International Journal of Assessment Tools in Education
2018, Vol. 5, No. 4, 682–700

DOI: 10.21449/ijate.472185

Published at http://www.ijate.net http://dergipark.gov.tr/ijate *Research Article*

Automating Simulation Research for Item Response Theory using R

Sunbok Lee [1,*], **Youn-Jeng Choi** [2], **Allan S. Cohen** [3]

[1] Department of Psychology, University of Houston, Houston, TX 77204
[2] Department of Educational Studies in Psychology, Research Methodology & Counseling, The University of Alabama, Tuscaloosa, AL 35487
[3] Department of Educational Psychology (Quantitative Methodology), University of Georgia, Athens GA 30602

ARTICLE HISTORY

Received: 22 August 2018
Revised: 12 October 2018
Accepted: 17 October 2018

KEYWORDS

IRT, Simulation, R

Abstract: A simulation study is a useful tool in examining how validly item response theory (IRT) models can be applied in various settings. Typically, a large number of replications are required to obtain the desired precision. However, many standard software packages in IRT, such as MULTILOG and BILOG, are not well suited for a simulation study requiring a large number of replications because they were developed as a stand-alone software package that is best suited for a single run. This article demonstrated how built-in R functions can be used to automate the simulation study using the stand-alone software packages in IRT. For a demonstration purpose, MULTILOG was used in the example codes in the appendices, but the overall framework of a simulation study and the built-in R functions used in this article can be applied for a simulation study using other stand-alone software packages as well.

1. INTRODUCTION

Item response theory (IRT) provides a family of statistical models that establish the correspondence between item responses and latent variables (De Ayala, 2009). When applying IRT models to real datasets, researchers often want to know how validly their IRT models can be applied to their datasets at hand (Harwell, Stone, Hsu, & Kirisci, 1996). For examples, researchers may concern the small sample sizes or multidimensionality of their datasets. Analytic solutions can provide exact and rigorous answers to those questions, but may not be always tractable because of the complexity of the problem. In such a case, a simulation study can be a useful alternative.

Typically, a simulation study requires a large number of replications to obtain the desired precision. However, traditional standard software packages in IRT, such as MULTILOG (Thissen, Chen, & Bock, 2003) and BILOG (Zimowski, Muraki, Mislevy, & Bock, 1996), are not well suited for a simulation study requiring a large number of replications because they were developed as stand-alone software packages for a single run. Running such stand-alone software packages requires input command files, which make it difficult to run those software

CONTACT: Sunbok Lee slee95@Central.UH.EDU Department of Psychology, University of Houston, Houston, TX 77204

ISSN-e: 2148-7456 /© IJATE 2018

packages a large number of times. More importantly, it is very difficult to extract the values of interest, such as parameter estimates and fit indices, from the output files. It would be an extremely time-consuming task to manually extract the values from a large amount of output files. Therefore, for a simulation study using stand-alone software packages, it is necessary to automate the simulation procedure.

Recently, many R packages, such as LTM and MIRT, have been developed for IRT. With those packages, a simulation study can be made much simpler by repeatedly running IRT estimation functions within a loop and also extracting values of interest from R objects. However, there is still need for conducting a simulation using stand-alone IRT software packages. Those stand-alone IRT software packages, such as MULTILOG and BILOG, have been widely used in psychometrics, and still are considered as standard software packages by many researchers. Although there are other options such as R and Mplus, MULTILOG and BILOG are still widedly used for IRT simulation studies on various topics such as linking (Kim & Lee, 2006), goodness of fit statistics (Stone, 2000), initial valud and convergence criterian (Nader, Tran & Voracek, 2015), speicification of ability distribution and numerical integration (Kim & Lee, 2017), and structure zeros (Kim, Brennan, & Lee, 2017). Moreover, when researchers want to develop new estimation algorithms, they may want to compare the results from their new algorithms with those from traditional stand-alone software packages. Given the typical recommendation for the large replications, using the standalone software packages such as MULTILOG and BILOG for simulation studies require us to automatically run the software packages a large number of times and also automatically extract necessary information from a large number of output files. The purpose of this paper is to demonstrate how the built-in functions in R software package (R Core Team, 2015) can be used to automate a simulation study using standard stand-alone software in IRT. The procedure for a simulation study can be divided into four parts: generating item responses, preparing input command files, running stand-alone software, and extracting statistics from output files. Some functions in R that are useful in automating each part of a simulation study are introduced. For a demonstration purpose, MULTILOG is used in the example code in the appendices, but the framework of the simulation study and the R functions used in this article can also be applied for a simulation study using other stand-alone software.

2. A SIMULATION STUDY IN IRT

A simulation study is a computer intensive procedure to evaluate statistical methods, given a known model and parameters. Most frequently, this is done as a Monte Carlo simulation study in which random numbers are used to generate data to be analyzed with the model. In a simulation study, the random samples thus generated enable us to create an empirical sampling distribution of the parameters under the conditions in which researchers are interested (Bandalos, 2006). The empirical sampling distribution can be used in evaluating the estimating procedures (e.g., maximum likelihood estimation) or properties of the statistics (e.g., goodness of fit). Harwell et al. (1996) provides a useful reference for a simulation study in IRT, illustrating advantages, limitations, and major steps of IRT simulation. The simulation steps consist of formulating a problem, designing a simulation study, generating item responses and estimating model parameters, and analyzing the results of a simulation. In this paper, we present how some steps of a IRT simulation study can be automated using the R software package.

In most simulation studies, handling a large number of replications is a typical problem. Use of stand-alone software, such as MULTILOG and BILOG, to estimate IRT parameters can pose some additional difficulties that are not present when implementing the whole simulation procedures within a single platform such as C, FORTRAN, or R. Within a single programming language, the intermediate results produced in a given step can be easily used by the following

step of the simulation. For example, in a simulation study in which the whole process is implemented within R, the generated data sets can be saved as array variables which are then passed to the next step in the process. Using the generated data sets in the later step of the simulation is just a matter of referencing these variables within R. The estimated parameters are also saved as array variables in R, making it easy to summarize the results of parameter estimation.

In contrast to using a single platform of programming language, using stand-alone software for a simulation study poses some practical problems, as such software is typically designed for a single run in which the input command file needs to be provided by the user. Given an input command file, results are usually saved as a text file. In order to obtain the desired number of replications using stand-alone software, it is necessary to prepare data and input command files for as many replications as needed. Furthermore, when the subsequent output files are produced, statistics of interest such as parameter estimates, standard errors, and fit indices need to be extracted from output files and organized into structured data for summary. Considering the large number of replications required for a simulation study, it is not normally practical to handle these procedures manually. In the following sections, we describe how some built-in functions in the R software package can be used to automate the simulation study using stand-alone IRT software.

3. R FOR THE SIMULATION STUDY IN IRT

R is a freely available open source language for statistical computing (R Core Team, 2015) and can be used to automate the simulation using stand-alone software. R has built-in functions which can manipulate character strings and run external stand-alone software. R also supports the regular expressions which are useful for extracting the statistics of interest from output files. These functions of R will be discussed in this section, and the example code in the appendices that was written using these functions will be discussed in detail in the next section.

3.1. Functions for String Manipulations in R

Stand-alone software is typically developed for a single run in which data and input command files need to be prepared by a user. In order to obtain the desired number of replications, stand-alone software should be run as many times as the number of replications using data and input command files specific to each replication. Therefore, to automate the procedure for preparing those files, file names and some commands in the input command file need to be automatically modified from replication to replication. In R, paste() and strsplit() are functions for string manipulations and can be used to modify the character strings for file names and commands. The gsub() is also useful in replacing a string in a text.

paste(). In R, the paste() function concatenates or combines an arbitrary number of arguments to form a combined string after converting each argument to a character string. For example, in the following R code, the paste() function combines character string ‘item’ and the number 7 to form the string ‘item7’. The sep option is used to specify the character string to separate the arguments. In the following example, the sep option is used to indicate no space by placing two double quotes together, with no space between them:

```
> paste("item", 7, sep="") [1] "item7"
```

A character vector which contains item names can be generated easily by using a numeric vector. In the example code below, the numeric vector is generated by 1:7.

```
> paste("item", 1:7, sep="")
```

```
 [1] "item1" "item2" "item3" "item4" "item5" "item6" "item7"
```

The paste() function is useful for changing text, such as file names and commands, in command files. The following example shows how the data file name can be changed from replication to replication using the paste() function within a for loop.

```
for (replications in 1:1000) {
      ...
filename <- paste("MULTILOGI20S100R", replications,
            ".dat", sep="")
}
```

In the code above, replications is a for loop index variable that varies from replication to replication. If replications changes from 1 to 1000 , the variable filename will be changed from 'MULTILOGI20S100R1.dat' to 'MULTILOGI20S100R1000.dat'.

strsplit(). The strsplit() function splits a character string into substrings using a delimiter specified in the split option. It provides a convenient way for users to separate the file extension from the whole file name. Since a simulation using stand-alone software generates a large number of data, input command, and output files, it is necessary to manage a large number of files, each with separate names. In the examples in this paper, we have adopted the following naming rule: The body of the whole file name remains the same and only the file extension is changed for each of the files needed for each replication. For example, in the data file name 'MULTILOGI20S100R1.dat', MULTILOG is the body of the file name and does not change. I20 is used to indicate 20 items, S100 is used to indicate 100 subjects, and R1 is used to indicate the 1st replication. MULTILOG requires the file extension '.mlg' to indicate the input command file to the software. So, the name for the input command file for a 20-item test with 100 examinees would be 'MULTILOGI20S100R1.mlg'. In the R code below, the strsplit() function splits a character string 'MULTILOGI20S100R1.dat' into two substrings using a dot as a delimiter to form a character vector containing two elements, 'MULTILOGI20S100R1' and 'dat'.

```
> strsplit("MULTILOGI20S100R1.dat", split=".", fixed=T)

[1] "MULTILOGI20S100R1" "dat"
```

Combined with the paste() function discussed above, the strsplit() function can be used to change only the file extension of the whole file name. In the code below, the unlist() function is used to convert the list data type produced by the strsplit() function to the vector data type, and '[1]' is used to index the first element of a character vector, which is 'MULTILOGI20S100R1'. In all, the following R code can be used to replace the file extension '.dat' with '.mlg'.

```
> paste(unlist(strsplit("MULTILOGI20S100R1.dat",

+    split=".",fixed=T))[1],".mlg",sep="")

[1] "MULTILOGI20S100R1.mlg"
```

gsub(). The gsub() function replaces a string in a vector. More specifically, gsub(pattern, replacement, x) finds the string pattern in a vector x and replaces the string pattern with another

string replacement. For example, in the following example, gsub() replaces 'apple' with 'orange' in the vector x.

```
> x <- "I like apple"
> gsub("apple", "orange", x)
[1] "I like orange"
```

In a simulation study, the gsub() function can be used to prepare an input command file specific to each replication using a template file. For example, in MULTILOG, the number of examinees needs to be provided in an input command file. Suppose that a template file to generate the input command file contains the following line specifying the number of examinees: 'NEXAMINEES =hnSubjectsi' . Then, the R command below will replace hnSubjectsi with the specific number, 10:

```
> x <- "NEXAMINEES = <nSubjects>"
> gsub("<nSubjects>", 10, x)
[1] "NEXAMINEES = 10"
```

3.2. Functions for Executing External Software

Once data and input command files are prepared, researchers need to run stand-alone software the desired number of times to generate the output files. Running stand-alone software in batch mode is a useful way of automating such a process. In batch mode, users can run software without manual intervention by simply providing a command at the DOS prompt. For example, the following command at the DOS prompt will execute MULTILOG using 'MULTILOGI5S500R9.mlg' as an input command file. Note that the file extension need to be omitted. Also, progra~1 is the short name for 'Program Files' folder in the Windows operating system.

```
C:\progra~1\MULTILOG> mlg MULTILOGI5S500R9
```

Since our goal is to automate the procedure for running external software, the command above needs to be provided to the DOS operating system using R by changing the input command file from replication to replication. This can be done using the system() function in R, which can execute the system command from R. The following R code executes MULTILOG using the 'MULTILOGI5S500R9.mlg' as an input command file. Note that, in R, the double backslash represents backslash.

```
> system("C:\\Progra~1\\MULTILOG\\mlg MULTILOGI5S500R9")
```

3.3. A Regular Expression in R

Given a large number of output files, the key in the next step of the simulation is to extract the statistics of interest from the output files and to organize them into structured data. In order to extract specific information from the output files, it is often necessary to use a more sophisticated expression than a simple string. A regular expression is a powerful tool for extracting the statistics of interest from output files.

An Example of the Regular Expression. Suppose that a researcher is using BILOG for a simulation, and the following is the part of a BILOG phase 2 output file named 'BILOGI20S200F2.PH2'.

```
  ITEM      INTERCEPT    SLOPE    THRESHOLD   LOADING   ASYMPTOTE   CHISQ
              S.E.       S.E.       S.E.       S.E.       S.E.     (PROB)
-------------------------------------------------------------------------


ITEM0001 | 0.292    |  2.090   | -0.140   |  0.902  | 0.000  |    4.1
         | 0.187*   |  0.339*  |  0.088*  |  0.146* | 0.000* | (0.3879)
         |          |          |          |         |        |
ITEM0002 | 0.043    |  0.802   | -0.053   |  0.626  | 0.000  |    2.9
         | 0.154*   |  0.180*  |  0.192*  |  0.140* | 0.000* | (0.8175)
         |          |          |          |         |        |
...
```

A researcher might want to extract thresholds or difficulty parameters from the output file. In the file above, the thresholds of item 1 and item 2, which are labeled ITEM0001 and ITEM0002, are -0.140 and -0.053 respectively. One way of extracting those numbers from the output files could be the following: i) extracting the lines that contain ITEM0001, ITEM0002, etc., and ii) extracting the numbers in the fourth column from those lines. In identifying the lines for extraction, it would be impractical to use a simple string such as ITEM0001, ITEM0002, etc. because a researcher needs to specify each string representing each item. Rather, it would be more practical if a researcher can specify the string for matching more flexibly like ITEMdddd, where d represents a single digit number. Also, in extracting thresholds, it would be convenient if there is a single expression representing any floating numbers to match -0.140 and -0.053.

A regular expression is the sequence of literal and special characters that describes a set of strings and provides a flexible means for matching strings in a text. In the example above, the lines that contain ITEM0001, ITEM0002, etc. can be matched using the regular expression ITEM[0-9]{4} in which [0-9] matches any single digit, and {4} matches 4 occurrences of the preceding element. Also, in R, the regular expression '[-+]?[0-9]+(\\.[0-9]+)?' matches a floating number. In a regular expression, the square bracket [] is used to match any single character listed within the bracket. For example, '[hs]eat' matches 'heat', 'seat', etc. and '[0-9]' match any single digit. Therefore, the leading [-+] allows a plus or minus sign. Since ? matches the preceding element zero or more times, a number with no sign will also be matched. [0-9]+ matches one or more digits since + matches the preceding element 1 or more times. The double backslash \\ is used for the escape sequence. In programming languages including R, many characters are reserved to represent a special meaning. Sometimes, however, users might want to use those special characters as literal strings. The dot is escaped from the special meaning by using the double backslash and just represents the decimal point in a floating number. ()? makes \\.[0-9]+ optional. In all, the regular expression, '[-+]?[0-9]+(\\.[0-9]+)?', matches any floating number such as -0.140 and -0.053. The actual R code for extracting the thresholds from 'BILOGI20S200F2.PH2' will be presented later in this section with some additional functions in R using the regular expression.

The Regular Expression in General. A regular expression is the sequence of literal and special characters that is used to match a specific string in a text. By using a regular expression, a user can locate the specific lines in output files by matching the string described in the regular expression with a string in the lines in a file. This, in turn, enables a user to extract the specific information matching that regular expression from unstructured data sources. Simply, the regular expression is a sequence of characters that forms a search pattern for matching. For example, the regular expression 'a.c' matches 'aac', 'abc', etc. since a dot character in a regular expression matches any single character. Also, the regular expression 'ab+' matches 'abb', 'abbb', etc. since the + character in a regular expression matches the preceding element, which is 'b' in this example, one or more times.

The regular expression is supported by many programming languages (e.g., R, SAS, C++, and Java) and command line utilities in UNIX (e.g., grep, sed, and awk) because of its usefulness in many applications. The wide applicability of the regular expression comes from its flexibility in matching a specific string of text using special characters. Some frequently used special characters are introduced with examples in Table 1. For more detailed discussion on the regular expression, readers are referred to other comprehensive literatures (Friedl, 2006; Spector, 2008).

Table 1. Special Characters in a Regular Expression

Character	Meaning	Notes
.	Matches any single character	'a.c' matches 'abc', 'acc', etc.
^	Start of string or line	'^app.e' matches apple but only at the beginning of the string or line.
$	End of string or line	'app.e$' matches apple but only at the end of the string or line.
[ ]	Matches any single character that is listed within the brackets	'[hs]eat' matches heat, seat, etc.
[^]	Matches any single character that is not listed within the brackets	[^hs]eat matches neat, beat, etc.
?	Matches the preceding element 0 or 1 times	'ab?' matches 'a' or 'ab', etc.
*	Matches the preceding element 0 or more times	'ab*' matches 'a' or 'ab' or 'abb', etc.
+	Matches the preceding element 1 or more times	ab+ matches ab or abb or abbb, etc.
\w	Alphanumeric characters	Equivalent to [A-Za-z0-9]
\W	Non alphanumeric characters	Equivalent to [^A-Za-z0-9]
\d	Digits	Equivalent to [0-9]
\D	Non-digits	Equivalent to [^0-9]
\s	White space characters	Matches space, tab, etc.
\S	Non white space characters	Matches anything except white space

R has several built-in functions that can be used in extracting the statistics of interest from output files using the regular expression. In the following, examples of the regular expression and functions in R supporting the regular expression will be presented.

grep(). The grep() function searches a file line by line and returns only those lines that match a particular pattern. The pattern is allowed to be a simple character string or the regular expression. For example, the following R command extracts only the lines that contain

ITEM0001, ITEM0002, etc. from the BILOG output file named 'BILOGI20S200F2.PH2' in the previous example.

```
> text <- readLines("BILOGI5S500R1.PH2")

> lines <- grep("ITEM[0-9]{4} \\|", text, value=TRUE)

> lines

 [1] " ITEM0001 | 0.292 |    2.090 | -0.140 |     0.902 |    0.000 | "
 [2] " ITEM0002 | 0.043 |    0.802 | -0.053 |     0.626 |    0.000 | "
 ...
```

In the example code above, the readLines() function reads 'BILOGI20S200F2.PH2' line by line and saves those lines in the character vector named text. Then, the grep() function returns the lines containing a character string that match the pattern described by the regular expression "ITEM[0-9]{4} \\|" from the character vector text and saves those lines in the character vector lines.

str_extract_all(). The str_extract_all() function extracts all pieces of a string that match a pattern. The user will need to ensure that the stringr package is installed to use this function. Given the vector named lines in the previous example, the floating numbers in the vector lines can be extracted using the following str_extract_all() function:

```
> numbers <- str_extract_all(lines, "[-+]?[0-9]+(\\.[0-9]+)?}")

> numbers

[[1]]

[1] "0.292" "2.090" "-0.140" "0.902" "0.000" "4.1"  "4.0"

[[2]]

[1] "0.043" "0.802" "-0.053" "0.626" "0.000" "2.9"  "6.0

…
```

Where [-+]?[0-9]+(\\.[0-9]+)? is the regular expression representing a floating number. Note that the result from the str_extract_all() function is a list object which is one of the data types in R. A list object has elements, each of which can contain any type of object in R. Since a matrix is easier to handle, the do.call() function is used to convert the list object to the matrix object as follows:

```
> numbers <- do.call("rbind", numbers)

> numbers

      [,1]    [,2]    [,3]     [,4]    [,5]    [,6]  [,7]

[1,] "0.292" "2.090" "-0.140" "0.902" "0.000" "4.1" "4.0"

[2,] "0.043" "0.802" "-0.053" "0.626" "0.000" "2.9" "6.0"

...
```

The do.call() function executes a function, which is the rbind() here, using a list of arguments to be passed to it. Since the rbind() combines vectors by rows, the do.call("rbind", numbers) command converts the list object numbers to a matrix object.

4. EXAMPLE CODE

In the previous section, we have discussed some useful functions of R that can be used to automate the simulation using stand-alone software. This section presents an example of a simple simulation study using those functions. The R codes for this example are presented in the appendices. In the example, item responses were generated using two-parameter logistic (2PL) model (Appendix A), input command files for MULTILOG were generated using a template file for the input command file (Appendix B), MULTILOG was automatically run as many replications as needed (Appendix C), and item parameters of 2PL were extracted and summarized using functions in R (Appendix D).

4.1. Setup and Data Generation

Appendix A provides the R codes that generate data sets for a simple example simulation study. The following information is required for this simulation: the number of items, the number of subjects, the number of replications, and the true item parameters for the 2PL model. All the files and results from this simulation will be saved in the folder specified by mF variable. The user also needs to specify the folder where MULTILOG is installed using multilogPath variable.

Item responses are generated in lines 29-39. The ability of each person is sampled from a standard normal distribution, and the probability of getting an item correct is calculated based on the equation for 2PL. Then, given a random number between 0 and 1 that is sampled from a uniform distribution, the item response for an item of a person becomes 1 if the sampled uniform random number is less than the probability of getting an item correct for a person and becomes 0 otherwise.

4.2. Preparing Input Command Files for MULTILOG

A template file and R codes for preparing input command files for MULTILOG are provided in Appendix B. A template file is used to generate the input command file for each replication by changing some parts of the template file to be specific for each replication. More specifically, a template file named ‘template.mlg’ contains MULTLOG commands for 2PL. Some parts of the template file are not specified and just marked using angle brackets such as <datafile> and <nItems>, which will be replaced with appropriate character strings using the gsub() function.

In line 5 of the R code in the second part of Appendix B, the list.files() function is used to read all the data file names in the main folder (mF). In line 11, each line of the template file is saved into text variable using readLines() function. In line 13, the name of data file for a specific replication represented by rep is saved into the variable dName. Then, in line 14, the string <datafile> in the template file is replaced with the specific file name, i.e., dName, for a given replication using the gsub() function.

```
5  dataFiles <- list.files(pattern = ".dat", file.path(mF))

...

11 text <- readLines("C:/template/template.mlg")

...

13 dName <- file.path(mF,dataFiles[rep])
```

```
14 text <- gsub(pattern = "<datafile>", replacement = dName, x = text)

...
```

The input command file in MULTILOG requires a string representing the format of a data file. In line 23, paste("(5A1,", nItems,"A1)", sep="") creates the string for the format of the data file. In our example, the number of items is 5, and therefore the string (5A1, 5A1) is saved into format variable. In line 24, the string <format> in a template file is replaced by the string saved in format.

4.3. Running MULTILOG in R

Given the data and input command files, the code in Appendix C runs MULTILOG as many replications as needed. In the previous section, we have introduced the system() function, and the following command executes MULTILOG using the input command file named 'MULTILOGI5S500R9.mlg':

```
> system("C:\\Progra~1\\MULTILOG\\mlg MULTILOGI5S500R9")
```

In order to repeatedly run MULTILOG many times , the system() function can be used within a for loop by changing the argument of the system() function using the paste() function.

```
...

dataFiles <- list.files(pattern = ".dat", file.path(mF))

...

for (rep in 1:nReplications) {

      system(paste(multilogPath,unlist(strsplit(dataFiles[rep],"\\.
             "))[1]))

}

...
```

In the codes above, the vector object dataFiles contains the names of all data filesin the main folder (mF). The for loop index variable rep changes from 1 to nReplications, and accordingly dataFiles[rep] indicates a different data file name for each replication. The strsplit() function is used to remove the file extension from the data file name, and the paste() function is used to combine the string for the path of MULTILOG and data file name.

4.3. Extracting and Summarizing Results

In the codes in Appendix D, item discrimination and difficulty parameters are extracted from output files and organized into structured format. It should be noted that the codes in Appendix D are just one way of extracting and summarizing parameters. Also, codes could be more complicated as the simulation becomes more complex. However, the functions used in this example would be useful in extracting statistics of interest from output files in other cases.

In Appendix D, the lapply() function is used through the codes. The lapply(x,fun), where the x is a list object and fun is any function in R, returns a list object of the same length as the list object x, each element of which is the result that can be obtained by applying the function fun to the corresponding element of x. Because of the implicit iterative nature of the lapply()

function, it can replace the for loop statement in R, which makes a code more simple and clear. The following simple example may be helpful in understanding the lapply() function:

```
> results <- lapply(1:3, function(x) x+1)

> results

[[1]]

[1] 2

[[2]]

[1] 3

[[3]]

[1] 4


> for (x in 1:3) {

      print(x+1)

  }

[1] 2

[1] 3

[1] 4
```

In the above example, the lapply() function picks up the first element of a vector 1:3, which is 1, and uses that element as an argument of the function(x), which is defined as x+1, to yield 2. Then, the result, which is 2 here, is saved as the first element of the list object results. The same procedure is repeated for the second and third elements of a vector 1:3. The same results can be obtained using a for loop. In this way, the lapply() function can replace a for loop, which makes codes more concise. In line 5 in Appendix D, the lapply() function is used to read output files:

```
15 outPut1 <- lapply(list.files(pattern=".OUT"), readLines)
```

In the code above, list.files(pattern=".OUT") creates a list object that contains file names having the file extension '.OUT'. Then, the lapply() function picks up the output file name from the list object one by one and use it as an argument of the readLines() function. In all, outPut1 is the list object, each element of which is the character vector containing each line of an output file as its element. For example, if there are 10 output files, each of which has 100 lines, then outPut1 would be the list object with 10 elements, each of which contains a character vector of size 100.

In line 13, the grep() function is used to extract the lines in output files that contain the discrimination parameters. For example, the following is a part of an output file from MULTILOG which contains the estimates for item discrimination of difficulty parameters:

```
 ITEM  1:    2 GRADED CATEGORIES
       P(#)   ESTIMATE (S.E.)
 A     1       1.06 (0.20)
 B( 1) 2      -1.76 (0.29)
```

Note that the estimates for discrimination and difficulty parameters are located at the second and third lines after the line that contains the string 'GRADED CATEGORIES'. To pick up the estimate for the discrimination parameter, which is 1.06 in this example, the following command in line 13 can be used:

```
13 outPut2<-lapply(outPut1,function(x) x[grep("GRADED CATEGORIES", x)+2])
```

In the code above, the lapply() function picks up each element of the list object outPut1, which is the character vector containing each line of an output file as its element, and uses it as an argument of the function defined by the grep() function. Hence, the argument x in the grep() function will be the character vector containing each line of an output file. Then, grep("GRADED CATEGORIES",x) will return the indices of the lines that matches the pattern 'GRADED CATEGORIES'. Also, grep("GRADED CATEGORIES",x)+2 will return the indices of the second lines after the line that match the pattern, and finally x[grep("GRADED CATEGORIES",x)+2]) will return the character strings in those lines. The following is the last part of the hypothetical results saved in outPut2 assuming 5 items and 100 replications:

```
> outPut2

> outPut2 [[1]]

 [1] " A     1      0.78   (0.19)" " A     3      1.55   (0.22)"
 [3] " A     5      0.53   (0.14)" " A     7      0.67   (0.15)"
 [5] " A     9      0.35   (0.13)"
 ...
 [[100]]
 [1] " A     1      1.47   (0.25)" " A     3      0.67   (0.15)"
 [3] " A     5      0.86   (0.15)" " A     7      0.44   (0.13)"
 [5] " A     9      0.57   (0.15)"
```

Note that each element of the list object outPut2 is the character vector containing the lines that contain estimates for discrimination parameters. Since the results in outPut2 are grouped by replication, it would be more convenient to sort the results by items. The following command in line 15 sorts the results by items:

```
15 outPut3<-lapply(1:nItems, function(x)unlist(lapply(outPut2,function(y)
                   y[x])))
```

Note that the lapply() function is used twice, which is much simpler than the nested for loop. The following is a part of the list object outPut3:

```
...

[[4]]

   [1]  " A   7      0.67   (0.15)" " A     7     0.07 (0.12)"

    ...

   [99] " A   7      0.59   (0.13)" " A     7     0.44 (0.13)"

[[5]]

    [1]  " A   9      0.35   (0.13)" " A     9      0.32   (0.14)"
     ...
    [99]" A   9      0.69   (0.15)" " A     9      0.57   (0.15)"
```

Given the list object outPut3, the code in line 18 will extract the numbers for the estimates of discrimination parameters as follows:

```
18 outPut4 <- sapply(outPut3, function(x)

+ as.numeric(str_extract(x,"[-+]?[0-9]+\\.[0-9]+")))

> outPut4

       [,1] [,2] [,3] [,4] [,5]

[1,]   0.78 1.55 0.53 0.67 0.35

...

[99,]  0.79 1.48 0.89 0.59 0.69

[100,] 1.47 0.67 0.86 0.44 0.57
```

Exactly the same strategy can be applied in obtaining estimates for item difficulty parameters and codes are provided in lines 22-32 in Appendix D. In lines 47-52, the same strategy is used to check whether output files contain the string 'NORMAL PROGRAM TERMINATION', which can provide the information about the convergency of model estimation for each replication

5. DISCUSSION

A simulation study is useful in investigating the behavior of IRT models in various settings. For a simulation study in IRT, traditional stand-alone software is often preferred because of its efficiency and reliability. In order to get the desired number of replications, it is almost essential to automate the simulation procedures. In this article, it was demonstrated how the R package can be used to automate a simulation study using stand-alone software in IRT.

Typically, a simulation using stand-alone software can be divided into four parts: generating data sets, preparing input command files, running software, and summarizing output files. Generating data sets is directly related to the research question of interest. In this article, we only presented the most simple case of item response generation. In preparing input command

files and summarizing output files, skills for handling text files are important. The R package provides many useful functions for text processing. In R,

character strings can be combined, split, and replaced using the paste(), strsplit(), and gsub() functions, respectively. Also, the grep(), str_extract_all(), and str_extract() functions are useful in extracting the statistics of interest from output files. In extracting the required information from output files, the regular expression is useful in forming flexible search patterns for matching. The system() function is also useful in running stand-alone software many times from R.

A couple of comments on some practical issues in a simulation study might be helpful. The estimation for a given replication could fail to converge. For the non-converged cases, researchers can exclude those cases from the summary statistics or re-run them using different starting values or different random samples (Harwell et al., 1996). Including some codes for checking the convergency of each replication might be useful. In our example codes, the convergency of the estimation for each replication was checked using the codes in lines 46-52 in Appendix D. On the other hand, printing some information about the current progress of a simulation might be helpful when the simulation study takes a long time. For example, IRT models can be formulated using the hierarchical generalized linear model, which can be estimated using the lme() function in the R package lme4. Sometimes, it may be necessary to use both stand-alone software and the lme() function. Usually, based on our experience, it took at least a couple of weeks to run those kinds of simulation because of the heavy computation in the lme() function. In that case, simply printing current values of the iteration could be helpful for checking the current status of a simulation. In our example, putting print(rep) within a for loop is enough to check the current status of a simulation.

Many useful R packages for IRT have been developed. For example, the R package irtoys (Partchev, 2009) provides the function est(), which can easily run BILOG and extract item parameters from an output file. If those packages can provide enough information for a simulation study, it would be more efficient to use built-in functions in those packages. However, sometimes, other information that is not summarized by the existing built-in functions may be necessary for a simulation study. More importantly, to our knowledge, there are not many R packages or other software that can help the automation of a simulation study using stand-alone software in IRT. In this paper, we have presented a framework of automating a simulation using stand-alone software and introduced some useful R functions that can be used in automating the simulation procedures. Also, an example of automating a simple simulation study using MULTILOG is provided in the appendices

ORCID

Sunbok Lee http://orcid.org/0000-0002-0924-7056

6. REFERENCES

Bandalos, D. L. (2006). The use of monte carlo studies in structural equation modeling research. In Structural equation modeling: A second course (pp. 385–426).

Greenwich, CT: Information Age.

De Ayala, R. J. (2009). *Theory and practice of item response theory*. New York, NY: The Guilford Press.

Finch, H. (2008). Estimation of item response theory parameters in the presence of missing data. *Journal of Educational Measurement*, *45*, 225–245.

Friedl, J. (2006). *Mastering regular expressions*. Sebastopol, CA: O'Reilly Media, Inc.

Harwell, M., Stone, C. A., Hsu, T.-C., & Kirisci, L. (1996). Monte carlo studies in item response theory. *Applied Psychological Measurement*, 20, 101–125.

Kim, H. J., Brennan, R. L., & Lee, W. C. (2017). Structural Zeros and Their Implications With Log-Linear Bivariate Presmoothing Under the Internal-Anchor Design. *Journal of Educational Measurement*, *54*, 145-164.

Kim, K. Y., & Lee, W. C. (2017). The Impact of Three Factors on the Recovery of Item Parameters for the Three-Parameter Logistic Model. *Applied Measurement in Education*, *30*, 228-242.

Kim, S., & Lee, W. C. (2006). An Extension of Four IRT Linking Methods for Mixed-Format Tests. *Journal of Educational Measurement*, *43*, 53-76.

Nader, I. W., Tran, U. S., & Voracek, M. (2015). Effects of Initial Values and Convergence Criterion in the Two-Parameter Logistic Model When Estimating the Latent Distribution in BILOG-MG 3. *PloS one*, *10*, e0140163.

Partchev, I. (2009). irtoys: Simple interface to the estimation and plotting of irt models. R package version 0.1, 2.

R Core Team. (2015). *R: A language and environment for statistical computing* [Computer software manual]. Vienna, Austria. Retrieved from http://www.R-project.org/ (ISBN 3-900051-07-0)

Reckase, M. D. (1979). Unifactor latent trait models applied to multifactor tests: Results and implications. *Journal of Educational and Behavioral Statistics*, *4*, 207–230.

Spector, P. (2008). *Data manipulation with r*. New York, NY: Springer.

Stone, C. A. (2000). Monte Carlo based null distribution for an alternative goodness-of-fit test statistic in IRT models. *Journal of Educational Measurement*, *37*, 58-75.

Thissen, D., Chen, W.-H., & Bock, R. D. (2003). Multilog 7 for windows: Multiple-category item analysis and test scoring using item response theory [computer software]. lincolnwood, il: Scientific software international. IL: Scientific Software International.

Zimowski, M. F., Muraki, E., Mislevy, R. J., & Bock, R. D. (1996). Bilog-mg: Multiple-group irt analysis and test maintenance for binary items. *Chicago: Scientific Software International*, *4*(85), 10.

Appendix A

Setup and Data Generation

```
# load R packages to use some functions in this code
library(gdata) # required for 'write.fwf()' function
library(stringr) # required for 'str_extract()' function
#######################################################################
# setting simulation conditions #
#######################################################################
# data, input files, and output files will be saved
# in the main folder (mF) below.
mF <- "C:/I5S500"
nReplications <- 100 # set the number of replications
nItems <- 5 # set the number of items
nSubjects <- 500 # set the number of subjects
# set the true item parameters for the 2PL model.
# a = discrimination parameters.
# b = difficulty parameters.
a <-c(1,1,1,0.5,0.5)
b <-c(-2,-1,0,1,2)
# specify the folder where MULTILOG is installed.
# 'Progra~1' is a short name for 'Program Files' folder
# in the Microsoft Windows operating system.
# \\ indicates escape sequence for \ in R.
multilogPath <- "C:\\Progra~1\\MULTILOG\\mlg"
#######################################################################
# generating item responses based on the 2PL #
#######################################################################
dir.create(file.path(mF)) # create main folder
for (replications in 1:nReplications) {
  # generating responses based on 2PL
  eta <- matrix(0,nSubjects,nItems)
  for (i in 1:nSubjects) {
     theta <- rnorm(1,0,1)
     for (j in 1:nItems) {
        eta[i,j] <- a[j]*(theta-b[j])
     }
  }
  randomMat <- matrix(runif(nSubjects*nItems,0,1),nSubjects,nItems)
  y <- ifelse(randomMat < (exp(eta)/(1+exp(eta))),1,0)
  colnames(y) <- c(paste("I",1:nItems,sep=""))
  # export responses y to text file with MULTILOG data format.
  # write.fwf() is used to create fixed width format data file for MULTILOG.
  # fN is the file name for data file.
  fN <-
paste("MULTILOGI",nItems,"S",nSubjects,"R",replications,".dat",sep="")
  write.fwf(data.frame(formatC(1:nSubjects,width=5,format="d",flag="0"),y)
            ,sep="",file=file.path(mF,fN),colnames=FALSE)
}
```

Appendix B

Preparing Input Files for MULTILOG

Contents of Template Input File, ”C:/template/template.mlg”

```
MULTILOG command file
for the Rasch model
>PROBLEM RANDOM,
INDIVIDUAL,
DATA = ’<datafile>’,
NITEMS = <nItems>,
NGROUPS=1,
NEXAMINEES = <nSubjects>,
NCHARS = 5;
>TEST ALL,
L2;
>END ;
2
01
<key>
N
<format>
```

Generating Input Files

```
#########################################################################
# Generating Input Files for MULTILOG using "template.mlg" #
#########################################################################
setwd(mF) # change working folder to main folder(mF)
dataFiles <- list.files(pattern = ".dat", file.path(mF)) # read data files
for (rep in 1:nReplications) {
  # template.mlg will be read and <datafile>, <nItems>, <nSubjects>
  # ,<key>, <format> in the template file will be replaced
  # with appropriate strings.
  # read template.mlg file
  text <- readLines("C:/template/template.mlg")
  # replace data file names in the template.mlg using gsub() function in R.
  dName <- file.path(mF,dataFiles[rep])
  text <- gsub(pattern = "<datafile>", replacement = dName, x = text)
  # replace number of items
  text <- gsub(pattern = "<nItems>", replacement = nItems, x = text)
  # replace number of subjects
  text <- gsub(pattern = "<nSubjects>", replacement = nSubjects, x = text)
  # replace key
  key <- paste(rep(1, nItems), collapse="")
  text <- gsub(pattern = "<key>", replacement = key, x = text)
  # replace format
  format <- paste("(5A1,", nItems,"A1)", sep="")
  text <- gsub(pattern = "<format>", replacement = format, x = text)
  # input file name is the same as the data file name
  inputFile <- paste(unlist(strsplit(dataFiles[rep], "\\."))[1], ".mlg", sep="")
  # write to text file
  writeLines(text, con = inputFile)
}
```

Appendix C

Running MULTILOG in R

```
1   ######################################################################
2   # Run MULTILOG using system() function in R #
3   ######################################################################
4   for (rep in 1:nReplications) {
5     # system(command) invokes the OS command specified by command.
6    # ex) the following command will run MULTILOG with input file 'I5S100R1.mlg':
7     # R> system("C:\\Progra~1\\MULTILOG\\mlg I5S500R9")
8      # strsplit() function is used to remove a file extension from data file
name.
9     system(paste(multilogPath,unlist(strsplit(dataFiles[rep],"\\."))[1]))
10  }
```

Appendix D

Extracting and Summarizing Results from Output files

```
1   ########################################################################
2   # Summarizing Results from MULTILOG Output Files #
3   ########################################################################
4   # read *.OUT files (MULTILOG output files) into outPut1 using lapply().
5   outPut1 <- lapply(list.files(pattern=".OUT"),readLines)
6
7   # extracting discrimination parameters from output files.
8   # extract the lines that containing discrimination parameters using grep().
9   # grep("GRADED CATEGORIES",x) returns indices for the lines that contains
10  # "GRADED CATEGORIES". Since discrimination parameters are located
11  # in the next second line after that, grep("GRADED CATEGORIES",x)+2
12  # will specify indices for the lines that contain discrimination parameters.
13  outPut2 <- lapply(outPut1, function(x) x[grep("GRADED CATEGORIES", x)+2])
14  # sort outPut2 according to the items
15  outPut3 <- lapply(1:nItems, function(x) unlist(lapply(outPut2,function(y) y[x])))
16  # extract only the decimal number from outPut3
17  # regular expression: [-+]?[0-9]+\\.[0-9]+ = any single decimal number
18  outPut4 <- sapply(outPut3, function(x) as.numeric(str_extract(x,"[-+]?[0-9]+\\.[0-
9]+")))
19  # put column name to outPut4
20  colnames(outPut4) <- paste("Item",1:nItems,"a",sep="")
21
22  # extracting difficulty parameters from output files.
23  # extract the lines that containing difficulty parameters using grep().
24  # Since difficulty parameters are located in the next third line after that,
25  # grep("GRADED CATEGORIES",x)+3 will specify indices for the lines
26  # that contain difficulty parameters.
27  outPut5 <- lapply(outPut1, function(x) x[grep("GRADED CATEGORIES",x)+3])
28  # sort according to item.
29  outPut6 <- lapply(1:nItems, function(x) unlist(lapply(outPut5,function(y) y[x])))
30  # extract decimal numbers.
31  outPut7 <- sapply(outPut6, function(x) as.numeric(str_extract(x,"[-+]?[0-9]+\\.[0-
9]+")))
32  colnames(outPut7) <- paste("Item",1:nItems,"b",sep="")
33
34  # calculate bias and RMSE across replications
35  mean_a <- apply(outPut4,2,mean)
36  bias_a <- mean_a - a
37  rmse_a <- sqrt(apply((outPut4-t(replicate(nReplications, a)))^2,2,mean))
38  mean_b <- apply(outPut7,2,mean)
39  bias_b <- mean_b - b
40  rmse_b <- sqrt(apply((outPut7-t(replicate(nReplications, b)))^2,2,mean))
41  results_a <- rbind(outPut4, mean_a, a, bias_a, rmse_a)
42  results_b <- rbind(outPut7, mean_b, b, bias_b, rmse_b)
43  finaloutput <- cbind(results_a, results_b)
44   rownames(finaloutput) <- c(1:nReplications, "mean", "true parameters", "bias",
"rmse")
45
46  # check whether the MULTILOG was terminated normally for each replication.
47  outPut8 <- lapply(outPut1, function(x) x[grep("NORMAL PROGRAM TERMINATION",x)])
48  outPut9 <- matrix(0,nReplications,1)
49  outPut9 <- lapply(1:nReplications, function(x) ifelse(length(outPut8[[x]])==0,
50              outPut9[x]<-NA, outPut9[x]<-outPut8[[x]] ) )
51  outPut9 <- do.call("rbind",outPut9)
52  outPut9 <- rbind(outPut9,0,0,0,0)
53  # export final result to text file, "finaloutput.csv"
54  finaloutput <- cbind(finaloutput,outPut9)
55  write.csv(finaloutput,file=file.path(mF, "finaloutput.csv"))
```