Getting and Organizing Outputs from Simulation Studies Using R

Sunthud Pornprasertmanit

June 15, 2015

In the previous class, a basic R code for simulation study was introduced. The for loop is used to repeat the same tasks or to alter the design conditions of the tasks. We just evaluated the Type I error only, however. In this class, we will evaluate the biases in parameter estimates, the biases in standard errors, and statistical power. The graphical tool in R is used to plot the results. Finally, we will see how to create functions to help us in writing codes.

The example in this note is to check the bias in parameter estimates, the bias in standard error, and statistical power of regression coefficients in simple regression. Simple regression, which is a submodel in general linear model, can be analyzed by the `lm` function as follows:

```
> out <- lm(complaints ~ advance, data = attitude)
> out

Call:
lm(formula = complaints ~ advance, data = attitude)

Coefficients:
(Intercept)      advance
    54.1222       0.2906
```

We will need the p-value to find statistical power. To get the p-value from the result of the `lm` function, the `summary` function is used.

```
> summary(out)

Call:
lm(formula = complaints ~ advance, data = attitude)

Residuals:
     Min       1Q   Median       3Q      Max
-31.3632  -8.7943   0.8682   9.3720  25.4150

Coefficients:
```

```
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  54.1222    10.5119   5.149 1.85e-05 ***
advance       0.2906     0.2383   1.220    0.233
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 13.2 on 28 degrees of freedom
Multiple R-squared:  0.05044,        Adjusted R-squared:  0.01652
F-statistic: 1.487 on 1 and 28 DF,  p-value: 0.2328
```

The p-value of the slope was .23. Then, we need to find an appropriate way to extract the p-value. To do it, the result of the `summary` function is saved. Then, the `names` function is used to find the appropriate objects, which is `"coefficients"`. The table contains the slope estimate, its standard error, and its p value.

```
> summaryout <- summary(out)
> names(summaryout)

 [1] "call"          "terms"         "residuals"     "coefficients"
 [5] "aliased"       "sigma"         "df"            "r.squared"
 [9] "adj.r.squared" "fstatistic"    "cov.unscaled"

> summary(out)[["coefficients"]]

             Estimate Std. Error  t value     Pr(>|t|)
(Intercept) 54.1222203 10.5119281 5.148648 1.847247e-05
advance      0.2906315  0.2383173 1.219515 2.328224e-01

> pvalue <- summary(out)[["coefficients"]][2, 4]
> est <- summary(out)[["coefficients"]][2, 1]
> se <- summary(out)[["coefficients"]][2, 2]
```

Let's analyze simple regression from a generated data set using the `rnorm` function.

```
> n <- 200
> b <- 0.5
> x <- rnorm(n, 0, 1)
> e <- rnorm(n, 0, sqrt(1 - b^2))
> y <- b*x + e
> dat <- data.frame(x = x, y = y)
> out <- lm(y ~ x, data = dat)
> summary(out)[["coefficients"]][2, c(1, 2, 4)]

    Estimate   Std. Error     Pr(>|t|)
5.382498e-01 6.012155e-02 2.536023e-16
```

In this code, `x`, `y`, `e` represent independent variable, dependent variable, and error, respectively. The standard deviation of the error is $\sqrt{1-b^2}$ because it makes the population standard deviation of the dependent variable of 1. The regression coefficient is equivalent to standardized coefficient. This code could be repeated multiple times and the parameter estimates, standard errors, and p values across replications are saved:

```
> set.seed(123321)
> nrep <- 1000
> n <- 200
> b <- 0.5
> result <- matrix(NA, nrep, 3)
> for(i in 1:nrep) {
+   x <- rnorm(n, 0, 1)
+   e <- rnorm(n, 0, sqrt(1 - b^2))
+   y <- b*x + e
+   dat <- data.frame(x = x, y = y)
+   out <- lm(y ~ x, data = dat)
+   result[i,] <- summary(out)[["coefficients"]][2, c(1, 2, 4)]
+ }
```

The for loop is used to generate data and analyze data by simple regression multiple times. The `result` object is a matrix saving parameter estimates, standard errors, and p values in Columns 1, 2, and 3, respectively. Then, the `sig` vector is attached to indicate which replication is significant. Then, the `colMeans` function is used to find the average of each column:

```
> sig <- result[,3] < 0.05
> result <- cbind(result, sig)
> colnames(result) <- c("est", "se", "p", "sig")
> colMeans(result)

         est           se            p          sig
4.957004e-01 6.159070e-02 1.414257e-08 1.000000e+00
```

The last value indicates statistical power, which is 1. That is, all replications provided significant results. The bias in parameter estimates is the averaged parameter estimates subtracted by the population value:

```
> mean(result[,1]) - b

[1] -0.004299648
```

The relative bias in parameter estimates is the bias divided by the population value. Note that if the population value is 0, the relative bias is not meaningful.

```
> (mean(result[,1]) - b) / b
```

```
[1] -0.008599295
```

Hoogland and Boomsma [1998] provided a guideline that the relative bias in parameter estimates is acceptable if its absolute value is not over 0.05. The bias in standard errors is the averaged standard errors (observed standard errors) subtracted by the standard deviation of parameter estimates (empirical standard errors):

```
> mean(result[,2]) - sd(result[,1])
```

```
[1] 0.001568895
```

The relative bias in standard error is the bias in standard error divided by the empirical standard errors:

```
> (mean(result[,2]) - sd(result[,1])) / sd(result[,1])
```

```
[1] 0.02613875
```

The relative bias in standard errors is acceptable if its absolute value is not over 0.1 [Hoogland and Boomsma, 1998].

Let's run this Monte Carlo simulation across different design conditions. Let sample size be 50, 100, 150, and 200 and let slope be 0, 0.1, 0.2, 0.3, 0.4, and 0.5. Following the previous class, we can use nested for loop to change design conditions:

```
> set.seed(123321)
> nrep <- 1000
> bs <- seq(0, 0.5, 0.1)
> ns <- seq(50, 200, 50)
> ovresult <- NULL
> for(k in 1:length(bs)){
+   b <- bs[k]
+   for(j in 1:length(ns)) {
+     n <- ns[j]
+     result <- matrix(NA, nrep, 3)
+     for(i in 1:nrep) {
+       x <- rnorm(n, 0, 1)
+       e <- rnorm(n, 0, sqrt(1 - b^2))
+       y <- b*x + e
+       dat <- data.frame(x = x, y = y)
+       out <- lm(y ~ x, data = dat)
+       result[i,] <- summary(out)[["coefficients"]][2, c(1, 2, 4)]
+     }
+     result <- cbind(b, n, result)
+     ovresult <- rbind(ovresult, result)
+   }
+ }
> sig <- ovresult[,5] < 0.05
> ovresult <- cbind(ovresult, sig)
> colnames(ovresult) <- c("b", "n", "est", "se", "p", "sig")
```

The `aggregate` function is used to find the averaged parameter estimates, the averaged standard errors, and statistical power of each design condition:

```
> aggresult <- aggregate(cbind(est, se, sig) ~ n + b, data = ovresult,
+   FUN = mean)
```

Note that `cbind` is used in the formula to set multiple response variables. The empirical standard errors can be calculated by the `aggregate` function using `sd` in the `FUN` argument:

```
> aggsdresult <- aggregate(est ~ n + b, data = ovresult, FUN = sd)
```

Then, the bias in parameter estimates, the relative bias in parameter estimates, the bias in standard error, the relative bias in standard error, and statistical power can be calculated:

```
> biasest <- aggresult[,"est"] - aggresult[,"b"]
> relbiasest <- biasest / aggresult[,"b"]
> biasse <- aggresult[,"se"] - aggsdresult[,"est"]
> relbiasse <- biasse / aggsdresult[,"est"]
> power <- aggresult[,"sig"]
> tableresult <- data.frame(aggresult[,c(1, 2)], biasest, relbiasest,
+   biasse, relbiasse, power)
> tableresult
```

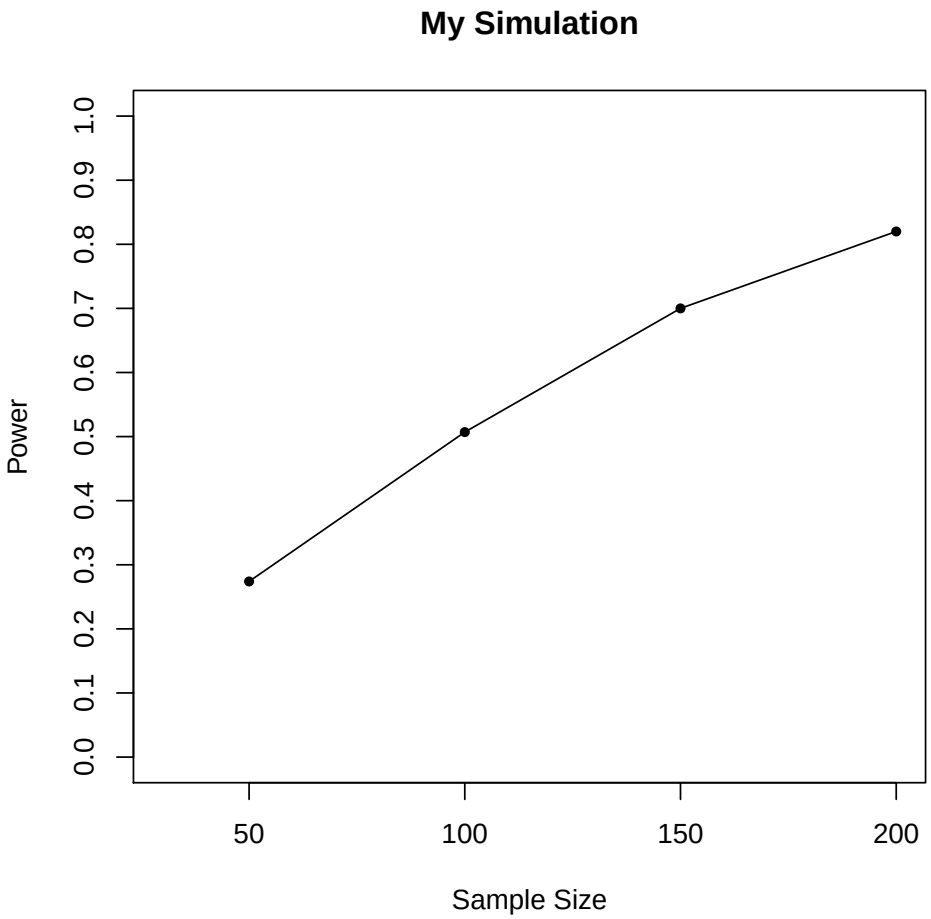
	n	b	biasest	relbiasest	biasse	relbiasse	power
1	50	0.0	-0.0025449724	-Inf	0.0008705738	0.006078938	0.043
2	100	0.0	-0.0009160590	-Inf	0.0004171006	0.004147008	0.047
3	150	0.0	-0.0001628894	-Inf	0.0031891959	0.040326010	0.040
4	200	0.0	0.0032798188	Inf	0.0004637597	0.006559787	0.040
5	50	0.1	-0.0031789529	-0.0317895294	0.0045604665	0.032601003	0.103
6	100	0.1	-0.0038724304	-0.0387243038	0.0001647485	0.001641587	0.161
7	150	0.1	-0.0015016226	-0.0150162264	0.0011449650	0.014177776	0.225
8	200	0.1	0.0023542600	0.0235426000	-0.0008452670	-0.011798156	0.321
9	50	0.2	-0.0006287235	-0.0031436177	0.0010263234	0.007375258	0.274
10	100	0.2	-0.0028844725	-0.0144223623	-0.0010005039	-0.010026082	0.507
11	150	0.2	-0.0011602905	-0.0058014526	-0.0011154595	-0.013713530	0.700
12	200	0.2	-0.0001023953	-0.0005119767	0.0024445877	0.036421985	0.820
13	50	0.3	-0.0017582840	-0.0058609467	-0.0034239962	-0.024240807	0.560
14	100	0.3	-0.0017446371	-0.0058154569	-0.0006798732	-0.006987941	0.851
15	150	0.3	0.0011626951	0.0038756504	0.0003224840	0.004128171	0.967
16	200	0.3	-0.0008407637	-0.0028025456	0.0001199815	0.001777146	0.993
17	50	0.4	0.0051079393	0.0127698482	0.0029054159	0.022365620	0.848
18	100	0.4	0.0062974297	0.0157435743	-0.0015790544	-0.016795849	0.986
19	150	0.4	-0.0004237251	-0.0010593127	-0.0001239285	-0.001639510	0.999
20	200	0.4	-0.0011192813	-0.0027982033	0.0017343393	0.027248927	1.000
21	50	0.5	-0.0005707375	-0.0011414750	0.0031769838	0.026104040	0.978
22	100	0.5	-0.0029018981	-0.0058037961	-0.0002835439	-0.003224112	0.999
23	150	0.5	0.0005541108	0.0011082217	-0.0012898151	-0.017752410	1.000
24	200	0.5	0.0018274760	0.0036549519	0.0012116184	0.020106452	1.000

The `data.frame` function is used to combine the multiple vectors into a single table. Next, let's make a graph to visualize the patterns of the results. The `plot` is usually used to create scatterplot or line graphs. Let's make a power plot when the regression coefficient is 0.2. The X and Y axes represent sample size and power, respectively.

```
> coord <- tableresult[tableresult,"b"] == 0.2, c("n", "power")]
> plot(coord, type="l", xlab="Sample Size", ylab="Power",
+   main = "My Simulation")
```

However, R provides a tool that can manipulate each element in the graph. Let's try to create a graph by successively adding the individual elements.

```
> plot(c(0,0), type = "n", xlim=c(30, 200), ylim=c(0, 1), xlab="Sample Size",
+   ylab="Power", main = "My Simulation", axes = FALSE)
> axis(1, at=c(50, 100, 150, 200))
> axis(2, at=seq(0, 1, 0.1))
> box()
> points(coord, pch = 20)
> lines(coord, lty = 1)
```

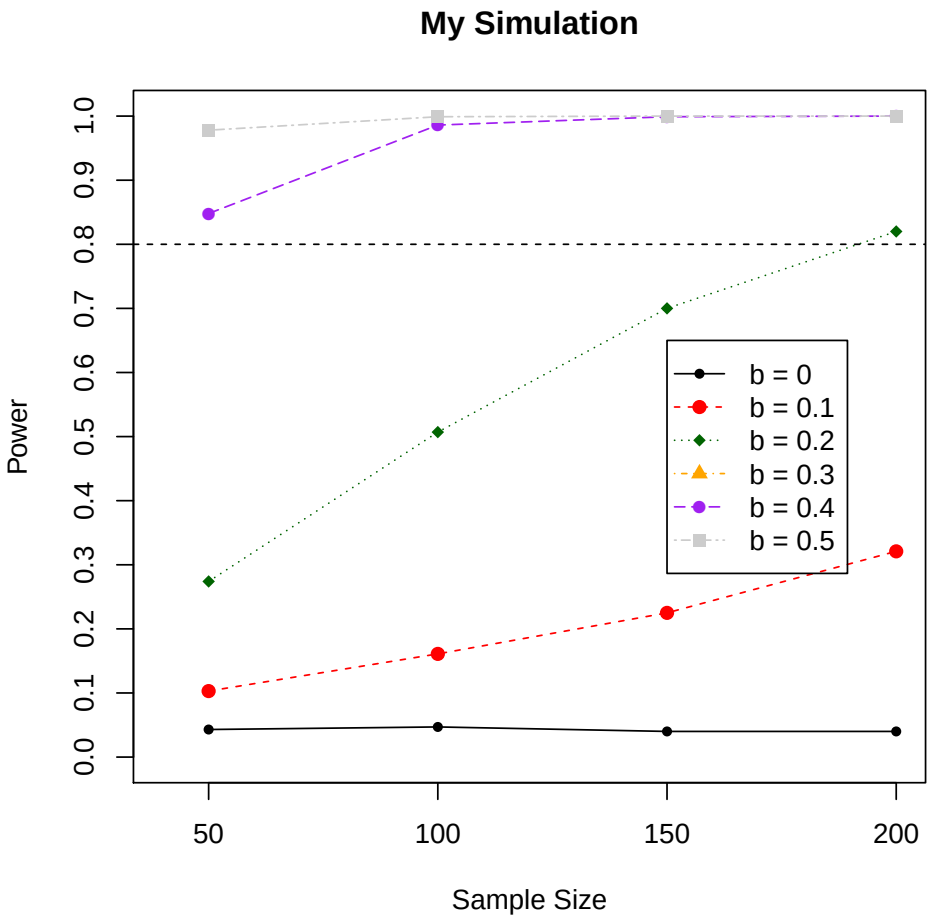


Lines 1-2 used the `plot` function to create a template for the graph. The first argument could be any arbitrary coordinate. This point will not be put in the graph because `type = "n"` that creates blank graph is used. Other arguments are similar to a regular `plot` function except that `axes = FALSE`. We will tailor the axes in Lines 3-4. The first argument of the `axis` function indicates the side of the axes: $1 = X$-axis and $2 = Y$-axis. The `at` argument indicates the scale in the axes. Then, the `box` function creates a box around the graph. The `points` function put the data points (representing by rows in a two-column matrix) in the graph. The `pch` argument sets the appearance of the data points. The `lines` function put the lines of data in the graph. The `lty` argument sets the appearance of the lines. We can google "pch R" and "lty R" to explore the options.

Now, we can plot graphs where multiple lines represent different the power for regression coefficients.

```
> plot(c(0,0), type = "n", xlim=c(40, 200), ylim=c(0, 1), xlab="Sample Size",
+   ylab="Power", main = "My Simulation", axes = FALSE)
> axis(1, at=c(50, 100, 150, 200))
> axis(2, at=seq(0, 1, 0.1))
> box()
> points(tableresult[tableresult,"b"] == 0, c("n", "power")],
```

```
+   pch = 20, col = "black")
> lines(tableresult[tableresult[,"b"] == 0, c("n", "power")],
+   lty = 1, col = "black")
> points(tableresult[tableresult[,"b"] == 0.1, c("n", "power")],
+   pch = 19, col = "red")
> lines(tableresult[tableresult[,"b"] == 0.1, c("n", "power")],
+   lty = 2, col = "red")
> points(tableresult[tableresult[,"b"] == 0.2, c("n", "power")],
+   pch = 18, col = "darkgreen")
> lines(tableresult[tableresult[,"b"] == 0.2, c("n", "power")],
+   lty = 3, col = "darkgreen")
> points(tableresult[tableresult[,"b"] == 0.3, c("n", "power")],
+   pch = 17, col = "orange")
> lines(tableresult[tableresult[,"b"] == 0.3, c("n", "power")],
+   lty = 4, col = "orange")
> points(tableresult[tableresult[,"b"] == 0.4, c("n", "power")],
+   pch = 16, col = "purple")
> lines(tableresult[tableresult[,"b"] == 0.4, c("n", "power")],
+   lty = 5, col = "purple")
> points(tableresult[tableresult[,"b"] == 0.5, c("n", "power")],
+   pch = 15, col = "grey80")
> lines(tableresult[tableresult[,"b"] == 0.5, c("n", "power")],
+   lty = 6, col = "grey80")
> abline(h = 0.8, lty = 2)
> legend(x = 150, y = 0.65,
+   legend = c("b = 0", "b = 0.1", "b = 0.2", "b = 0.3",
+   "b = 0.4", "b = 0.5"),
+   pch = c(20, 19, 18, 17, 16, 15), lty = 1:6,
+   col = c("black", "red", "darkgreen", "orange", "purple", "grey80"))
```



These codes are similar to the previous code. The additional codes are to put multiple `points` and `lines` to represent multiple power lines. The `pch`, `lty`, and `col` arguments are different across different power lines. Next, the `abline` function is used to create horizontal line by specifying the `h` argument at 0.8. Note that the `abline` function can be used to create linear regression line, horizontal line, or vertical line. Finally, the `legend` is used to create the legend in the graph. The `x` and `y` is used to specify the coordinate of the upper-left corner of the legend. The `legend` argument specifies the narratives of each line. The `pch`, `lty`, and `col` arguments specify the characteristics of each line. The graphs for the bias in parameter estimates or the bias in standard errors can be created by simply adjusting the codes for statistical power.

We can use the for loop to do repeated task again (i.e., the `points` and `lines` functions). The more flexible approach is to create a function to do repeated components. Let's make an example of a simple function changing temperatures from Celsius to Fahrenheit:

```
> convertCtoF <- function(temp) {
+   f <- temp * (9/5) + 32
+   return(f)
+ }
> convertCtoF(25)
```

```
[1] 77
```

The `convertCtoF` is assigned as a function that takes one argument, `temp`. Then, the codes inside the function will use the `temp` argument to create a desired output. The output is exported from the function by `return`. In fact, in this situation (and in most situations), `return` is not required. This function can be written in other forms:

```
> convertCtoF2 <- function(temp) {
+   f <- temp * (9/5) + 32
+   f
+ }
> convertCtoF3 <- function(temp) {
+   temp * (9/5) + 32
+ }
> convertCtoF4 <- function(temp) temp * (9/5) + 32
> convertCtoF2(25)

[1] 77

> convertCtoF3(25)

[1] 77

> convertCtoF4(25)

[1] 77
```

The object evaluated in the last line is automatically returned. Furthermore, if the function has only one line, curly bracket is not required. As another example, we can create a function for calculating relative bias:

```
> relbias <- function(v1, v2) {
+   result <- (v1 - v2) / v2
+   return(result)
+ }
> relbias(102, 100)

[1] 0.02

> relbias(aggresult[,3], aggresult[,2])

 [1]          -Inf          -Inf          -Inf           Inf -0.0317895294
 [6] -0.0387243038 -0.0150162264  0.0235426000 -0.0031436177 -0.0144223623
[11] -0.0058014526 -0.0005119767 -0.0058609467 -0.0058154569  0.0038756504
[16] -0.0028025456  0.0127698482  0.0157435743 -0.0010593127 -0.0027982033
[21] -0.0011414750 -0.0058037961  0.0011082217  0.0036549519
```

This function takes two arguments and calculates the relative bias using two inputs.

If you noticed the previous graph, in most machines, you could not see the orange line representing the power for regression coefficient of 0.3. I was confused at the first place as well. The codes are supposed to be correct. The problem is actually the nature of R computation. Let's see the following example:

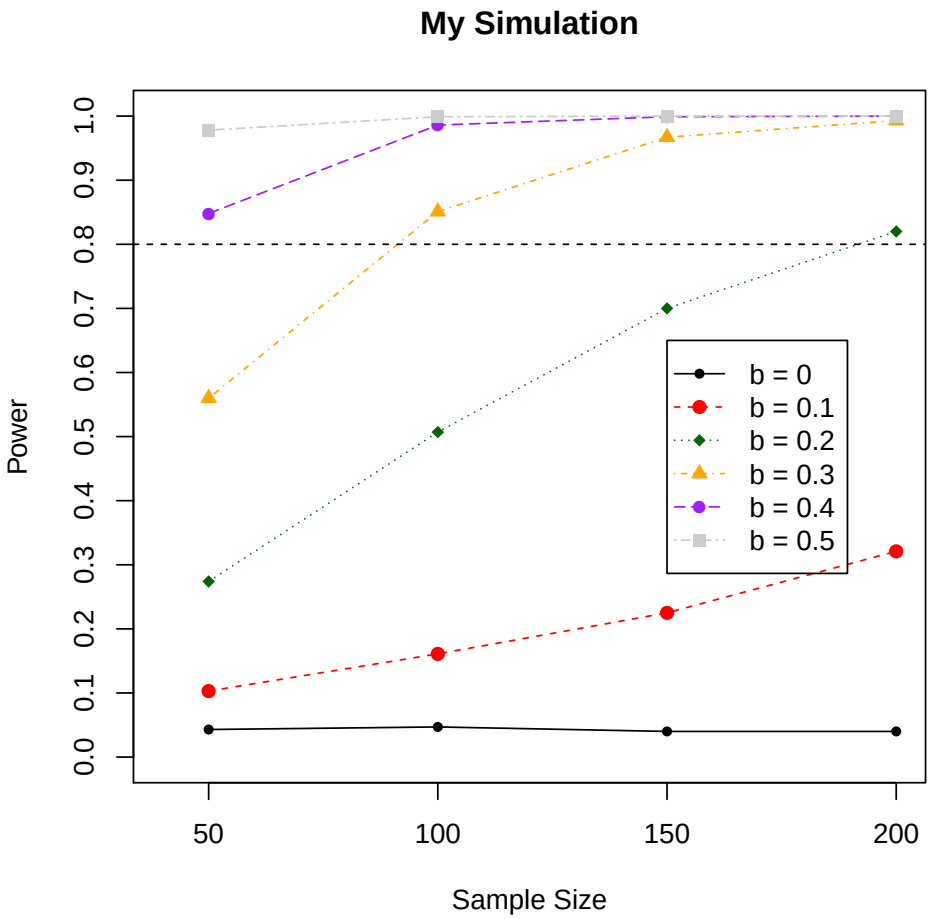
```
> p <- 0.3
> q <- 0.3 - 0.1 + 0.1
> p == q

[1] TRUE

> a <- 0.5 - 0.3
> b <- 0.3 - 0.1
> a == b

[1] FALSE

> x <- seq(0, 0.5, 0.1)[4]
> y <- 0.3
> x == y

[1] FALSE
```

`a` and `b`, as well as `x` and `y`, are not exactly equal because R computation may make these two numbers different in a very very tiny decimal point. Thus, they are not exactly equal. See `?"=="` for the details. We could make a function to check whether two numbers are close enough, however:

```
> equal <- function(a, b) {
+   abs(a - b) < 0.000001
+ }
> equal(p, q)

[1] TRUE

> equal(a, b)

[1] TRUE

> equal(x, y)

[1] TRUE
```

Let's use the function feature to help us plot the graphs. We can see from the plotting codes that we repeatedly used the `points` and `lines` functions. Let's make a function to make the codes clearer.

```
> putLine <- function(mat, lty, pch, col) {
+   points(mat, pch = pch, col = col)
+   lines(mat, lty = lty, col = col)
+ }
> legendtxt <- c("b = 0", "b = 0.1", "b = 0.2", "b = 0.3",
+   "b = 0.4", "b = 0.5")
> lty <- 1:6
> pch <- c(20, 19, 18, 17, 16, 15)
> col <- c("black", "red", "darkgreen", "orange", "purple", "grey80")
> plot(c(0,0), type = "n", xlim=c(40, 200), ylim=c(0, 1), xlab="Sample Size",
+   ylab="Power", main = "My Simulation", axes = FALSE)
> axis(1, at=c(50, 100, 150, 200))
> axis(2, at=seq(0, 1, 0.1))
> box()
> putLine(tableresult[equal(tableresult[,"b"], 0), c("n", "power")],
+   lty = lty[1], pch = pch[1], col = col[1])
> putLine(tableresult[equal(tableresult[,"b"], 0.1), c("n", "power")],
+   lty = lty[2], pch = pch[2], col = col[2])
> putLine(tableresult[equal(tableresult[,"b"], 0.2), c("n", "power")],
+   lty = lty[3], pch = pch[3], col = col[3])
> putLine(tableresult[equal(tableresult[,"b"], 0.3), c("n", "power")],
+   lty = lty[4], pch = pch[4], col = col[4])
> putLine(tableresult[equal(tableresult[,"b"], 0.4), c("n", "power")],
+   lty = lty[5], pch = pch[5], col = col[5])
> putLine(tableresult[equal(tableresult[,"b"], 0.5), c("n", "power")],
+   lty = lty[6], pch = pch[6], col = col[6])
> abline(h = 0.8, lty = 2)
> legend(x = 150, y = 0.65, legend = legendtxt, pch = pch, lty = lty,
+   col = col)
```

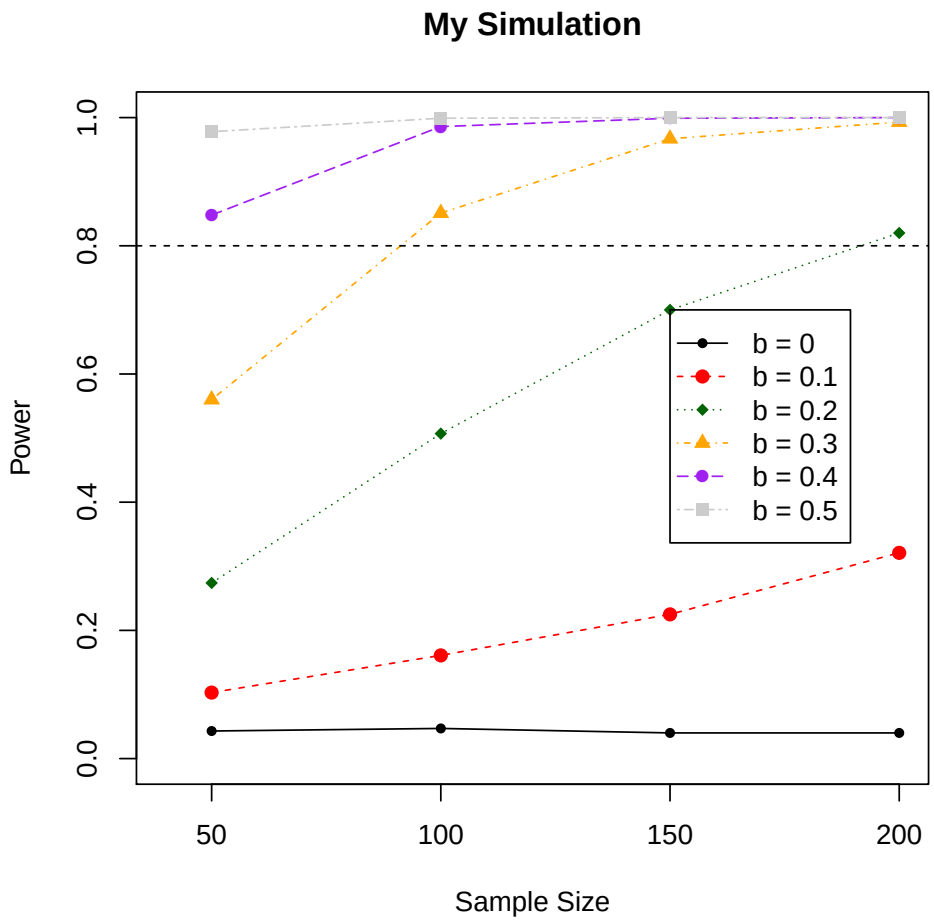


Note that the `putLine` function does not provide any outputs but rather run the `points` and `lines` functions. Four arguments are used: desired points in a matrix format (`mat`), line style (`lty`), point style (`pch`), and color (`col`). In this code, the `putLine` function is used six times to plot the power lines. In selecting desired rows of the matrices, the `equal` function is used instead of `==`.

Let's make another function to plot the whole graph:

```
> plotResult <- function(mat, targetcol, aty, ylab, poslegend, hline) {
+   lty <- 1:6
+   pch <- c(20, 19, 18, 17, 16, 15)
+   col <- c("black", "red", "darkgreen", "orange", "purple", "grey80")
+   legendtxt <- c("b = 0", "b = 0.1", "b = 0.2", "b = 0.3", "b = 0.4", "b = 0.5")
+   plot(c(0,0), type = "n", xlim=c(40, 200), ylim=range(aty), xlab="Sample Size",
+   ylab=ylab, main = "My Simulation", axes = FALSE)
+   axis(1, at=c(50, 100, 150, 200))
+   axis(2, at=aty)
+   box()
+   putLine(mat[equal(mat[,"b"], 0), c("n", targetcol)], lty = lty[1],
+     pch = pch[1], col = col[1])
```

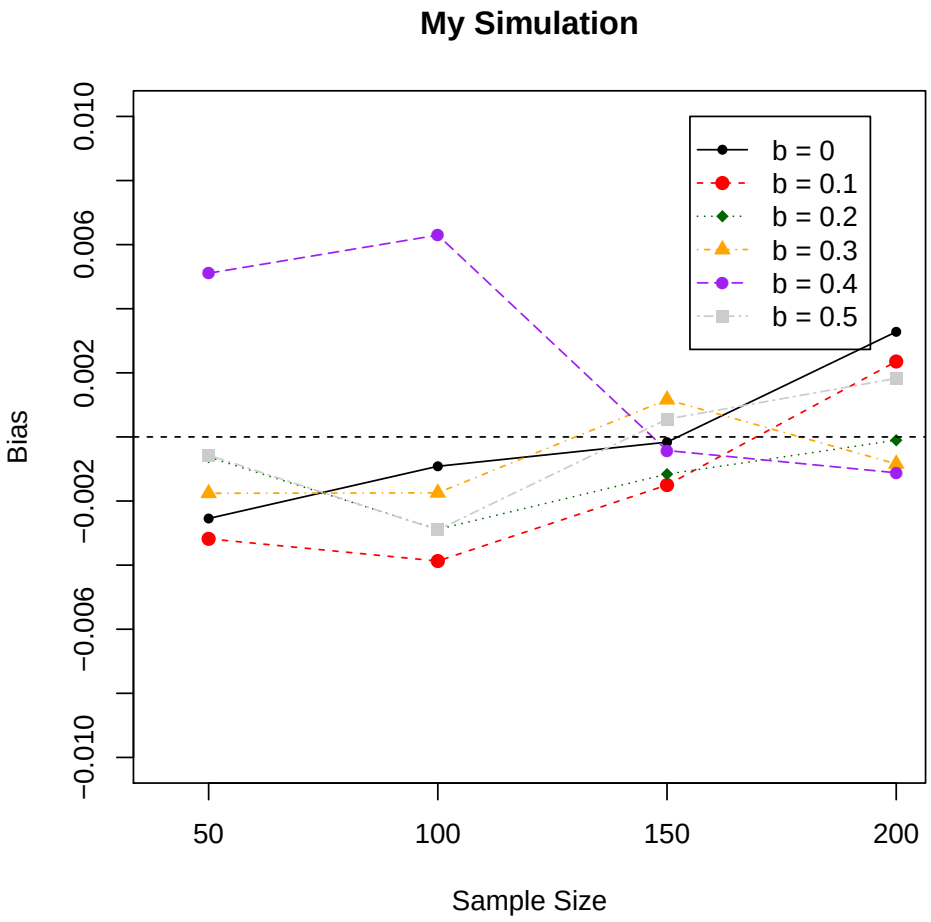
```
+   putLine(mat[equal(mat[,"b"], 0.1), c("n", targetcol)], lty = lty[2],
+     pch = pch[2], col = col[2])
+   putLine(mat[equal(mat[,"b"], 0.2), c("n", targetcol)], lty = lty[3],
+     pch = pch[3], col = col[3])
+   putLine(mat[equal(mat[,"b"], 0.3), c("n", targetcol)], lty = lty[4],
+     pch = pch[4], col = col[4])
+   putLine(mat[equal(mat[,"b"], 0.4), c("n", targetcol)], lty = lty[5],
+     pch = pch[5], col = col[5])
+   putLine(mat[equal(mat[,"b"], 0.5), c("n", targetcol)], lty = lty[6],
+     pch = pch[6], col = col[6])
+   abline(h = hline, lty = 2)
+   legend(x = poslegend[1], y = poslegend[2], legend = legendtxt,
+     pch = pch, lty = lty, col = col)
+ }
> plotResult(tableresult, "power", seq(0, 1, 0.2), ylab = "Power",
+   poslegend = c(150, 0.7), hline = 0.8)
```



The codes for plotting graphs are simply modified and put in the function. The `plotResult` function takes six arguments: our data frame containing data (`mat`), target dependent

variable (`targetcol`), scale at the Y axis (`aty`), the label of the Y axis (`ylab`), the position of the legend (`poslegend`), the horizontal line (`hline`). These six arguments substitute the appropriate objects inside the function. For example, `mat` substitutes `tableresult`. Note that the `putLine` function is used in the `plotResult` function because the `putLine` function has already been defined in the R workspace. The `plotResult` function can be used to make the graph for the bias in parameter estimates:

```
> plotResult(tableresult, "biasest", seq(-0.01, 0.01, 0.002),
+   ylab = "Bias", poslegend = c(155, 0.01), hline = 0)
```

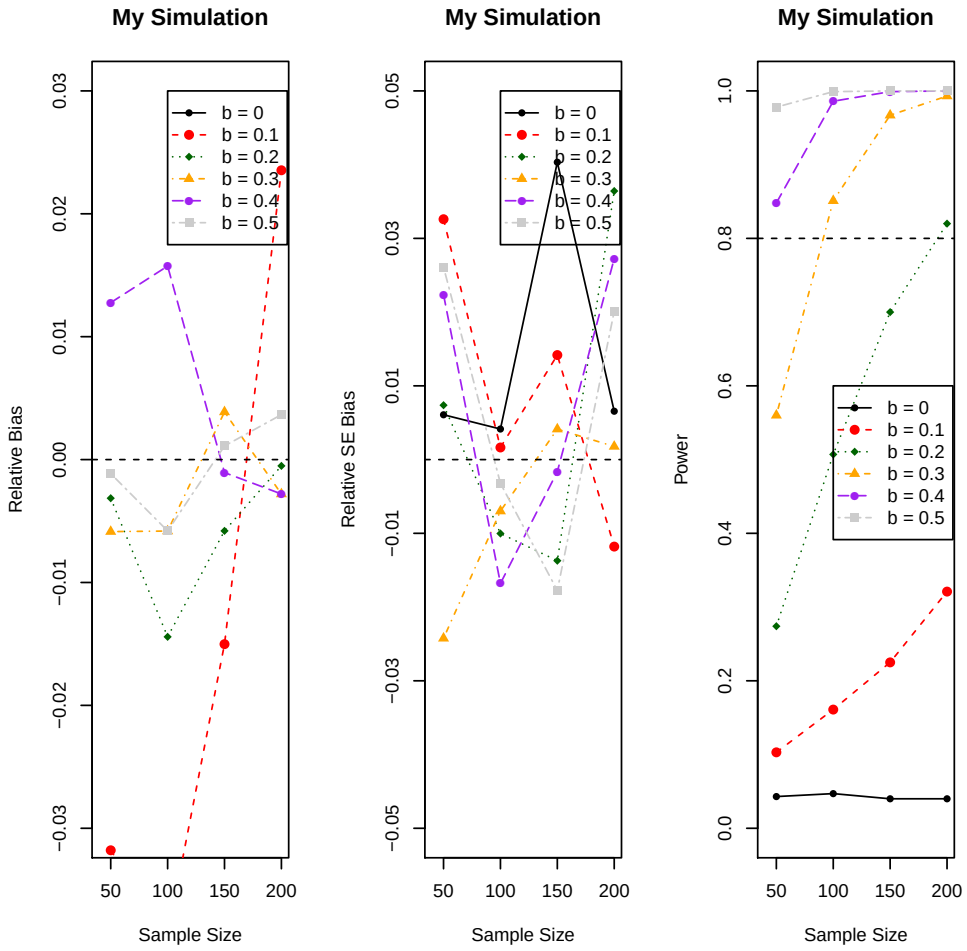


The relative bias in parameter estimates, the bias in standard errors, and the relative bias in standard errors can be plotted. Note that these graphs are not shown to save space.

```
> plotResult(tableresult, "relbiasest", seq(-0.03, 0.03, 0.01),
+   ylab = "Relative Bias", poslegend = c(150, 0), hline = 0)
> plotResult(tableresult, "biasse", seq(-0.004, 0.004, 0.002),
+   ylab = "SE Bias", poslegend = c(150, 0), hline = 0)
> plotResult(tableresult, "relbiasse", seq(-0.05, 0.05, 0.02),
+   ylab = "Relative SE Bias", poslegend = c(150, 0), hline = 0)
```

You may plot multiple graphs in the same figure by using the `par` function:

```
> par(mfrow=c(1,3))
> plotResult(tableresult, "relbiasest", seq(-0.03, 0.03, 0.01),
+   ylab = "Relative Bias", poslegend = c(100, 0.03), hline = 0)
> plotResult(tableresult, "relbiasse", seq(-0.05, 0.05, 0.02),
+   ylab = "Relative SE Bias", poslegend = c(100, 0.05), hline = 0)
> plotResult(tableresult, "power", seq(0, 1, 0.2), ylab = "Power",
+   poslegend = c(100, 0.6), hline = 0.8)
```



This figure plots three graphs: the relative bias on parameter estimates, the relative bias in standard errors, and power. The `par` function has one argument, `mfrow`, to specify the dimensions of the figure. The figure has three graphs in one row and three columns. If you plot a new graph, it will replace one of the graphs in this figure. You may cancel this feature by simply closing the window or type `dev.off()`.

1 Exercise

Use the following code to help you evaluate the relative bias in parameter estimates, the relative bias in standard errors, and statistical power for the slope in simple logistic regression. The logistic model predicts dichotomous outcomes using logit transformation:

$$f(X) = \beta_0 + \beta_1 X,$$
$$p(Y|X = x) = \frac{\exp f(X)}{1 + \exp f(X)}.$$

In this simulation, X is always drawn from standard normal distribution and β_0 is fixed as 0. The design conditions are as follows:

- Sample size: 50, 100, 200, and 400.
- The regression coefficient: 0, 0.5, 1, and 1.5.

Do not forget to plot graphs for the relative bias in parameter estimates, the relative bias in standard errors, and statistical power for the slope in logistic regression.

```
> n <- 100
> b0 <- 0
> b1 <- 1
> x <- rnorm(n, 0, 1)
> fy <- b0 + b1*x
> p <- exp(fy)/(1 + exp(fy))
> y <- rep(NA, n)
> for(w in 1:n) y[w] <- rbinom(1, 1, p[w])
> dat <- data.frame(x = x, y = y)
> out <- glm(y ~ x, data = dat, family = binomial(link = "logit"))
> sumout <- summary(out)
> sumout[["coefficients"]][2, c(1, 2, 4)]

   Estimate  Std. Error    Pr(>|z|)
0.828806946 0.256171635 0.001214903
```

References

Jeffrey J Hoogland and Anne Boomsma. Robustness studies in covariance structure modeling an overview and a meta-analysis. *Sociological Methods & Research*, 26(3):329–367, 1998.