Conducting Monte Carlo Simulations in R

Okan Bulut

University of Alberta, bulut@ualberta.ca

Contents

Chapter 1

Introduction

1.1 Overview

Both researchers and practitioners often use Monte Carlo simulations to answer a variety of research questions. Over the past decade, `R` (R Core Team, 2019) has been one of the most popular programming languages for conducting Monte Carlo simulation studies. `R` (https://www.r-project.org/) is a free, open-source programming language for statistical computing and data visualization. Both built-in functions and many user-created packages in `R` allow researchers and

practitioners to design and implement a very simple to very comprehensive simulation studies.

This short book will explain the **major** steps in conducting Monte Carlo simulations using `R`. Here is the outline of the book[1]:

Part	Description
1	**Introduction**
	Why Simulations?
	Typical Simulation Scenarios
	Additional Resources
2	**Designing Simulations**
	Simulation Factors
	Evaluation Criteria
	Other Design Elements
3	**Running Simulation**
	Custom Functions
	Debugging the Code
	Putting the Functions Together
	Benchmarking
4	**Summarizing Simulation Results**
	Tables and Figures
	Exporting the Results

1.2 Why Simulations?

There are many reasons to conduct Monte Carlo simulations. Researchers and practitioners often choose to simulate data instead of collecting empirical data because:

- it is **impractical** and **costly** to collect empirical data while manipulating several conditions
- it is not possible to investigate the **real impact** of the study conditions without knowing the characteristics of the target population as well as the variables of interest.
- it is more difficult to deal with empirical data because it typically includes **missingness** – which may be in large amounts and nonrandom.

[1] This book was created using the `bookdown` (Xie, 2020a) and `knitr` (Xie, 2020b) packages.

1.3 Typical Simulation Scenarios

We can use Monte Carlo simulations to answer various research questions. Typical research questions in which Monte Carlo simulations can be useful are:

- Does a particular type of estimation (e.g., maximum likelihood) yield accurate results?
  - What is the level of bias?
  - What is the standard error of estimates?
  - What conditions would affect the accuracy of the estimation?
  - Does the estimation remain robust when assumptions are violated?
- Which estimation method (e.g., maximum likelihood, EAP, and MAP) is more accurate?
  - Do the performances of these methods vary by different conditions?
  - Which estimator, method, or model is the most robust?
- Can a statistical method or model (e.g., logistic regression) successfully detect a value of interest (e.g., differential item functioning)?
  - How accurate is the method when the null hypothesis is *false*?
  - How accurate is the method when the null hypothesis is *true*?

1.4 Additional Resources

If you are interested in learning more about Monte Carlo simulations, there are many online resources available. Some of these resources include:

- Bulut and Sunbul (2017)'s article: Monte Carlo Simulation Studies in Item Response Theory with the R Programming Language
- Hallgren (2013)'s article: Conducting Simulation Studies in the R Programming Environment
- Roger Peng's online book: R Programming for Data Science. In the book, Chapter 20 specifically focuses on simulations in R.
- The `SimDesign` (Chalmers, 2020) package in `R`. For examples of `SimDesign`, you can check out its Wiki page: https://github.com/philchalmers/SimDesign/wiki
- There is also another `R` package called `MonteCarlo` (Leschinski, 2019) for more general simulation studies.

Chapter 2

Designing Simulations

To better explain the steps of conducting a Monte Carlo simulation study, let's assume a hypothetical research scenario in which a researcher wants to examine item parameter estimation in item response theory (IRT). Here the researcher aims to determine the robustness of parameter estimation for the 3PL model, especially when the sample size is small.

2.1 Simulation Factors

The researcher can investigate the impact of several factors (i.e., conditions) within the same study but the goal is to create a feasible study with the factors that are essential for this study. Therefore, the following factors are selected:

- Test length (10, 15, 20, or 25 items)
- Sample size (250, 500, 750, or 1000 examinees)
- Whether or not the guessing (c) parameter should be fixed for all items (e.g., $c = 0.16$)

Let's see the total number of conditions from these factors:

4 (test length) x 4 (sample size) x 2 (fixed guessing or free estimation) = 32 conditions

This calculation assumes that these factors are **fully crossed**. Two factors are fully crossed when each level of one factor occurs in combination with each level of the other factor. However, some factors may not be crossed with the other factors. These are called **nested** factors. Two factors are nested when each level of a factor occurs in combination with different levels of another factor (see Schielzeth and Nakagawa's paper for more information about crossed and nested factors).

2.2 Evaluation Criteria

The researcher is interested in **accuracy**. Therefore, we have to use several measures of accuracy to evaluate the simulation results. Some of these measures (i.e., indices) include:

- **Bias:** $bias = \frac{1}{R}\Sigma_{r=1}^{R}(\hat{x_r} - x)$, where R is the number of replications, $\hat{x_r}$ is the estimated value of the parameter, and x is the true value of the parameter. Bias, which can be either positive or negative, should be close to zero for higher accuracy.
- **Root-mean square error (RMSE):** $RMSE = \sqrt{\frac{1}{R}\Sigma_{r=1}^{R}(\hat{x_r} - x)^2}$, where R is the number of replications, $\hat{x_r}$ is the estimated value of the parameter, and x is the true value of the parameter. RMSE, which is either zero or a positive value, should be close to zero for higher accuracy.
- **(Pearson) Correlation:** $\rho(x, \hat{x}) = \frac{\text{cov}(x, \hat{x})}{\sigma_x \sigma_{\hat{x}}}$, where $\hat{x_r}$ is the estimated value of the parameter, x is the true value of the parameter, the numerator is the covariance of the two parameters, and the denominator is the product of their standard deviations. Correlation should be closer to 1 as the accuracy increases.

Note that there are other types of evaluation criteria, such as power, type I error, relative efficiency, precision, and recall. In this study, the accuracy measures

listed above should be adequate for evaluating the accuracy of estimated item paramaters. We will discuss how the simulation results with these evaluation criteria can be summarized and presented in Part 4 of this book.

2.3 Other Design Elements

Item and ability parameters: The researcher must use either a fixed set of item parameters from an existing instrument or simulated item parameters similar to those from an existing instrument. Thus, the researcher uses the distributions provided in Casabianca and Lewis (2015)'s article based on a 2008 National Mathematics Assessment:

- $a \sim N(1.13, 0.25)$
- $b \sim N(0.21, 0.51)$
- $c \sim N(0.16, 0.05)$

For the ability distribution, ability values are drawn from a normal distribution, $\theta \sim N(0, 1)$.

Number of replications: The number of replications should be adequate to create enough variation in the simulation. The higher the number of replications, the longer the simulation study will take. Therefore, it is important to choose a suitable number. Typically, 100 replications are enough for this type of simulation study. We can test this by increasing the number of iterations to 150 and check whether the overall results would change substantially.

Replication mechanism: In a typical Monte Carlo simulation study, the levels of simulation factors remain fixed, while the data, parameters, and other parts vary. In other words, these are the parts that we actually simulate. For the sake of simplicity, some of these parts can be generated once while the other parts continuously change from one replication to another. In this study, the researcher does not have a specific set of item parameters to test. Instead, the goal of the study is more general: investigating the impact of the identified simulation factors on item parameter estimation. Therefore, unique sets of item parameters, ability values, and response data can be generated with each replication.

Chapter 3

Running Simulations

The secret of building a successful Monte Carlo simulation is to write several custom functions that work efficiently and more importantly – **correctly**. Then, we can:

- combine the custom functions within a single function and run the simulations using this function with nested loops, or
- write several nested loops that run each custom function independently and carry its output to the next function.

As we put the simulation functions together, we should be aware of potential errors that could either produce incorrect results or interrupt the execution of the simulation at different stages. Therefore, it is important to check all the

functions carefully and patiently before running the simulations. Adding either notifications or diagnostic messages into the functions is also quite helpful for debugging problems that might occur in the functions.

Once all the simulation functions are checked, the next step is determine how to run the simulations as efficiently as possible. Loops are particularly slow in `R` when it comes to handling heavy computations and data operations. The `apply()` function collection including `apply()`, `sapply()`, and `lapply()` can help users avoid the loops and perform many computations and data operations very efficiently in `R`. For users who want to run their simulations with loops, parallel computing in `R` is also an excellent way to make the simulations faster. Regardless of which of these methods has been selected, it is useful to check running time of `R` codes – which is also known as **benchmarking**. We will discuss all of these steps in the remainder of Part 3.

3.1 Custom Functions

Writing custom functions does **not** mean that we must write every single algorithm from scratch. Instead, it means that we bring several functions (either new or existing) together so that they run the analysis that we want and return the results in the way we prefer. Going back to the hypothetical simulation scenario that we introducted in Part 1, the researcher needs several custom functions for:

1. Generating response data
2. Estimating item parameters
3. Calculating bias, RMSE, and correlation

In the first function, we want the function to generate item parameters based on the parameter distributions that we have mentioned in Part 2 and to use these parameters to generate dichotomous response data following the 3PL model. We call this function `generate_data`. Our function has several arguments based on the simulation factors: `nitem` for the number of items, `nexaminee` for the number of examinees, and `seed` that allows us to control the random number generation. Setting the seed is very important in Monte Carlo simulations because it enables the reproducibility of simulation results. When we run the same function using the same seeds in the future, we should be able to generate the same parameters and response data.

To generate responses, we will use the `sim` function from the `irtoys` package (Partchev and Maris, 2017). We could make `irtoys` a required package for our custom function, using `require("irtoys")`. This would activate the package every time we use `generate_data`. However, some packages can mask functions from other packages with the same names, when they are activated together within the same `R` session. Therefore, it is better to call the exact function that we want to use, instead of activating the entire package. Here we use

`irtoys::sim` to call the `sim` function from `irtoys`.

Note that the three arguments, `nitem`, `nexaminee`, and `seed`, in the `generate_data` function are **required**. That is, all of these arguments must be specified as we run the simulations. This might create some problems in the simulation. We will talk about this in the debugging section later on.

```
generate_data <- function(nitem, nexaminee, seed) {
  # Set the seed
  set.seed(seed)

  # Generate item parameters
  itempar <- cbind(
    rnorm(nitem, mean = 1.13, sd = 0.25)*1.702, #a
    rnorm(nitem, mean = 0.21, sd = 0.51)*1.702, #b
    rnorm(nitem, mean = 0.16, sd = 0.05))       #c

  # Generate ability parameters
  ability <- rnorm(nexaminee, mean = 0, sd = 1)

  # Generate response data according to the 3PL model
  respdata <- irtoys::sim(ip = itempar, x = ability)
  colnames(respdata) <- paste0("item", 1:nitem)

  # Combine the generated values in a list
  data <- list(itempar = itempar,
               ability = ability,
               seed = seed,
               respdata = respdata)

  # Return the simulated data
  return(data)
}
```

Our function saves the item parameters, ability parameters, simulated responses, and the seed for the `set.seed` argument. Let's see if our function returns what we asked for.

```
temp1a <- generate_data(nitem = 10, nexaminee = 1000, seed = 666)
head(temp1a$itempar)
head(temp1a$ability)
head(temp1a$respdata)
```

```
      [,1]    [,2]    [,3]
[1,] 2.244  2.2237 0.12540
[2,] 2.780 -1.1792 0.10085
[3,] 1.772  1.1080 0.22344
[4,] 2.786 -1.1357 0.14437
```

```
[5,] 0.980  0.4738 0.16153
[6,] 2.246  0.2916 0.08589
```

```
[1]  0.7550 -0.6415  1.4311 -0.6246  0.2290  0.2632
```

```
     item1 item2 item3 item4 item5 item6 item7 item8 item9 item10
[1,]     0     1     1     1     1     1     1     0     1      1
[2,]     0     1     1     1     1     0     0     0     0      0
[3,]     0     1     1     1     1     1     1     0     1      1
[4,]     0     1     0     0     0     0     0     0     1      0
[5,]     0     1     1     1     1     1     1     0     1      0
[6,]     0     1     1     1     0     1     0     1     0      1
```

Our second function is the `mirt` function from the `mirt` package (Chalmers, 2019). We will create a custom function to be able to add a fixed guessing parameter and to extract the estimated item parameters. Here we will set `guess` as a negative value (e.g., -1), indicating that we want guessing to be freely estimated. However, if `guess` is greater than or equal to zero, then the guessing parameter will be fixed to this value in the function.

```
estimate_par <- function(data, guess = -1) {
  # If guessing is fixed
  if(guess >= 0) {

    # Model set up
    mod3PL <- mirt::mirt(data, # response data
                         1,    # unidimensional model
                         guess = guess, # fixed guessing
                         verbose = FALSE, # Don't print verbose
                         # Increase the number of EM cycles
                         # Turn off estimation messages
                         technical = list(NCYCLES = 1000,
                                          message = FALSE))
  } else {
    mod3PL <- mirt::mirt(data, # response data
                         1,    # unidimensional model
                         itemtype = "3PL", # IRT model
                         verbose = FALSE, # Don't print verbose
                         # Increase the number of EM cycles
                         # Turn off estimation messages
                         technical = list(NCYCLES = 1000,
                                          message = FALSE))
  }

  # Extract item parameters in typical IRT metric
  itempar_est <- as.data.frame(mirt::coef(mod3PL, IRTpars = TRUE, simplify = TRUE)$ite
  return(itempar_est)
}
```

Let's see if our `estimate_par` function can return what we want.

```
temp1b <-estimate_par(data = temp1a$respdata, guess = -1)
head(temp1b)
```

```
           a       b        g
item1 4.257  2.1529 0.123380
item2 3.192 -1.2152 0.002664
item3 1.738  1.1514 0.194192
item4 2.628 -1.1979 0.006361
item5 1.312  0.7376 0.315269
item6 2.103  0.3627 0.080536
```

The last function necessary for this simulation study is a summary function that will provide bias, RMSE, and correlation values for the estimated parameters. We can come up with so many solutions for this step of the simulation. We can determine the best solution based on the answers to the following questions:

1. Do we want to save all the simulation results and summarize them all together once all the replications are completed? *This is a very safe option although saving all the results might take a lot of space or memory in the computer*
2. Do we want to compute each evaluation index one by one and combine them in a data frame at the end? *This is a good option although this would require implementing separate functions for bias, RMSE, and correlation*
3. Can we write a single function that computes all three indices and returns them in a single data frame? *This sounds more feasible because all the indices are computed together and then stored for each replication, though it might be harder to debug if a problem occurs*

We wil follow the third option and create a single summary function. Note that this will return the average bias and RMSE across **all** items in a **single** replication. Therefore, we would still have to summarize the results across all replications once the simulations are complete. We create `summarize` as a new function which takes two arguments: `est_params` as the estimated parameters and `true_params` as the true parameters. The function will calculate bias, RMSE, and correlations among the estimated and true parameters. Then, it will return them in a long-format data set (i.e., one row per parameter and the columns indicating our evaluation indices). Not to repeat the same computations for each column (i.e., parameter), we use `sapply` to apply the functions so that bias, RMSE, and correlation can be calculated for the a, b, and c parameters together.

```
summarize <- function(est_params, true_params) {
  result <- data.frame(
    parameter = c("a", "b", "c"),
    bias = sapply(1L:3L, function(i) mean((est_params[, i] - true_params[,i]))),
    rmse = sapply(1L:3L, function(i) sqrt(mean((est_params[, i] - true_params[,i])^2))),
```

```
    correlation = sapply(1L:3L, function(i) cor(est_params[, i], true_params[,i])))
  return(result)
}
```

Finally, let's test whether our final function, `summarize`, works properly.

```
temp1c <- summarize(temp1b, temp1a$itempar)
temp1c
```

```
  parameter       bias    rmse correlation
1         a  0.2462034 0.67355      0.7435
2         b  0.0604238 0.11273      0.9957
3         c -0.0004633 0.07805      0.4464
```

3.2 Debugging the Code

Debugging R codes is a rather tedious task, though this step is essential for the success of Monte Carlo simulation studies. It is important to test each custom function in a Monte Carlo simulation study as much as possible before we run them together within a single function or a loop. Otherwise, it might be harder to locate where the error occurs once the simulation begins. Especially with complex simulations, some errors might appear many hours (or even days) after we start running the simulations. Therefore, it is better to test each function in the simulation independently and then all together by following the sequence that all the functions would normally follow in the complete simulation.

In the following example, we will test one of the custom functions (`generate_data`) that we have created earlier. The function requires `nitem`, `nexaminee`, and `seed` as its arguments. What if we forget to include a seed when we are running this particular function? What would the function return? Using this example, we will review some of the debugging methods in `R`.

3.2.1 Custom Messages

There are many ways to find where the error occurs in R codes. An old-fashioned and yet effective way for debugging R codes is to place custom messages (similar to annotations or comments in the code) throughout the functions. If the simulation stops at a certain point in the function, we can see the latest message printed and try to find the location of the problem accordingly.

Now let's run the `generate_data` function without a seed and see what happens.

```
temp2 <- generate_data(nitem = 10, nexaminee = 1000)
```

```
Error in set.seed(seed): argument "seed" is missing, with no default
```

The output shows that the error is happening because we did not specify the seed. To fix the problem, we will revise our function by adding a condition: if seed is missing (i.e., NULL), then the function will randomly select an integer between 1 and 10,000 and use this random integer as the seed. In addition, we will add a few custom messages in the code, using `cat` and `print`. These will print messages in different forms as `R` runs each line of the code.

```
generate_data <- function(nitem, nexaminee, seed = NULL) {

  if(!is.null(seed)) {
    set.seed(seed)}
  else {
    seed <- sample.int(10000, 1)
    set.seed(seed)
    cat("Random seed = ", seed, "\n")
  }

  print("Generated item parameters")
  itempar <- cbind(
    rnorm(nitem, mean = 1.13, sd = 0.25), #a
    rnorm(nitem, mean = 0.21, sd = 0.51), #b
    rnorm(nitem, mean = 0.16, sd = 0.05)) #c

  print("Generated ability parameters")
  ability <- rnorm(nexaminee, mean = 0, sd = 1)

  print("Generated response data")
  respdata <- irtoys::sim(ip = itempar, x = ability)
  colnames(respdata) <- paste0("item", 1:nitem)

  print("Combined everything in a list")
  data <- list(itempar = itempar,
               ability = ability,
               seed = seed,
               respdata = respdata)

  # Return the simulated data
  return(data)
}
```

Now let's see what our updated function would return if we forget to provide a seed.

```
temp2 <- generate_data(nitem = 10, nexaminee = 1000)
```

```
Random seed =  7680
[1] "Generated item parameters"
```

```
[1] "Generated ability parameters"
[1] "Generated response data"
[1] "Combined everything in a list"
```

3.2.2 Messages and Warnings

There are other ways to add debugging-related messages into functions, such as:

- `message`: A custom, diagnostic message created by the `message()` function. The message gets printed only if a particular condition is or is not met. This does **not** stop the execution of the function.
- `warning`: A custom, warning message via `warning()`. The message indicates that something is wrong with the function but the execution of the function continues.

Using the previous example, we will modify our function by adding a warning message showing there is no seed specified in the function and a message indicating the random seed assigned by the function itself (instead of a user-defined seed).

```
generate_data <- function(nitem, nexaminee, seed = NULL) {

  if(!is.null(seed)) {
    set.seed(seed)}
  else {
    warning("No seed provided!", call. = FALSE)
    seed <- sample.int(10000, 1)
    set.seed(seed)
    message("Random seed = ", seed, "\n")
  }

  itempar <- cbind(
    rnorm(nitem, mean = 1.13, sd = 0.25), #a
    rnorm(nitem, mean = 0.21, sd = 0.51), #b
    rnorm(nitem, mean = 0.16, sd = 0.05)) #c

  ability <- rnorm(nexaminee, mean = 0, sd = 1)

  respdata <- irtoys::sim(ip = itempar, x = ability)
  colnames(respdata) <- paste0("item", 1:nitem)

  data <- list(itempar = itempar,
               ability = ability,
               seed = seed,
               respdata = respdata)
```

```
  return(data)
}
```

What if we want to catch all errors and warnings as the function runs and store these messages to be review at the end? The answer comes from a solution offered in the R-help mailing list (see `demo(error.catching)`):

```
tryCatch.W.E <- function(expr) {
  W <- NULL
  w.handler <- function(w){ # warning handler
    W <<- w
    invokeRestart("muffleWarning")
    }
  list(value = withCallingHandlers(tryCatch(expr, error = function(e) e),
                                   warning = w.handler), warning = W)
}

temp3 <- tryCatch.W.E(generate_data(nitem = 10, nexaminee = 1000))
```

`temp3` stores both the results of `generate_data` and the warning message.

```
str(temp3)
```

```
List of 2
 $ value  :List of 4
  ..$ itempar : num [1:10, 1:3] 1.36 1.02 1.2 1.3 1.26 ...
  ..$ ability : num [1:1000] -0.7446 0.0013 -0.8943 -0.6846 -0.8861 ...
  ..$ seed    : int 6984
  ..$ respdata: num [1:1000, 1:10] 0 1 0 0 1 1 1 0 1 1 ...
  .. ..- attr(*, "dimnames")=List of 2
  .. .. ..$ : NULL
  .. .. ..$ : chr [1:10] "item1" "item2" "item3" "item4" ...
 $ warning:List of 2
  ..$ message: chr "No seed provided!"
  ..$ call   : NULL
  ..- attr(*, "class")= chr [1:3] "simpleWarning" "warning" "condition"
```

Let's see if our function returned any warnings:

```
temp3$warning
```

```
<simpleWarning: No seed provided!>
```

3.2.3 Interactive Debugging with `browser()`

A relatively more sophisticated way to diagnose a problem in R codes is to use `browser()`. This stops the execution of our codes and opens an interactive debugging screen. Using this debugging screen, we can do several things such

as viewing what objects we have in the current `R` environment, modifying these objects, and them continuing running the code after these changes.

Now assume that users of our `generate_data` function *must* use their own seed, instead of having the function to generate one automatically. Therefore, we want to the function to stop if that occurs. We will use the `stop` function for this process. This function can be placed within an `if` statement to prevent the function for continuing – *if a particular condition (or multiple conditions) occurs*. Another version of this function is `stopifnot` which stops the function if a particular condition does **not** occur. To see how `stopifnot` works, you can type `?stopifnot` in the `R` console and check out its help page. In the following example, we will use `stop`.

```
generate_data <- function(nitem, nexaminee, seed = NULL) {

  if(is.null(seed)) {
    stop("A seed must be provided!")}
  else {
    set.seed(seed)
  }

  print("Generated item parameters")
  itempar <- cbind(
    rnorm(nitem, mean = 1.13, sd = 0.25), #a
    rnorm(nitem, mean = 0.21, sd = 0.51), #b
    rnorm(nitem, mean = 0.16, sd = 0.05)) #c

  print("Generated ability parameters")
  ability <- rnorm(nexaminee, mean = 0, sd = 1)

  print("Generated response data")
  respdata <- irtoys::sim(ip = itempar, x = ability)
  colnames(respdata) <- paste0("item", 1:nitem)

  print("Combined everything in a list")
  data <- list(itempar = itempar,
               ability = ability,
               seed = seed,
               respdata = respdata)

  # Return the simulated data
  return(data)
}

temp3 <- generate_data(nitem = 10, nexaminee = 1000)
```

Running the above code above would activate the debugging mode in RStudio

and the console would like this:

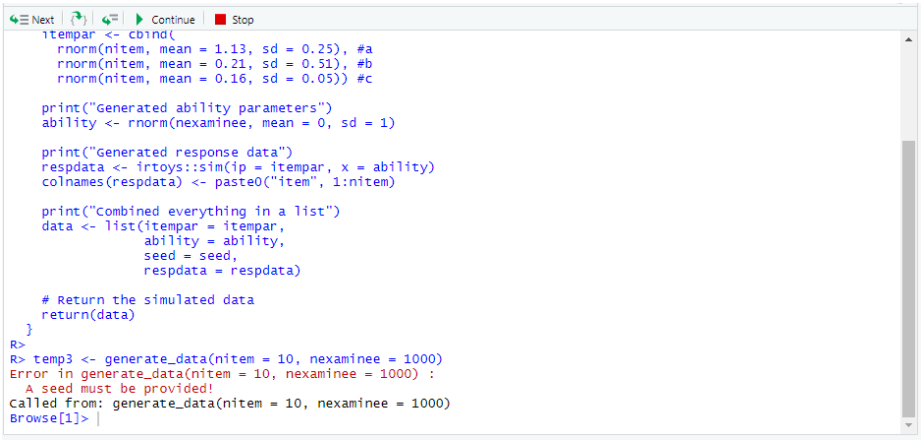


Figure 3.1: Console once the interactive debugging begins

RStudio enables the browsing mode automatically in most cases – this is why we have seen `Browse[1]>` after the error occured in the code. Using the browser buttons, we can evaluate the next statement, evaluate the next statement but step into the current function, execute the remainder of the function, continue executing the code until the next error or breaking point, or exit the debug mode, based on the buttons from left to right (see the figure below).

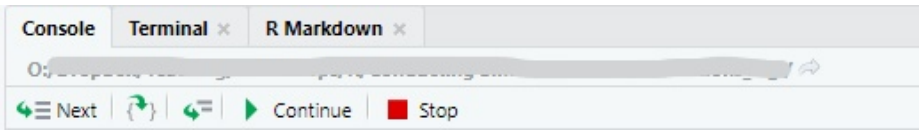


Figure 3.2: Debugging menu options

In addition to the console, the source (where we are writing the codes) would show:

The figure above shows that the browser was able to find the location of the error. We can also activate the debugging mode for a particular function using `debug()`. For example, using `debug(generate_data)` and running `temp3 <- generate_data(nitem = 10, nexaminee = 1000)` will show the following output in the console. By clicking on the "next" button, we can debug the code slowly and see where the error occurred. Once we are satisfied with the function, we can turn off the interactive debugging mode using the `undebug()` function.

If we want to see how we came to this particular error in the function, we can hit "enter" to exit the browser mode and type `traceback()`. This would return

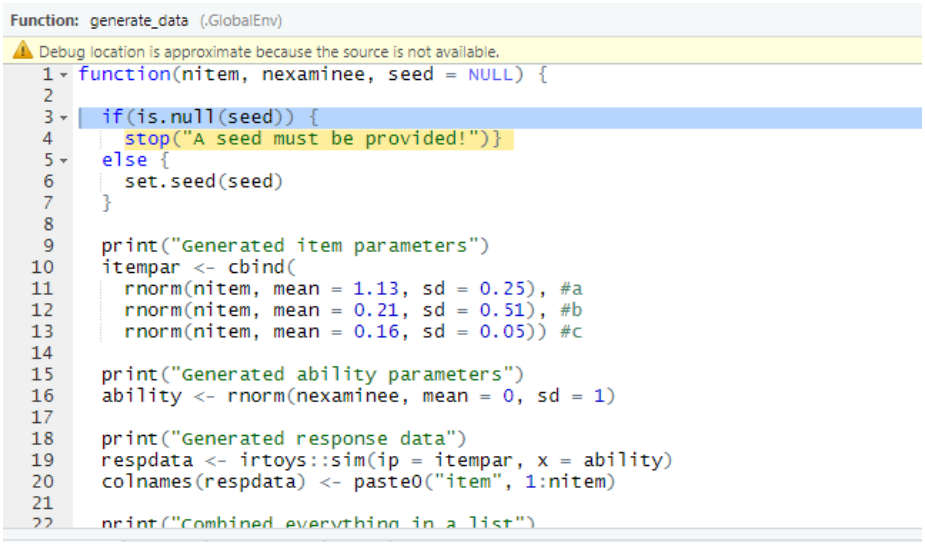


Figure 3.3: Error location in the interactive debugging

the steps that we have taken until the error message:

For more further information about debugging in `R`, I highly recommend you to check out:

- the Debugging chapter in Hadley Wickham's Advanced R, and
- the Debugging R Code chapter in Jennifer Bryan and Jim Hester's What They Forgot to Teach You About R

3.3 Putting the Functions Together

As we put all the custom functions together, we must determine:

1. whether we want to run each custom function separately or in a single wrapper function that simply calls all of our custom functions
2. how we want to replicate our custom functions:

- the `replicate` function
- the `apply` function collection
- loops (with single cluster or parallel computing)

```
R> debug(generate_data)
R> temp3 <- generate_data(nitem = 10, nexaminee = 1000)
debugging in: generate_data(nitem = 10, nexaminee = 1000)
debug at #1: {
    if (is.null(seed)) {
        stop("A seed must be provided!")
    }
    else {
        set.seed(seed)
    }
    print("Generated item parameters")
    itempar <- cbind(rnorm(nitem, mean = 1.13, sd = 0.25), rnorm(nitem,
        mean = 0.21, sd = 0.51), rnorm(nitem, mean = 0.16, sd = 0.05))
    print("Generated ability parameters")
    ability <- rnorm(nexaminee, mean = 0, sd = 1)
    print("Generated response data")
    respdata <- irtoys::sim(ip = itempar, x = ability)
    colnames(respdata) <- paste0("item", 1:nitem)
    print("Combined everything in a list")
    data <- list(itempar = itempar, ability = ability, seed = seed,
        respdata = respdata)
    return(data)
}
Browse[2]> n
debug at #3: if (is.null(seed)) {
    stop("A seed must be provided!")
} else {
    set.seed(seed)
}
Browse[2]> n
debug at #4: stop("A seed must be provided!")
Browse[2]> n
Error in generate_data(nitem = 10, nexaminee = 1000) :
  A seed must be provided!
Browse[3]> n
R>
```

Figure 3.4: Using the debug() function

```
R> temp3 <- generate_data(nitem = 10, nexaminee = 1000)
Error in generate_data(nitem = 10, nexaminee = 1000) :
  A seed must be provided!
Called from: generate_data(nitem = 10, nexaminee = 1000)
Browse[1]>
R> traceback()
2: stop("A seed must be provided!") at #4
1: generate_data(nitem = 10, nexaminee = 1000)
```

Figure 3.5: The traceback() option in R

3.3.1 Avoding Loops

One of the easiest ways to repeat the same function without a loop is to use the `replicate` function in `R`. The way this function works is very simple. Let's take a look at its basic structure:

```
replicate(<number of replications>, <function to be replicated>)
```

Using the `repeat` function, we can run the `generate_data` function many times very quickly. For example, let's create 5 data sets (i.e., 5 replications):

```
nreps = 5L
x <- replicate(nreps, generate_data(nitem = 10, nexaminee = 1000))
head(x)
```

```
         [,1]          [,2]          [,3]          [,4]          [,5]
itempar  Numeric,30    Numeric,30    Numeric,30    Numeric,30    Numeric,30
ability  Numeric,1000  Numeric,1000  Numeric,1000  Numeric,1000  Numeric,1000
seed     8022          3891          7049          811           6957
respdata Numeric,10000 Numeric,10000 Numeric,10000 Numeric,10000 Numeric,10000
```

This would return an array (i.e., vector) called `x` which would store our simulated data sets returned from `generate_data`. To call a particular data set from this array (e.g., data set 1), we would use:

```
x[,1]
```

which would return all the components (i.e., item parameters, ability parameters, seed, and response data) from data set 1.

Another effective method to repeat a function several times without a loop is to use the `apply` function collection: `apply()`, `sapply()`, and `lapply()`. The purpose of these functions is primarily to avoid explicit uses of loops when repeating a particular task in `R`[1]. Among these three functions, `sapply()` and `lapply()` are particularly useful in simulations because they can apply an existing function (e..g, mean, median, sd) or a custom function to each element of an object. The difference between `lapply()` and `sapply()` lies between the output they return. `lapply()` applies a function and returns a list as its output, whereas `sapply()` returns a vector as its output.

In the following example, we want to extract the true item parameters from each replication stored in `x`. This is the first element of the output returned from `generate_data`. Therefore, we will use both `lapply` and `sapply` to select the first element of 5 replications stored in `x`.

```
# lapply - returning a list
lapply(1L:nreps, function(i) x[,i][1])
```

```
[[1]]
```

[1] See the Guru99 website for more information on these functions.

```
[[1]]$itempar
        [,1]    [,2]    [,3]
 [1,] 1.1415 -0.3682 0.07736
 [2,] 1.3827  0.3420 0.24551
 [3,] 1.0107  0.4589 0.14046
 [4,] 0.9634  0.1928 0.10050
 [5,] 1.0133  0.7065 0.23558
 [6,] 0.9482  0.1607 0.16483
 [7,] 0.8283 -0.3466 0.11231
 [8,] 1.3640  0.3054 0.17842
 [9,] 0.5806  0.3969 0.16591
[10,] 1.8895  0.1825 0.12764


[[2]]
[[2]]$itempar
        [,1]      [,2]    [,3]
 [1,] 0.8686  0.211830 0.23417
 [2,] 1.0666 -0.647674 0.22204
 [3,] 1.3665 -0.179088 0.15410
 [4,] 0.8196  0.206515 0.19359
 [5,] 1.2334  0.005374 0.25132
 [6,] 1.0186 -0.120199 0.27701
 [7,] 1.0917 -0.096444 0.20174
 [8,] 1.1214  0.387298 0.06626
 [9,] 1.4620 -0.353471 0.18427
[10,] 0.9542 -0.386885 0.11406


[[3]]
[[3]]$itempar
        [,1]      [,2]    [,3]
 [1,] 0.8242  0.596302 0.07725
 [2,] 1.6065 -0.290621 0.17878
 [3,] 1.2838  0.728311 0.20805
 [4,] 1.0859 -0.440566 0.16173
 [5,] 1.1749 -0.045554 0.23987
 [6,] 1.1786  0.006729 0.22537
 [7,] 0.8680  1.019832 0.15020
 [8,] 1.4489 -0.435660 0.14789
 [9,] 1.4630  1.346885 0.12883
[10,] 0.5805 -0.130311 0.26612


[[4]]
[[4]]$itempar
```

```
        [,1]     [,2]    [,3]
 [1,] 1.0833 -0.62881 0.18135
 [2,] 1.4569  0.35718 0.16981
 [3,] 1.4572  0.58300 0.15132
 [4,] 0.8332  0.77767 0.19911
 [5,] 0.7998  0.84951 0.07186
 [6,] 1.2964  1.40714 0.19965
 [7,] 1.0498  0.25638 0.23851
 [8,] 1.1080 -0.58124 0.16927
 [9,] 0.7451  0.08085 0.18025
[10,] 0.8302  0.61757 0.18997


[[5]]
[[5]]$itempar
        [,1]     [,2]    [,3]
 [1,] 1.2428  0.38038 0.15683
 [2,] 0.9222  0.06897 0.14285
 [3,] 1.2159 -0.53983 0.08549
 [4,] 1.3092  0.47354 0.17746
 [5,] 1.0709 -0.15980 0.20135
 [6,] 0.8401  0.70899 0.20493
 [7,] 1.0457 -0.57915 0.19977
 [8,] 1.0311  0.96456 0.22195
 [9,] 1.4022 -0.48486 0.16690
[10,] 0.8243 -0.12529 0.11996
```

```
# sapply - returning a vector or matrix
sapply(1L:nreps, function(i) x[,i][1])
```

```
$itempar
        [,1]    [,2]    [,3]
 [1,] 1.1415 -0.3682 0.07736
 [2,] 1.3827  0.3420 0.24551
 [3,] 1.0107  0.4589 0.14046
 [4,] 0.9634  0.1928 0.10050
 [5,] 1.0133  0.7065 0.23558
 [6,] 0.9482  0.1607 0.16483
 [7,] 0.8283 -0.3466 0.11231
 [8,] 1.3640  0.3054 0.17842
 [9,] 0.5806  0.3969 0.16591
[10,] 1.8895  0.1825 0.12764

$itempar
        [,1]      [,2]    [,3]
 [1,] 0.8686  0.211830 0.23417
```

```
 [2,] 1.0666 -0.647674 0.22204
 [3,] 1.3665 -0.179088 0.15410
 [4,] 0.8196  0.206515 0.19359
 [5,] 1.2334  0.005374 0.25132
 [6,] 1.0186 -0.120199 0.27701
 [7,] 1.0917 -0.096444 0.20174
 [8,] 1.1214  0.387298 0.06626
 [9,] 1.4620 -0.353471 0.18427
[10,] 0.9542 -0.386885 0.11406

$itempar
        [,1]      [,2]    [,3]
 [1,] 0.8242  0.596302 0.07725
 [2,] 1.6065 -0.290621 0.17878
 [3,] 1.2838  0.728311 0.20805
 [4,] 1.0859 -0.440566 0.16173
 [5,] 1.1749 -0.045554 0.23987
 [6,] 1.1786  0.006729 0.22537
 [7,] 0.8680  1.019832 0.15020
 [8,] 1.4489 -0.435660 0.14789
 [9,] 1.4630  1.346885 0.12883
[10,] 0.5805 -0.130311 0.26612

$itempar
        [,1]     [,2]    [,3]
 [1,] 1.0833 -0.62881 0.18135
 [2,] 1.4569  0.35718 0.16981
 [3,] 1.4572  0.58300 0.15132
 [4,] 0.8332  0.77767 0.19911
 [5,] 0.7998  0.84951 0.07186
 [6,] 1.2964  1.40714 0.19965
 [7,] 1.0498  0.25638 0.23851
 [8,] 1.1080 -0.58124 0.16927
 [9,] 0.7451  0.08085 0.18025
[10,] 0.8302  0.61757 0.18997

$itempar
        [,1]     [,2]    [,3]
 [1,] 1.2428  0.38038 0.15683
 [2,] 0.9222  0.06897 0.14285
 [3,] 1.2159 -0.53983 0.08549
 [4,] 1.3092  0.47354 0.17746
 [5,] 1.0709 -0.15980 0.20135
 [6,] 0.8401  0.70899 0.20493
 [7,] 1.0457 -0.57915 0.19977
 [8,] 1.0311  0.96456 0.22195
```

```
 [9,] 1.4022 -0.48486 0.16690
[10,] 0.8243 -0.12529 0.11996
```

This is a very simple example of how `lapply` and `sapply` work. Using the same structure, we could also estimate the item parameters with `mirt` and save all the results together.

```
data <- lapply(1L:nreps,
               function(i) x[,i][4]) # the 4th part is respdata

models <- lapply(lapply(data, '[[', 'respdata'), # select respdata from each list
                 function(i) mirt::mirt(data = i, 1, itemtype = "3PL")) # apply mirt t

parameters <- lapply(models,
                     function(x) mirt::coef(x, IRTpars = TRUE, simplify = TRUE)$items[
```

3.3.2 Loops

Loops in `R` are useful when:

- there is a series of functions to be executed and
- the functions return a particular object (e.g., an integer, a data frame, or a matrix) that is necessary for the subsequent functions.

There are several control statements essential for loops:

- `if` and `else` for testing a condition and acting on it
- `for` for setting up and running a loop a fixed number of iterations
- `while` for executing a loop while a condition is true
- `break` for stopping the execution of a loop
- `next` for skipping an iteration of a loop

Let's take a quick look at how each of these control statements works:

if-else statement

```
if(<condition1>) {
  ## do action #1
  } else if(<condition2>)  {
    ## do action #2
    } else {
      ## do action #3
      }
```

for statement

```
for(<index1> in <values of index1>) {
  for(<index2> in <values of index2>) {
    ## do an action for each index1 and index2
```

```
  }
}
```

while statement

```
while(<condition>) {
  ## do an action while condition == TRUE
}
```

break statement

```
for(<index> in <values of index>) {
  ## do an action for each index
  if(<condition>) {
    break # Stop the action if condition == TRUE
  }
}
```

next statement

```
for(<index> in <values of index>) {
  if(<condition>) {
    next ## Skip the action if condition == TRUE
  }
  ## do an action for each index
}
```

In general, loops are very useful in Monte Carlo simulations; however, they can be very slow when:

- we are dealing with heavy computations
- we apply a function to each row or column of a large data set

Therefore, compared with traditional `for` loops, the `apply` function collection is often more preferable because it can run a computation and return its results much more quickly.

3.3.3 Parallel Computing

In some computations, creating a loop might be necessary. The regular `for` loop statement executes all the functions using a single processor in the computer. Similarly, the `apply` function collection also utilizes a single processor in the computer. The use of a single processor is the default setting in `R`. That is, regardless of whether we are using a 64-bit version of `R` in a multi-core, powerful computer, `R` always uses only **one** processor by default.

For example, in a simulation study with 100 replications via `for` loops, a single processor would have to run each iteration one by one and return the results once all replications are completed. This may not be a problem especially if

we are running a very simple simulation. However, completing a simulation study involving heavy computations for each replication, using a single processor could be very time consuming. Fortunately, there is a solution to this problem: **parallel computing**[2].

To benefit from parallel computing when running a Monte Carlo simulation study, we will use two packages: `foreach` (Microsoft and Weston, 2020) and `doParallel` (Corporation and Weston, 2019a). `doParallel` essentially provides a mechanism needed to execute `foreach` loops in parallel computing (see the vignette for the `doParallel` package).

There are several steps in setting up parallel computing with `doParallel`:

1. Make clusters for parallel computing
2. Register clusters with `doParallel`
3. Run `foreach` loops using parallel computing.

Now, let's take a look at a short example. First, we will activate `doParallel`. Next, we will check how many processors (i.e., cores) are available in our computer. The number of processors is the maximum number that we can use when setting up parallel computing. This would use all the processors available, though it also slows down the computer significantly and prevents us from doing other tasks in the computer. If this is a concern, then we can assign fewer processors (instead of all of them) to parallel computing.

```
library("doParallel")
detectCores()
```

```
[1] 16
```

It seems that there are 16 processors available. We want to use 4 of these processors for running parallel loops[3].

```
cl <- makeCluster(4) # Register four clusters (Windows)
registerDoParallel(cl)
```

In a `foreach` loop, there are two ways that a loop can be set up: `%do%` for a regular loop and `%dopar%` for running the same code sequentially through multiple processors. In the following example, we will use `%dopar%`.

```
foreach(i=1:4) %dopar% sqrt(i)
stopCluster(cl) # Stop using clusters
```

Once the simulation is over, we will turn off the clusters.

```
stopCluster(cl) # Stop using clusters
```

In this small example, it is hard to see the impact of parallel computing. Parallel computing will show its strengths more clearly when we check the computing

[2] For a quick introduction to parallel computing in `R`, you can check out this website.

[3] If the computer has a Mac operating system, then `cl <- makeCluster(4, outfile="")` should be used.

time (i.e., running time) of our codes. We will talk about this in the next section.

To demonstrate how to put all the simulation functions together, we will combine the custom functions and run them together within a loop. First, we will run the latest (i.e., tested) versions of our custom functions:

```r
# Function #1
generate_data <- function(nitem, nexaminee, seed = NULL) {

  if(!is.null(seed)) {
    set.seed(seed)}
  else {
    warning("No seed provided!", call. = FALSE)
    seed <- sample.int(10000, 1)
    set.seed(seed)
    message("Random seed = ", seed, "\n")
  }

  itempar <- cbind(
    rnorm(nitem, mean = 1.13, sd = 0.25), #a
    rnorm(nitem, mean = 0.21, sd = 0.51), #b
    rnorm(nitem, mean = 0.16, sd = 0.05)) #c

  ability <- rnorm(nexaminee, mean = 0, sd = 1)

  respdata <- irtoys::sim(ip = itempar, x = ability)
  colnames(respdata) <- paste0("item", 1:nitem)

  data <- list(itempar = itempar,
               ability = ability,
               seed = seed,
               respdata = respdata)

  return(data)
}

# Function 2
estimate_par <- function(data, guess = -1) {
  # If guessing is fixed
  if(guess >= 0) {

    # Model set up
    mod3PL <- mirt::mirt(data, # response data
                         1,    # unidimensional model
                         guess = guess, # fixed guessing
                         verbose = FALSE, # Don't print verbose
                         # Increase the number of EM cycles
```

```
                          # Turn off estimation messages
                          technical = list(NCYCLES = 1000,
                                           message = FALSE))
  } else {
    mod3PL <- mirt::mirt(data, # response data
                          1,    # unidimensional model
                          itemtype = "3PL", # IRT model
                          verbose = FALSE, # Don't print verbose
                          # Increase the number of EM cycles
                          # Turn off estimation messages
                          technical = list(NCYCLES = 1000,
                                           message = FALSE))
  }

  # Extract item parameters in typical IRT metric
  itempar_est <- as.data.frame(mirt::coef(mod3PL, IRTpars = TRUE, simplify = TRUE)$ite
  return(itempar_est)
}

# Function #3
summarize <- function(est_params, true_params) {
  result <- data.frame(
    parameter = c("a", "b", "c"),
    bias = sapply(1L:3L, function(i) mean((est_params[, i] - true_params[,i]))),
    rmse = sapply(1L:3L, function(i) sqrt(mean((est_params[, i] - true_params[,i])^2)))
    correlation = sapply(1L:3L, function(i) cor(est_params[, i], true_params[,i])))
  return(result)
}
```

Second, we will set up our simulation conditions so that it would be easier to update them as we run all the conditions. Note that the number of iterations (i.e., replications) is only 4 for now. If the functions work as expected, we will increase the number of iterations to 100.

```
iterations = 4 # Only 4 iteration for now
seed = sample.int(10000, 100)
nitem = 10 # 10, 15, 20, or 25
nexaminee = 1000 # 250, 500, 750, or 1000
guess = -1 # A negative value or a value from 0 to 1
```

Third, we will set up parallel computing and register multiple processors for the simulation.

```
cl <- makeCluster(4) # Register four clusters
registerDoParallel(cl)
```

Finally, we will run our simulation and save the results as `simresults`.

```
simresults <- foreach(i=1:iterations,
                      .packages = c("mirt", "doParallel"),
                      .combine = rbind) %dopar% {
                        # Generate item parameters and data
                        step1 <- generate_data(nitem=nitem, nexaminee=nexaminee, seed=seed[i])
                        # Estimate item parameters
                        step2 <- estimate_par(step1$respdata, guess = guess)
                        # Summarize results
                        summarize(step2, step1$itempar)
                        }
```

Now, we can see the results (assuming no error messages appeared on our console):

```
simresults
```

```
   parameter      bias   rmse correlation
1          a  0.128163 0.3748     0.78904
2          b -0.059314 0.5941     0.62520
3          c -0.025765 0.1377     0.24511
4          a -0.030007 0.3135     0.49103
5          b -0.244383 0.4692     0.88362
6          c -0.052045 0.1444    -0.27393
7          a  0.199876 0.7391    -0.08689
8          b  0.017008 0.2874     0.82013
9          c  0.002217 0.1187     0.13883
10         a  0.310501 0.4886     0.78836
11         b  0.129384 0.3310     0.84102
12         c  0.072968 0.1203     0.42944
```

Remember that this is only for **one** of the crossed conditions (10 items, 1000 examinees, no fixed guessing) across **four** iterations. We can summarize the results across all iterations, add additional information to remind us what we have done in the simulation, and save the results. We will use the `dplyr` package (Wickham et al., 2020) for summarizing our simulation results.

```
library("dplyr")

simresults_final <- simresults %>%
  group_by(parameter) %>%
  # Find the average and rounded values
  summarise(bias = round(mean(bias),3),
            rmse = round(mean(rmse),3),
            correlation = round(mean(correlation),3)) %>%
  mutate(nitem = nitem,
         nexaminee = nexaminee,
         guess = guess) %>%
  select(nitem, nexaminee, guess, parameter, bias, rmse, correlation) %>%
```

```
  as.data.frame()

simresults_final
```

```
  nitem nexaminee guess parameter   bias  rmse correlation
1    10      1000    -1         a  0.152 0.479       0.495
2    10      1000    -1         b -0.039 0.420       0.792
3    10      1000    -1         c -0.001 0.130       0.135
```

We can also create a nested loop where we can all the conditions together. Here we use `%:%` to set up three nested loops without curly brackets. The final loop contains `%dopar%` with curly brackets and closes all the loops. Note that we included the final summary function (see how `step3` is summarized) inside these nested loops. This will return a long-format summary data set with all the conditions based on **100** iterations.

```
# Set all the conditions
iterations = 100
seed = sample.int(10000, 100)
nitem = c(10, 15, 20, 25)
nexaminee = c(250, 500, 750, 1000)
guess = c(-1, 0.16)

# Register four clusters
cl <- makeCluster(4)
registerDoParallel(cl)

# Run nested foreach loops
simresults <- foreach(i=1:iterations,
                      .packages = c("mirt", "doParallel", "dplyr"),
                      .combine = rbind) %:%
  foreach(j=nitem,
          .packages = c("mirt", "doParallel", "dplyr"),
          .combine = rbind)  %:%

 foreach(k=nexaminee,
          .packages = c("mirt", "doParallel", "dplyr"),
          .combine = rbind)  %:%

  foreach(m=guess,
          .packages = c("mirt", "doParallel", "dplyr"),
          .combine = rbind)  %dopar% {
            # Generate item parameters and data
            step1 <- generate_data(nitem=j, nexaminee=k, seed=seed[i])
            # Estimate item parameters
            step2 <- estimate_par(step1$respdata, guess = m)
```

```
            # Summarize results
            step3 <- summarize(step2, step1$itempar)
            # Finalize results
            step3 %>%
              group_by(parameter) %>%
              summarise(bias = round(mean(bias),3),
                        rmse = round(mean(rmse),3),
                        correlation = round(mean(correlation),3)) %>%
              mutate(nitem = j,
                     nexaminee = k,
                     guess = m) %>%
              select(nitem, nexaminee, guess, parameter, bias, rmse, correlation) %>%
              as.data.frame()
          }

# Stop the clusters
stopCluster(cl)
```

If you are interested in high performance computing, I also recommend the following resources:

- Lim and Tjhi's book: R High Performance Programming
- The Performance chapter in Hadley Wickham's Advanced R book

---

3.4 Benchmarking

There are several options to measure running time of a Monte Carlo simulation in `R`.

1. Functions included in base `R`:

- `system.time()`
- `Sys.time()`
- `Rprof()` and `summaryRprof()`

2. Packages on benchmarking

- The `tictoc` package (Izrailev, 2020)
- The `rbenchmark` package (Kusnierczyk, 2012)
- The `microbenchmark` package (Mersmann, 2019)
- The `bench` package (Hester, 2020) (check out its website)

3. Progress bar with `txtProgressBar` and `doSNOW` (Corporation and Weston, 2019b)

The three packages, `rbenchmark`, `microbenchmark`, and `bench`, provide detailed information about running time as well as memory usage in `R`. Furthermore, `microbenchmark` and `bench` are capable of visualizing benchmarking results (utilizing `ggplot2`). For more information on benchmarking in `R`, I recommend you to check out this nice blog post.

In the following example, we will run a simple benchmarking test using our simulation. We will use `%do%` and `%dopar%` to see the difference. We will use both `system.time` and `Sys.time` together.

```
iterations = 4 #Eventually this will be 100
seed = sample.int(10000, 100)
nitem = 10 #10, 15, 20, or 25
nexaminee = 1000 #250, 500, 750, or 1000
guess = -1 # A negative value or a value from 0 to 1

# No parallel computing
start_time <- Sys.time() # Starting time
system.time(
  simresults <- foreach(i=1:iterations,
                        .packages = c("mirt", "doParallel"),
                        .combine = rbind) %do% {
                          # Generate item parameters and data
                          step1 <- generate_data(nitem=nitem, nexaminee=nexaminee, seed
                          # Estimate item parameters
                          step2 <- estimate_par(step1$respdata, guess = guess)
                          # Summarize results
                          summarize(step2, step1$itempar)
                        }
)
```

```
   user  system elapsed
   9.89    0.06    9.95
```

```
end_time <- Sys.time() # End time
end_time - start_time # Time difference
```

```
Time difference of 10.11 secs
```

```
# With parallel computing
start_time <- Sys.time() # Starting time
system.time(
  simresults <- foreach(i=1:iterations,
                        .packages = c("mirt", "doParallel"),
                        .combine = rbind) %dopar% {
                          # Generate item parameters and data
                          step1 <- generate_data(nitem=nitem, nexaminee=nexaminee, seed
                          # Estimate item parameters
```

```
                          step2 <- estimate_par(step1$respdata, guess = guess)
                          # Summarize results
                          summarize(step2, step1$itempar)
                        }

)
```

```
   user  system elapsed
   0.04    0.00    3.68
```

```
end_time <- Sys.time() # End time
end_time - start_time # Time difference
```

```
Time difference of 3.827 secs
```

In the output returned from `system.time`, "elapsed" is the time taken to execute the entire process, "user" gives the CPU time spent by the current process (i.e., the current R session), and "system" gives the CPU time spent by the kernel (the operating system) on behalf of the current process (see this post on R-help mailing list for further information).

What if we want to see the time taken by each function in the whole simulation? `Rprof()` provides us with this type of information. To activate profiling in `R`, we need to run:

```
Rprof()
```

Then, we can see the running time of each function after profiling has been activated. We can use `summaryRprof()` to export the results at the end. We wil do this step for a single iteration to see the time distribution across all functions[4].

```
Rprof() # Turn on profiling
# Generate item parameters and data
step1 <- generate_data(nitem=nitem, nexaminee=nexaminee, seed=seed[1])
# Estimate item parameters
step2 <- estimate_par(step1$respdata, guess = guess)
# Summarize results
step3 <- summarize(step2, step1$itempar)
summaryRprof()$by.self
```

```
                     self.time self.pct total.time total.pct
"Estep.mirt"              0.98    35.00       1.12     40.00
"computeItemtrace"        0.42    15.00       0.46     16.43
"LogLikMstep"             0.30    10.71       0.68     24.29
"gr"                      0.20     7.14       0.22      7.86
"reloadPars"              0.12     4.29       0.14      5.00
"EM.group"                0.10     3.57       2.76     98.57
```

[4]The results would be very similar across all iterations using the same conditions.

"fn"	0.10	3.57	0.90	32.14
"@<-"	0.06	2.14	0.06	2.14
"<Anonymous>"	0.04	1.43	1.24	44.29
"Estep"	0.04	1.43	1.16	41.43
"getClass"	0.04	1.43	0.08	2.86
".getClassesFromCache"	0.04	1.43	0.04	1.43
"as.matrix"	0.04	1.43	0.04	1.43
"get0"	0.04	1.43	0.04	1.43
"ifelse"	0.04	1.43	0.04	1.43
"order"	0.04	1.43	0.04	1.43
"evaluate_call"	0.02	0.71	2.80	100.00
"Mstep"	0.02	0.71	1.24	44.29
".External2"	0.02	0.71	1.18	42.14
"getClassDef"	0.02	0.71	0.08	2.86
"getCallingDLLe"	0.02	0.71	0.06	2.14
".requirePackage"	0.02	0.71	0.02	0.71
"matrix"	0.02	0.71	0.02	0.71
"numeric"	0.02	0.71	0.02	0.71
"rev.default"	0.02	0.71	0.02	0.71
"solve"	0.02	0.71	0.02	0.71

```
Rprof(NULL) # Turn off profiling
```

In the output, "self.pct" is the most important column because it indicates the percentage of time that each of the listed tasks has taken in the estimation process.

Next, let's see how the `tictoc` package works for finding running time of our code. Between each `tic()` and `toc()`, it saves the time spent. So, we place several `tic()` and `toc()` functions. The first one, `tic("Total simulation time:")` will be closed at the end so that we can see the total time spent on the simulation. The others in between will calculate the time for each step (i.e., step 1, step 2, and step 3).

```
library("tictoc")
tic("Total simulation time:")
tic("Generate item parameters and data")
step1 <- generate_data(nitem=nitem, nexaminee=nexaminee, seed=seed[1])
toc()
```

```
Generate item parameters and data: 0.02 sec elapsed
```

```
tic("Estimate item parameters")
step2 <- estimate_par(step1$respdata, guess = guess)
toc()
```

```
Estimate item parameters: 3.01 sec elapsed
```

```
tic("Summarize results")
step3 <- summarize(step2, step1$itempar)
toc()
```

```
Summarize results: 0 sec elapsed
```

```
toc()
```

```
Total simulation time:: 3.05 sec elapsed
```

Finally, let's add a progress bar to our simulation study. We will use the `txtProgressBar` function from base R and the `doSNOW` package (Corporation and Weston, 2019b) together. The `txtProgressBar` function is a standalone function and thus it does not require other packages to print a progress bar. However, in the following example, we will do parallel computing using the `doSNOW` package (it is very similar to `doParallel`) and add a progress bar into our simulation. This will require us to add `.options.snow` in the loop so that the progress bar works along with the simulation.

```
iterations = 10 #Eventually this will be 100
seed = sample.int(10000, 100)
nitem = 10 #10, 15, 20, or 25
nexaminee = 1000 #250, 500, 750, or 1000
guess = -1 #A negative value or a value from 0 to 1

library("doSNOW")
cl <- makeCluster(4)
registerDoSNOW(cl)

# Set up the progress bar
pb <- txtProgressBar(max = iterations, style = 3) # Initiate progress bar
progress <- function(n) {setTxtProgressBar(pb, n)}
opts <- list(progress = progress)

# Run the simulation
simresults <- foreach(i=1:iterations,
                      .packages = c("mirt", "doSNOW"),
                      .options.snow = opts, # see the additional line for doSNOW
                      .combine = rbind) %dopar% {
                        # Generate item parameters and data
                        step1 <- generate_data(nitem=nitem, nexaminee=nexaminee, seed=seed[i])
                        # Estimate item parameters
                        step2 <- estimate_par(step1$respdata, guess = guess)
                        # Summarize results
                        summarize(step2, step1$itempar)
                      }
close(pb) # Close progress bar
```

```
stopCluster(cl)
```

As the simulation progresses, we should see a progress bar with a percent sign at the end, which is `style = 3` in the `txtProgressBar` function[5]. Once the simulation is complete, we should see the following screen in the console:

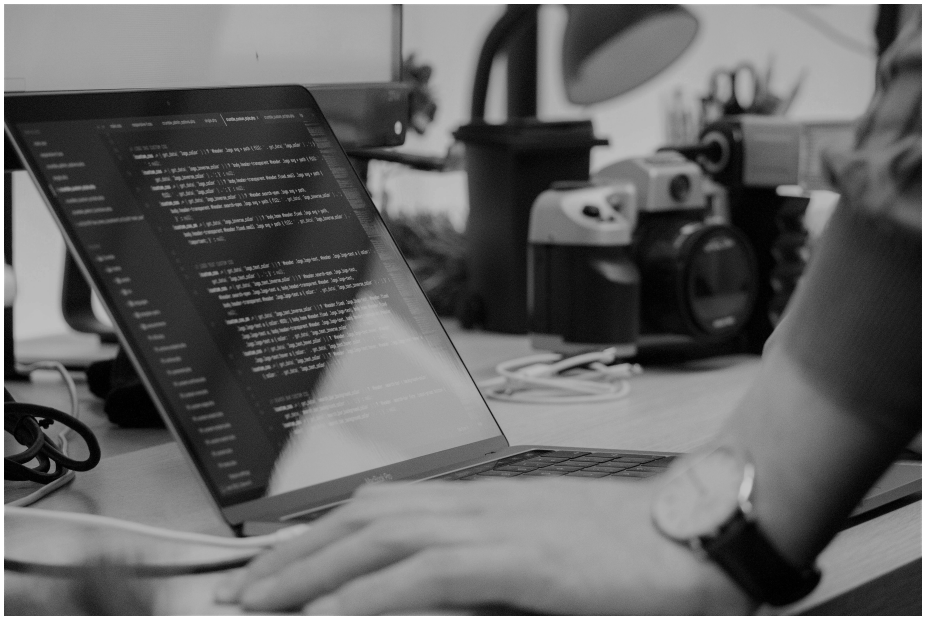
```
R> simresults <- foreach(i=1:iterations,
                         .packages = c("mirt", "doSNOW"),
                         .options.snow = opts, # see the additional line for doSNOW
                         .combine = rbind) %dopar% {
                           # Generate item parameters and data
                           step1 <- generate_data(nitem=nitem, nexaminee=nexaminee, seed=seed[i])
                           # Estimate item parameters
                           step2 <- estimate_par(step1$respdata, guess = guess)
                           # Summarize results
                           summarize(step2, step1$itempar)
                         }
  |=============================================================================================| 100%
```

Figure 3.6: Status of progress bar once the simulation is complete

[5]There are also styles 1 and 2. Check out `?txtProgressBar` for examples.

Chapter 4

Summarizing Simulation Results



4.1 Tables and Figures

For plotting simulation results:

- `ggplot2` (Wickham et al., 2019)
- `lattice` (Sarkar, 2018)

- `plotly` (Sievert et al., 2020)

For making tables with simulation results:

- Check out **R Markdown**
  - RStudio guidelines for `R` Markdown: https://rmarkdown.rstudio.com/
  - `R` Markdown: The Definitive Guide: https://bookdown.org/yihui/rmarkdown/
  - Reproducible APA manuscripts with `R` Markdown: https://crsh.github.io/papaja_man/reporting.html
  - HTML tables with `knitr::kable` and `kableExtra`: http://haozhu233.github.io/kableExtra/
  - A nice blog post about stylish tables in `R`: https://www.littlemissdata.com/blog/prettytables

4.2 Exporting the Results

I recommend exporting all the simulation results if:

- writing all the results into a file would not take a lot of space in the computer
- existing computations are heavy and thus it is hard to summarize the results right away
- it is likely that the simulation may be interrupted for some reason

Regardless what we decide to save (either all output or the summarized output), a nice way to save the results is to place a `write` function inside the loop. Using the nested `foreach` loops as an example, we will first create an empty .csv file called "results.csv":

```
write.table(matrix(c("nitem", "nexaminee", "guess", "parameter", "bias", "rmse", "corre
                     nrow = 1,
                     ncol = 7),
            file = "results.csv",
            sep = ",",
            col.names = FALSE,
            row.names = FALSE)
```

Then, we modify our code in a way that we will **not** save the results and instead write them into the results.csv file.

```
# Set all the conditions
iterations = 100
seed = sample.int(10000, 100)
nitem = c(10, 15, 20, 25)
nexaminee = c(250, 500, 750, 1000)
guess = c(-1, 0.16)
```

```r
# Register four clusters
cl <- makeCluster(4)
registerDoParallel(cl)

# Run nested foreach loops
foreach(i=1:iterations,
        .packages = c("mirt", "doParallel", "dplyr"),
        .combine = rbind) %:%

  foreach(j=nitem,
          .packages = c("mirt", "doParallel", "dplyr"),
          .combine = rbind)  %:%

 foreach(k=nexaminee,
          .packages = c("mirt", "doParallel", "dplyr"),
          .combine = rbind)  %:%

  foreach(m=guess,
          .packages = c("mirt", "doParallel", "dplyr"),
          .combine = rbind)  %dopar% {
            # Generate item parameters and data
            step1 <- generate_data(nitem=j, nexaminee=k, seed=seed[i])
            # Estimate item parameters
            step2 <- estimate_par(step1$respdata, guess = m)
            # Summarize results
            step3 <- summarize(step2, step1$itempar)
            # Finalize results
            final <- step3 %>%
              group_by(parameter) %>%
              summarise(bias = round(mean(bias),3),
                        rmse = round(mean(rmse),3),
                        correlation = round(mean(correlation),3)) %>%
              mutate(nitem = j,
                     nexaminee = k,
                     guess = m) %>%
              select(nitem, nexaminee, guess, parameter, bias, rmse, correlation) %>%
              as.data.frame()
            # Write the results
            write.table(final, "results.csv",
                        sep = ",",
                        col.names = FALSE,
                        row.names = FALSE,
                        append = TRUE) # This will keep the file open for appending the results
            }
```

```
# Stop the clusters
stopCluster(cl)
```

Bibliography

Bulut, O. and Sunbul, O. (2017). Monte carlo simulation studies in item response theory with the r programming language. *Journal of Measurement and Evaluation in Education and Psychology*, 8(3):266–287.

Casabianca, J. M. and Lewis, C. (2015). Irt item parameter recovery with marginal maximum likelihood estimation using loglinear smoothing models. *Journal of Educational and Behavioral Statistics*, 40(6):547–578.

Chalmers, P. (2019). *mirt: Multidimensional Item Response Theory*. R package version 1.31.

Chalmers, P. (2020). *SimDesign: Structure for Organizing Monte Carlo Simulation Designs*. R package version 2.0.1.

Corporation, M. and Weston, S. (2019a). *doParallel: Foreach Parallel Adaptor for the 'parallel' Package*. R package version 1.0.15.

Corporation, M. and Weston, S. (2019b). *doSNOW: Foreach Parallel Adaptor for the 'snow' Package*. R package version 1.0.18.

Hallgren, K. A. (2013). Conducting simulation studies in the r programming environment. *Tutorials in Quantitative Methods for Psychology*, 9(2):43–60.

Hester, J. (2020). *bench: High Precision Timing of R Expressions*. R package version 1.1.1.

Izrailev, S. (2020). *tictoc: Functions for timing R scripts, as well as implementations of Stack and List structures*. R package version 1.0.1.

Kusnierczyk, W. (2012). *rbenchmark: Benchmarking routine for R*. R package version 1.0.0.

Leschinski, C. H. (2019). *MonteCarlo: Automatic Parallelized Monte Carlo Simulations*. R package version 1.0.6.

Mersmann, O. (2019). *microbenchmark: Accurate Timing Functions*. R package version 1.4-7.

Microsoft and Weston, S. (2020). *foreach: Provides Foreach Looping Construct*. R package version 1.4.8.

Partchev, I. and Maris, G. (2017). *irtoys: A Collection of Functions Related to Item Response Theory (IRT)*. R package version 0.2.1.

R Core Team (2019). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Sarkar, D. (2018). *lattice: Trellis Graphics for R*. R package version 0.20-38.

Sievert, C., Parmer, C., Hocking, T., Chamberlain, S., Ram, K., Corvellec, M., and Despouy, P. (2020). *plotly: Create Interactive Web Graphics via 'plotly.js'*. R package version 4.9.2.

Wickham, H., Chang, W., Henry, L., Pedersen, T. L., Takahashi, K., Wilke, C., Woo, K., and Yutani, H. (2019). *ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics*. R package version 3.2.1.

Wickham, H., François, R., Henry, L., and Müller, K. (2020). *dplyr: A Grammar of Data Manipulation*. R package version 0.8.4.

Xie, Y. (2020a). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.17.

Xie, Y. (2020b). *knitr: A General-Purpose Package for Dynamic Report Generation in R*. R package version 1.28.