(lapply(Data500, dim)))

```

This code expands the previous example by defining two empty lists (*Data50* and *Data500*) and then filling them with data sets. (Warning: this will overwrite the object of the same name from the previous example.) The data sets are generated within a single loop by including two separate lines inside the braces saving the result of a *generate()* function. The end result consists of two lists of 1000 data sets each. The last two lines provide a check that the data sets have the desired dimensions of 3 columns and either 50 or 500 rows.

```
> table(unlist(lapply(Data50, dim))) # Check
```

```

      3    50
1000 1000
> table(unlist(lapply(Data500, dim)))

```

```

      3    500
1000 1000
>

```

Just as in the previous example, the next steps involve running each cell of the design using the *sim()* function with the stored data sets appropriate to that condition. For example, here is the code for just the first of the four conditions.

```

# Run simulation with ML (standard) standard errors, N=500
simOutput_ML500 <- sim(
  nRep = NULL,
  model = fitModel,
  rawData = Data500,
  lavaanfun = 'lavaan',
  se='standard'
)

```

The objects *simOutput_ML500* (shown above), *simOutput_ML50*, *simOutput_MLR500*, and *simOutput_MLR50* contain the results, each for one of the four cells of the design. The only differences between the code for the four cells are (a) the name of the R object, (b) the value of the *rawData* parameter (*Data50* or *Data500*) and (c) the value of the *se* parameter ('standard' or 'robust.huber.white').

Once all four conditions have run, the next step again involves combining the desired results into a single data frame. This example does not involve any new concepts, it just has more pieces to combine.

```

# Compare results for indirect effect
iSE.ML50 <- inspect(simOutput_ML50, "se", improper = TRUE,
  nonconverged = TRUE)[,6]
iSE.MLR50 <- inspect(simOutput_MLR50, "se", improper = TRUE,
  nonconverged = TRUE)[,6]
iSE.ML500 <- inspect(simOutput_ML500, "se", improper = TRUE,
  nonconverged = TRUE)[,6]
iSE.MLR500 <- inspect(simOutput_MLR500, "se", improper = TRUE,
  nonconverged = TRUE)[,6]
iSE.all <- c(iSE.ML50, iSE.MLR50, iSE.ML500, iSE.MLR500)

```

```

NCondition <- rep(c(50,500),each=2*length(iSE.ML50))
SECondition <- rep(c('ML','MLR','ML','MLR'),
  each=length(iSE.ML50))
Case=rep(1:(2*length(iSE.ML50)), times=2)
mixedFactorial.df <- data.frame(Case,NCondition, SECondition,
  iSE=iSE.all)
str(mixedFactorial.df)
summary(mixedFactorial.df)

```

This example stores the data in a narrow format data frame. The first several lines again use *inspect()* to extract the desired data and the next line uses the *c()* function to concatenate these into a single vector called *iSE.all*. The next lines construct each of the desired variables -- case identification number, two independent variables and the dependent variable -- and arrange these into a data frame. The last two lines provide output to confirm the desired result.

```

> str(mixedFactorial.df)
'data.frame': 4000 obs. of 4 variables:
 $ Case      : int  1 2 3 4 5 6 7 8 9 10 ...
 $ NCondition: num  50 50 50 50 50 50 50 50 50 50 ...
 $ SECondition: Factor w/ 2 levels "ML","MLR": 1 1 1 1 1 1 1 1 1 1
1 ...
 $ iSE       : num  0.0901 0.0777 0.0794 0.0738 0.0616 ...
> summary(mixedFactorial.df)
      Case      NCondition  SECondition      iSE
Min.   : 1.0    Min.     : 50    ML :2000    Min.     :0.02050
1st Qu.: 500.8  1st Qu.: 50    MLR:2000   1st Qu.:0.02738
Median :1000.5  Median  :275                    Median  :0.03518
Mean   :1000.5  Mean    :275                    Mean    :0.05729
3rd Qu.:1500.2  3rd Qu.:500                    3rd Qu.:0.08614
Max.   :2000.0  Max.    :500                    Max.    :0.18905
>

```

The *str()* function confirms the expected data frame of 4000 cases and 4 variables. The first case has $n = 50$ and in the ML condition produces a standard error of 0.0901. The *summary()* function provides a summary of those four variables across all 4000 cases. The case numbers range from 1 to 2000, and the sample size independent variable ranges from 50 to 500. The data contains exactly 2000 cases analyzed using each type of standard error and the mean standard error was 0.05729.

A mixed-effects model summarizes the results, allowing for the differences in variance across sample-size conditions (Pinheiro & Bates, 2000). Note that because sample size was coded as

numeric, the two conditions differ by 450 resulting in a small number for the effect size estimate (an alternative would be to recode this independent variable as a factor or as a binary numeric variable).

```
> fit1 <- lme(fixed=iSE ~ NCondition * SECondition,
+  random= ~ 1|Case,
+  weights=varIdent(form= ~ 1 | NCondition),
+  data=mixedFactorial.df)
> summary(fit1)
Linear mixed-effects model fit by REML
Data: mixedFactorial.df
      AIC      BIC   logLik
-29655.34 -29611.29 14834.67

Random effects:
Formula: ~1 | Case
      (Intercept)  Residual
StdDev: 0.0008115548 0.01857034

Variance function:
Structure: Different standard deviations per stratum
Formula: ~1 | NCondition
Parameter estimates:
      50      500
1.00000000 0.08978166
Fixed effects: iSE ~ NCondition * SECondition
              Value      Std.Error   DF    t-value p-value
(Intercept)      0.09527837 0.0006530259 1998   145.90289 0.0000
NCondition      -0.00013570 0.0000013102 1998  -103.56590 0.0000
SEConditionMLR   -0.00285692 0.0009235181 1998   -3.09352 0.0020
NCondition:SEConditionMLR 0.00000553 0.0000018530 1998    2.98500 0.0029
Correlation:
              (Intr) NCndtn SECMLR
NCondition      -0.996
SEConditionMLR   -0.707  0.704
NCondition:SEConditionMLR 0.704 -0.707 -0.996

Standardized Within-Group Residuals:
      Min      Q1      Med      Q3      Max
-3.32089971 -0.66711247 -0.03089443  0.60595634  5.55081114

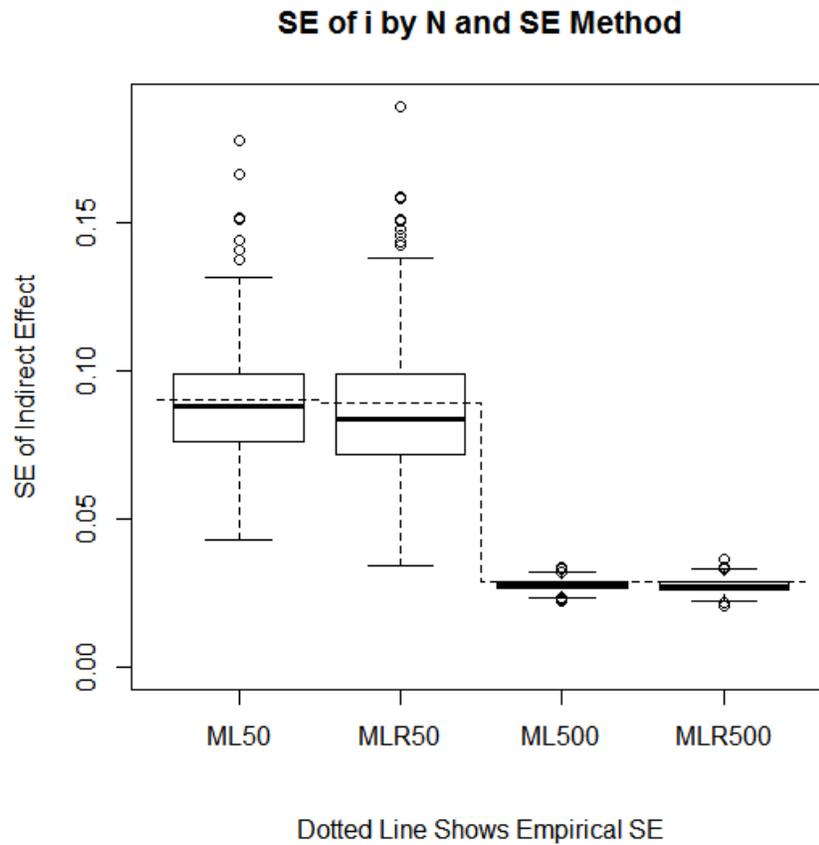
Number of Observations: 4000
Number of Groups: 2000
>
```

The box-and-whisker plot now has four conditions.

```
boxplot(x=list(
  ML50=mixedFactorial.df$iSE[NCondition==50 & SECondition=='ML'],
  MLR50=mixedFactorial.df$iSE[NCondition==50 &
SECondition=='MLR'],
  ML500=mixedFactorial.df$iSE[NCondition==500 &
SECondition=='ML'],
  MLR500=mixedFactorial.df$iSE[NCondition==500 &
SECondition=='MLR']),
  ylim=c(0, max(mixedFactorial.df$iSE))
)
title(main='SE of i by N and SE Method',
  ylab='SE of Indirect Effect',
  sub='Dotted Line Shows Empirical SE')

i.ML50 <- sd(inspect(simOutput_ML50, "coef", improper = TRUE,
nonconverged = TRUE)[,6])
i.MLR50 <- sd(inspect(simOutput_MLR50, "coef", improper = TRUE,
nonconverged = TRUE)[,6])
i.ML500 <- sd(inspect(simOutput_ML500, "coef", improper = TRUE,
nonconverged = TRUE)[,6])
i.MLR500 <- sd(inspect(simOutput_MLR500, "coef", improper = TRUE,
nonconverged = TRUE)[,6])
lines(
  x=c(0.5, 1.5, 1.5, 2.5, 2.5, 3.5, 3.5, 4.5),
  y=rep(c(i.ML50, i.MLR50, i.ML500, i.MLR500), each=2),
  lty=2)
```

Figure 4.
Mixed Factorial Design



Consistent with the effect size estimates, the effect of sample size appears much larger than that of se estimation method. The two methods appear more comparable with the larger sample size. Some underestimation appears present.

To further illustrate some of the available graphical tools in R, here is a more traditional line graph of the same results.

```
# Line graph as alternative
```

```
linePlotData <- cbind(
  ML50=mixedFactorial.df$iSE[NCondition==50 & SECondition=='ML'],
  MLR50=mixedFactorial.df$iSE[NCondition==50 &
SECondition=='MLR'],
  ML500=mixedFactorial.df$iSE[NCondition==500 &
SECondition=='ML'],
  MLR500=mixedFactorial.df$iSE[NCondition==500 &
SECondition=='MLR'])
yPointsSE <- colMeans(linePlotData) # Mean estimated SE of i
```

```

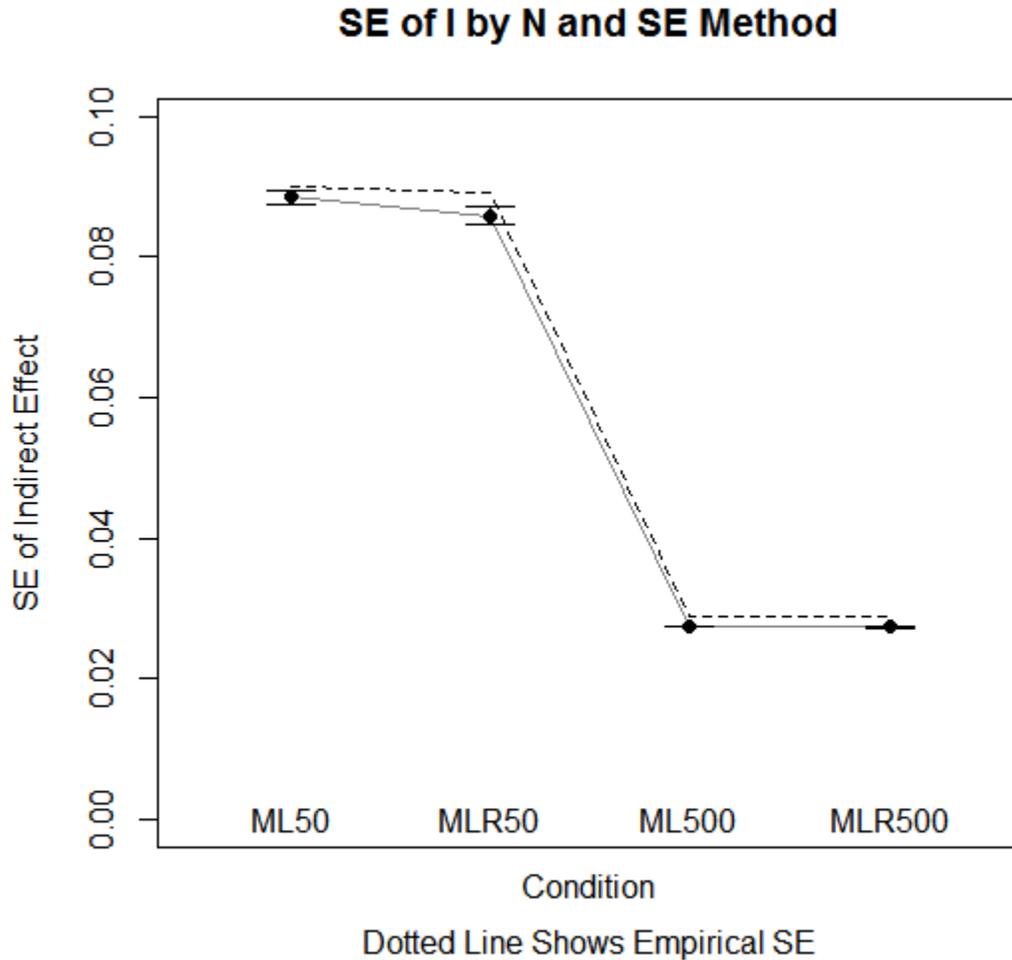
SEM <- apply(X=linePlotData, MARGIN=2, FUN=sd) /
  sqrt(apply(X=linePlotData, MARGIN=2, FUN=length)) # SE of Mean
yPointsUB <- yPointsSE + (1.96 * SEM) # 95% CI upper bound
yPointsLB <- yPointsSE - (1.96 * SEM) # 95% CI lower bound
yPointsSD <- c(i.ML50, i.MLR50, i.ML500, i.MLR500) # Empirical
SD of SE

plot(x=c(0.5,4.5), y=c(0, 1.1 * max(yPointsUB)), xaxt='n',
  type='n', xlab='', ylab='SE of Indirect Effect',
  main='SE of I by N and SE Method')
title(sub='Dotted Line Shows Empirical SE', line=2)
title(xlab='Condition', line=0.5)
lines(x=1:4, y=yPointsSD, lty=2) # dotted line
lines(x=1:4, y=yPointsSE, col=gray(.5)) # not to interfere with
error bars
arrows(x0=1:4, y0=yPointsLB, y1=yPointsUB, angle=90, code=3,
  length=.125) # error bars
points(x=1:4, y=yPointsSE, pch=16)
text(x=1:4, y=0, labels=dimnames(linePlotData)[[2]]) # condition
labels

```

The first block of code prepares all the values needed for the graph; the second block plots the graph. The generic `plot()` function with the parameter `type='n'` creates a blank plot. The `line` parameter to the `title()` function controls the spacing between text and the graph. The `lines()`, `arrows()`, `points()`, and `text()` functions layer on various elements of the graph. Order matters because each new element appears on top of the previous where plot elements intersect. It might seem natural to plot the points first and then add to them but it almost always works better to plot the elements in the reverse order, plotting the points last so that nothing obscures them.

Figure 5.
Line Plot



To summarize, like the two-group designs, the mixed factorial design simulation began by making the needed packages available, setting the random number seed, and defining the models used in the simulation. Like the within-subjects design, the next step involved generating the desired data sets and saving them. The difference was that this example saved two sets of data sets each in its own list. The next steps involved running each of the four conditions in the simulation. Finally the last steps extracted the desired outcomes, combined them into a narrow-format data frame, and used that data frame to evaluate the results.

From Procedural to Function-Based Coding

The present tutorial used a programming approach referred to as procedural programming. Each of the scripts follows a linear sequence of steps from beginning to end to complete a task. This is a simple, intuitive and transparent approach that offers a good starting place and works well

enough for relatively simple projects that are not likely to be repeated. This is the approach most data analysts and researchers begin writing code with, however, it has some important limitations. Procedural programming makes code more difficult to test because it is more difficult to break the code into independently testable pieces. Likewise, procedural code is more difficult to debug. Procedural code is often more difficult to edit, modify or maintain because it is not modular and often requires similar changes to be repeated in multiple places.

Once you gain some comfort with the methods illustrated in this tutorial, consider switching to programming with functions (Venables, Smith & R Core Team, 2015, Chapter 10). This approach has several advantages. You only need one piece of code to run all the conditions in your design. So, anything you change or fix only needs to be changed once rather than repeated in the code for each condition. You can then run the conditions by calling your home-made function with different parameter values. Another advantage is that you can break a complex task into subtasks each accomplished by a different function. That makes each function easier to test and debug. A master function can then call the simpler functions, or they can call one another, as needed. Finally, your functions can be saved as part of your personal library of code and reused for future projects.

To illustrate, the following lines code a function to generate a list of data sets such as the one used in the second example.

```
# Define function
dataList <- function(Nrep, Nobs, model, ...){
  DD <- vector('list', Nrep)
  for(rep in 1:Nrep){
    DD[[rep]] <- generate(model=model, n=Nobs)
  }
  return(DD)
}
```

The *function()* function creates a function as output which the first line saves as an R object named *dataList*. The parenthesis define the parameters for the new function. If desired, you can also define these with default values such as *nrep = 1000*. Form a habit of including "..." at the end of the list so that users can pass unanticipated parameters through our function to any functions that it might call. Remember that R is not static and you want your function to keep working as R changes with minimal upkeep. The lines of code inside the outer set of braces define the function to be executed when the new function is used. *DD* is a temporary R object that exists only in the scope of the function. Note that the first 'model' in *model=model* is a parameter to the *generate()* function whereas the second 'model' is the parameter to the *dataList()* function, assigning the value of the latter to the former. The *model* parameter to the *dataList()*

function could have taken a different name, they do not need to match. The *return()* function passes the list of data sets out of the function into the context in which it was called as the value returned by the function.

Once the function is saved in the workspace, you can call it like any other R function.

```
newData50 <- dataList(Nrep=1000, Nobs=50, model=simModel)
```

The file <Data Generation Function.R> provides self-contained code illustrating this function.

Conclusion

The basic techniques illustrated in these three examples can be combined and expanded to generate more complex designs with more levels for an independent variable or more independent variables. As the design becomes more complex, cells multiply, requiring more steps to run the simulation as a whole. However, the basic building blocks remain the same.

Feedback

Please send any feedback regarding this document to kmarkus@aol.com.

Acknowledgements

I am grateful to Amin Mousavi for suggesting an improvement to this tutorial.

References

- Boker, S. M., Neale, M. C., Maes, H. H., Wilde, M. J., Spiegel, M., Brick, T. R., Estabrook, R., Bates, T. C., Mehta, P., Oertzen, T. v., Gore, R. J., Hunter, M. D., Hackett, D. C., Karch, J., Brandmaier, A. M., Pritikin, J. N., Zahery, M., & Kirkpatrick, R. M. (2014) *OpenMx version 2.0: Multipurpose Software for Statistical Modeling*. University of Virginia, Department of Psychology, Box 400400, Charlottesville, VA 22904. <http://openmx.psyc.virginia.edu>
- Burns, P. (2011). *The R Inferno*. http://www.burns-stat.com/pages/Tutor/R_inferno.pdf
- Pinheiro, J. C. & Bates, D. M. (2000). *Mixed-effects models in S and S-Plus*. New York: Springer.
- Pornprasertmanit, S., Miller, P. & Schoemann, A. (2016). *simsem: SIMulated Structural Equation Modeling*. R package version 0.5-13. <https://CRAN.R-project.org/package=simsem>
- R Core Team (2015). *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria. URL <https://www.R-project.org/>.
- Rosseel, Y. (2012). lavaan: An R Package for Structural Equation Modeling. *Journal of Statistical Software*, 48, 1-36. URL <http://www.jstatsoft.org/v48/i02/>.
- Venables, W. N., Smith, D. M. & R Core Team (2015). *An introduction to R* (ver. 3.2.3). URL <https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf>