Background

A user on Stackoverflow recently asked a question about a program to generate Monte Carlo simulations on Bernoulli trials to calculate coverage percentages using Wald confidence intervals. One of the problems in the code is that probability value calculations are executed on individual observations rather than sums of successes and failures. R is primarily a vector processor and the code does not aggregate the individual observations to counts of successes and failures in order to calculate observed probability values for each sample.

This causes the code to generate 0 for coverage percentage for p values above 0.01 for sample sizes tested in the original post. We use code from the original post to isolate where error is introduced into the algorithm.

We set a seed, and assign values to `m`, `n`, and `p`, and attempt to generate 10,000 Bernoulli trials of size `n`.

```
set.seed(95014)
m<-10000
n<-5
p<-0.01
x <- rbinom(m,size=1,prob = p)
```

At this point `x` is a vector containing 10,000 true = 1, false = 0 values.

```
> table(x)
x
   0    1
9913   87
```

However, `x` is **NOT** 10,000 runs of samples of 5 Bernoulli trials. Given this fact, all subsequent processing by the algorithm in the original code will be incorrect.

The next line of code calculates a value for `p.hat`. This should be a single value, not a vector of 10,000 elements.

```
p.hat <- x/n
table(p.hat)

> table(p.hat)
p.hat
   0  0.2
9913   87
```

An accurate calculation for `p.hat` would be the following:

```
> p.hat <- sum(x)/length(x)
> p.hat
[1] 0.0087
```

…which is very close to the population p-value of 0.01 that we assigned earlier in the code, but still does not represent 10,000 trials of sample size 5. Instead, `p.hat` as defined above represents one Bernoulli trial with sample size 10,000.

Two minor changes to fix the code

After independently developing a Monte Carlo simulator for Bernoulli trials (see below for details), it becomes clear that with a couple of tweaks we can remediate the code from the original post to make it produce valid results.

First, we multiply `m` by `n` in the first argument to `rbinom()`, so the number of trials produced is 10,000 times sample size. We also cast the result as a matrix with 10,000 rows and `n` columns.

Second, we use `rowSums()` to sum the trials to counts of successes, and divide the resulting vector of 10,000 elements by `n`, producing correct values for `p.hat`, given sample size. Once `p.hat` is corrected, the rest of the code works as originally intended.

```
f3 <- function(n,probs) {
     res1 <- lapply(n, function(i) {
          setNames(lapply(probs, function(p) {
               m<-10000
               n<-i
               p<-p
               # make number of trials m*n, and store
               # as a matrix of 10,000 rows * n columns
               x <- matrix(rbinom(m*n,size=1,prob =
p),nrow=10000,ncol=i)
               # p.hat is simply rowSums(x) divided by n
               p.hat <- rowSums(x)/n
               lower.Wald <- p.hat - 1.96 * sqrt(p.hat*(1-p.hat)/n)
               upper.Wald <- p.hat + 1.96 * sqrt(p.hat*(1-p.hat)/n)
               p.in.CI <- (lower.Wald Starting from scratch: a basic
simulator for one p-value / sample size
```

Here we develop a solution that iteratively builds on a set of basic building blocks: one p-value, one sample size, and a 95% confidence interval. The simulator also tracks parameters so we can combine results from multiple simulations into data frames that are easy to read and interpret.

First, we create a simulator that tests 10,000 samples of size drawn from a Bernoulli distribution with a given probability value. It aggregates successes and failures, and then calculates Wald confidence intervals, and generates an output data frame. For the purposes of the simulation, the p-values we pass to the simulator represent the the "true" population probability value. We will see how frequently the simulations include the population p-value in their confidence intervals.

We set parameters to represent a true population p-value of 0.5, a sample size of 5, and z-value of 1.96 representing a 95% confidence interval. We created function arguments for these constants so we can vary them in subsequent code. We also use `set.seed()` to make the results reproducible.

```
set.seed(90125)
```

```
simulationList <- lapply(1:10000,function(x,p_value,sample_size,z_val){
     trial <- x
     successes <- sum(rbinom(sample_size,size=1,prob = p_value))
     observed_p <- successes / sample_size
     z_value <- z_val
     lower.Wald <- observed_p - z_value * sqrt(observed_p*(1-
observed_p)/sample_size)
     upper.Wald <- observed_p + z_value * sqrt(observed_p*(1-
observed_p)/sample_size)
     data.frame(trial,p_value,observed_p,z_value,lower.Wald,upper.Wald)
},0.5,5,1.96)
```

A key difference between this code and the code from the original question is that we take samples of 5 from `rbinom()` and immediately sum the number of true values to calculate the number of successes. This allows us to calculate `observed_p` as `successes / sample_size`. Now we have an empirically generated version of what was called `p.hat` in the original question.

The resulting list includes a data frame summarizing the results of each trial.

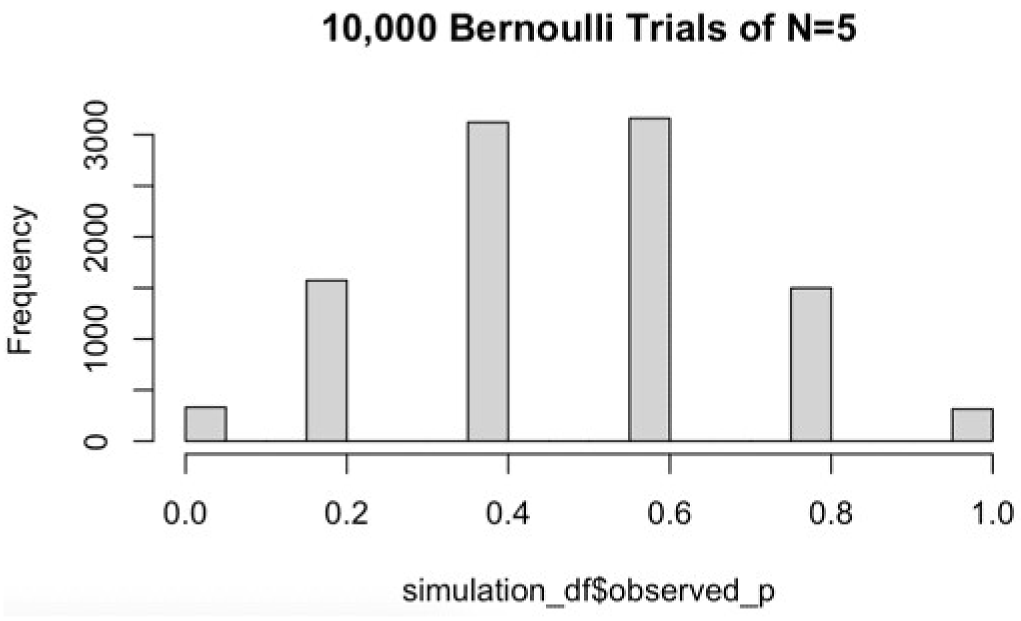
We combine the list of data frames into a single data frame with `do.call()`

```
simulation_df <- do.call(rbind,simulationList)
```

At this point `simulation_df` is a data frame containing 10,000 rows and 6 columns. Each row represents the results from one simulation of `sample_size` Bernoulli trials. We'll print the first few rows to illustrate the contents of the data frame.

```
> dim(simulation_df)
[1] 10000     6
> head(simulation_df)
  trial p_value observed_p z_value  lower.Wald upper.Wald
1     1     0.5        0.6    1.96  0.17058551  1.0294145
2     2     0.5        0.2    1.96 -0.15061546  0.5506155
3     3     0.5        0.6    1.96  0.17058551  1.0294145
4     4     0.5        0.2    1.96 -0.15061546  0.5506155
5     5     0.5        0.2    1.96 -0.15061546  0.5506155
6     6     0.5        0.4    1.96 -0.02941449  0.8294145
>
```

Notice how the `observed_p` values are distinct values in increments of 0.2. This is because when sample size is 5, the number of TRUE values in each sample can vary between 0 and 5. A histogram of `observed_p` makes this clear.



Even with a sample size of 5, we can see the shape of a binomial distribution emerging in the histogram.

Next, we calculate the coverage percentage by summing the rows where the population p-value (represented as `p_value`) is within the Wald confidence interval.

```
# calculate coverage: % of simulations where population p-value is
# within Wald confidence limits generated via simulation
sum(simulation_df$p_value > simulation_df$lower.Wald &
         simulation_df$p_value < simulation_df$upper.Wald) / 10000 *
100

 > sum(simulation_df$p_value > simulation_df$lower.Wald &
+          simulation_df$p_value < simulation_df$upper.Wald) / 10000 *
100
[1] 93.54
```

Coverage of 93.54% is a reasonable result for the simulation, given that we calculated a 95% confidence interval. We interpret the result as 93.5% of the samples generated Wald confidence intervals that included the population p-value of 0.5.

Therefore, we conclude that our simulator appears to be generating valid results. We will build on this basic design to execute simulations with multiple p-values and sample sizes.

Simulating multiple p-values for a given sample size

Next, we'll vary the probability values to see the percentage coverage for 10,000 samples of 5 observations. Since the statistics literature such as Sauro and Lewis, 2005 tells us that Wald confidence intervals have poor coverage for very low and very high p-values, we've added an argument to calculate Adjusted Wald scores. We'll set this argument to `FALSE` for the time being.

```
p_val_simulations <- lapply(c(0.01,0.1,0.4,.5,.8),function(p_val){
     aSim <- lapply(1:10000,function(x,p_value,sample_size,z_val,
adjWald){
          trial <- x
          successes <- sum(rbinom(sample_size,size=1,prob = p_value))
          if(adjWald){
               successes <- successes + 2
               sample_size <- sample_size + 4
          }
          observed_p <- sum(successes) / (sample_size)
          z_value <- z_val
          lower.Wald <- observed_p - z_value * sqrt(observed_p*(1-
observed_p)/sample_size)
          upper.Wald <- observed_p + z_value * sqrt(observed_p*(1-
observed_p)/sample_size)
          data.frame(trial,p_value,sample_size,observed_p,z_
value,adjWald,lower.Wald,upper.Wald)
     },p_val,5,1.96,FALSE)
     # bind results to 1 data frame & return
     do.call(rbind,aSim)
})
```

The resulting list, `p_val_simulations` contains one data frame for each p-value run through the simulation.

We combine these data frames and calculate coverage percentages as follows.

```
do.call(rbind,lapply(p_val_simulations,function(x){
     p_value <- min(x$p_value)
     adjWald <- as.logical(min(x$adjWald))
     sample_size <- min(x$sample_size) - (as.integer(adjWald) * 4)
     coverage_pct <- (sum(x$p_value > x$lower.Wald &
                x$p_value < x$upper.Wald) / 10000)*100
     data.frame(p_value,sample_size,adjWald,coverage_pct)

}))
```

As expected, the coverage is very poor the further we are away from a p-value of 0.5.

```
  p_value sample_size adjWald coverage_pct
1    0.01           5   FALSE         4.53
2    0.10           5   FALSE        40.23
3    0.40           5   FALSE        83.49
4    0.50           5   FALSE        94.19
5    0.80           5   FALSE        66.35
```

However, when we rerun the simulation with `adjWald = TRUE`, we get the following results.

```
  p_value sample_size adjWald coverage_pct
1    0.01           5    TRUE        95.47
2    0.10           5    TRUE        91.65
3    0.40           5    TRUE        98.95
4    0.50           5    TRUE        94.19
5    0.80           5    TRUE        94.31
```

These are much better coverage values, particularly for p-values close the the ends of the distribution.

The final task remaining is to modify the code so it executes Monte Carlo simulations at varying levels of sample size. Before proceeding further, we calculate the runtime for the code we've developed thus far.

`system.time()` tells us that the code to run 5 different Monte Carlo simulations of 10,000 Bernoulli trials with sample size of 5 takes about 38 seconds to run on a MacBook Pro 15 with a 2.5 Ghz Intel i-7 processor. Therefore, we expect that the next simulation will take multiple minutes to run.

Varying p-value and sample size

We add another level of `lapply()` to account for varying the sample size. We have also set the `adjWald` parameter to `FALSE` so we can see how the base Wald confidence interval behaves at p = 0.01 and 0.10.

```
set.seed(95014)
system.time(sample_simulations <- lapply(c(10, 15, 20, 25, 30, 50,100,
150, 200),function(s_size){
     lapply(c(0.01,0.1,0.8),function(p_val){
          aSim <- lapply(1:10000,function(x,p_value,sample_size,z_val,
adjWald){
               trial <- x
               successes <- sum(rbinom(sample_size,size=1,prob =
p_value))
               if(adjWald){
                    successes <- successes + 2
                    sample_size <- sample_size + 4
               }
               observed_p <- sum(successes) / (sample_size)
               z_value <- z_val
               lower.Wald <- observed_p - z_value * sqrt(observed_p*(1-
observed_p)/sample_size)
               upper.Wald <- observed_p + z_value * sqrt(observed_p*(1-
observed_p)/sample_size)
               data.frame(trial,p_value,sample_size,observed_p,z_
value,adjWald,lower.Wald,upper.Wald)
          },p_val,s_size,1.96,FALSE)
          # bind results to 1 data frame & return
          do.call(rbind,aSim)
     })
}))
```

Elapsed time on the MacBook Pro was 217.47 seconds, or about 3.6 minutes. Given that we ran 27 different Monte Carlo simulations, the code completed one simulation each 8.05 seconds.

The final step is to process the list of lists to create an output data frame that summarizes the analysis. We aggregate the content, combine rows into data frames, then bind the resulting list of data frames.

```
summarizedSimulations <- lapply(sample_simulations,function(y){
     do.call(rbind,lapply(y,function(x){
          p_value <- min(x$p_value)
          adjWald <- as.logical(min(x$adjWald))
          sample_size <- min(x$sample_size) - (as.integer(adjWald) * 4)
          coverage_pct <- (sum(x$p_value > x$lower.Wald &
                                   x$p_value < x$upper.Wald) /
10000)*100
          data.frame(p_value,sample_size,adjWald,coverage_pct)
     }))
})

results <- do.call(rbind,summarizedSimulations)
```

One last step, we sort the data by p-value to see how coverage improves as sample size increases.

```
results[order(results$p_value,results$sample_size),]
```

…and the output:

```
> results[order(results$p_value,results$sample_size),]
   p_value sample_size adjWald coverage_pct
1     0.01          10   FALSE         9.40
4     0.01          15   FALSE        14.31
7     0.01          20   FALSE        17.78
10    0.01          25   FALSE        21.40
13    0.01          30   FALSE        25.62
16    0.01          50   FALSE        39.65
19    0.01         100   FALSE        63.67
22    0.01         150   FALSE        77.94
25    0.01         200   FALSE        86.47
2     0.10          10   FALSE        64.25
5     0.10          15   FALSE        78.89
8     0.10          20   FALSE        87.26
11    0.10          25   FALSE        92.10
14    0.10          30   FALSE        81.34
17    0.10          50   FALSE        88.14
20    0.10         100   FALSE        93.28
23    0.10         150   FALSE        92.79
26    0.10         200   FALSE        92.69
3     0.80          10   FALSE        88.26
6     0.80          15   FALSE        81.33
9     0.80          20   FALSE        91.88
12    0.80          25   FALSE        88.38
15    0.80          30   FALSE        94.67
18    0.80          50   FALSE        93.44
21    0.80         100   FALSE        92.96
24    0.80         150   FALSE        94.48
27    0.80         200   FALSE        93.98
>
```

Interpreting the results

The Monte Carlo simulations illustrate that Wald confidence intervals provide poor coverage at a p-value of 0.01, even with a sample size of 200. Coverage improves at p-value of 0.10, where all but one of the simulations at sample sizes 25 and above exceeded 90%. Coverage is even better for the p-value of 0.80, where all but one of the sample sizes above 15 exceeded 91% coverage.

Coverage improves further when we calculate Adjusted Wald confidence intervals, especially at lower p-values.

```
results[order(results$p_value,results$sample_size),]
   p_value sample_size adjWald coverage_pct
1     0.01          10    TRUE        99.75
4     0.01          15    TRUE        98.82
7     0.01          20    TRUE        98.30
10    0.01          25    TRUE        97.72
13    0.01          30    TRUE        99.71
16    0.01          50    TRUE        98.48
19    0.01         100    TRUE        98.25
22    0.01         150    TRUE        98.05
25    0.01         200    TRUE        98.34
2     0.10          10    TRUE        93.33
5     0.10          15    TRUE        94.53
8     0.10          20    TRUE        95.61
11    0.10          25    TRUE        96.72
14    0.10          30    TRUE        96.96
17    0.10          50    TRUE        97.28
20    0.10         100    TRUE        95.06
23    0.10         150    TRUE        96.15
26    0.10         200    TRUE        95.44
3     0.80          10    TRUE        97.06
6     0.80          15    TRUE        98.10
9     0.80          20    TRUE        95.57
12    0.80          25    TRUE        94.88
15    0.80          30    TRUE        96.31
18    0.80          50    TRUE        95.05
21    0.80         100    TRUE        95.37
24    0.80         150    TRUE        94.62
27    0.80         200    TRUE        95.96
```

The Adjusted Wald confidence intervals provide consistently better coverage across the range of p-values and sample sizes, with an average coverage of 96.72% across the 27 simulations. This is consistent with the literature that indicates Adjusted Wald confidence intervals are more conservative than unadjusted Wald confidence intervals.

At this point we have a working Monte Carlo simulator that produces valid results for multiple p-values and sample sizes. We can now review the code to find opportunities to optimize its performance.

Optimizing the solution

Following the old programming aphorism of Make it work, make it right, make it fast, working the solution out in an iterative manner helped enabled me to develop a solution that produces valid results.

Understanding of how to make it right enabled me not only to see the flaw in the code posted in the question, but it also enabled me to envision a solution. That solution, using `rbinom()` once with an argument of `m * n`, casting the result as a `matrix()`, and then using `rowSums()` to calculate p-values, led me to see how I could optimize my own solution by eliminating thousands of `rbinom()` calls from each simulation.

Refactoring for performance

We create a function, `binomialSimulation()`, that generates Bernoulli trials and Wald confidence intervals with a single call to `rbinom()`, regardless of the number of trials in a single simulation. We also aggregate results so each simulation generates a data frame containing one row describing the results of the test.

```
set.seed(90125)
binomialSimulation <- function(trial_size,p_value,sample_size,z_value){
     trials <- matrix(rbinom(trial_size * sample_size,size=1,prob =
p_value),
                      nrow = trial_size,ncol = sample_size)
     observed_p <- rowSums(trials) / sample_size
     lower.Wald <- observed_p - z_value * sqrt(observed_p*(1-
observed_p)/sample_size)
     upper.Wald <- observed_p + z_value * sqrt(observed_p*(1-
observed_p)/sample_size)
     coverage_pct <- sum(p_value > lower.Wald &
                         p_value < upper.Wald) / 10000 * 100
     data.frame(sample_size,p_value,avg_observed_p=mean(
observed_p),coverage_pct)

}
```

We run the function with a population p-value of 0.5, a sample size of 5, and 10,000 trials and a confidence interval of 95%, and track the execution time with `system.time()`. The optimized function is 99.8% faster than the original implementation described earlier in the article, which runs in about 6.09 seconds.

```
system.time(binomialSimulation(10000,0.5,5,1.96))

> system.time(binomialSimulation(10000,0.5,5,1.96))
   user  system elapsed
  0.015   0.000   0.015
```

We skip the intermediate steps from the solution above, instead presenting the optimized version of the iteratively developed solution.

```
system.time(results <- do.call(rbind,lapply(c(5,10,
15,20,25,50,100,250),
                                function(aSample_size,p_values) {
     do.call(rbind,lapply(p_values,function(a,b,c,d){
             binomialSimulation(p_value = a,
                                trial_size = b,
                                sample_size = aSample_size,
                                z_value = d)
     },10000,5,1.96))
},c(0.1,0.4,0.8))))
```

As expected, elimination of the thousands of unnecessary calls to `rbinom()` radically improves performance of the solution.

```
   user  system elapsed
  0.777   0.053   0.830
```

Given that our prior solution ran in 217 seconds, performance of the optimized version is really impressive. Now we have a solution that not only generates accurate Monte Carlo simulations of Bernoulli trials, but it’s also very fast.