Answering Theoretical Questions with R

James H. Steiger

Department of Psychology and Human Development
Vanderbilt University

Using R to Answer Theoretical Questions

Two Uses for R

- Today we'll introduce the R statistical program that is a primary tool both for statistical *analysis* and for statistical *investigation* in Psychology 310.
- By that, I mean that we will use R to perform statistical analyses, but we will also use it to investigate statistical theory.
- We can use R to perform a huge variety of analyses, because many procedures have been programmed in R, and are available in libraries.
- We can use R to investigate theoretical questions and follow statistical hunches, because R is extremely powerful *and* easily extensible. In a few lines of code, we can get R to create data, analyze it, perform the operation thousands of times, store all the results, and tell us what happened.
- So in many situations where we ask, "What would happen if. . . " we can get an answer from R.

Two Uses for R

- Today we'll introduce the R statistical program that is a primary tool both for statistical *analysis* and for statistical *investigation* in Psychology 310.
- By that, I mean that we will use R to perform statistical analyses, but we will also use it to investigate statistical theory.
- We can use R to perform a huge variety of analyses, because many procedures have been programmed in R, and are available in libraries.
- We can use R to investigate theoretical questions and follow statistical hunches, because R is extremely powerful *and* easily extensible. In a few lines of code, we can get R to create data, analyze it, perform the operation thousands of times, store all the results, and tell us what happened.
- So in many situations where we ask, "What would happen if. . . " we can get an answer from R.

Two Uses for R

- Today we'll introduce the R statistical program that is a primary tool both for statistical *analysis* and for statistical *investigation* in Psychology 310.
- By that, I mean that we will use R to perform statistical analyses, but we will also use it to investigate statistical theory.
- We can use R to perform a huge variety of analyses, because many procedures have been programmed in R, and are available in libraries.
- We can use R to investigate theoretical questions and follow statistical hunches, because R is extremely powerful *and* easily extensible. In a few lines of code, we can get R to create data, analyze it, perform the operation thousands of times, store all the results, and tell us what happened.
- So in many situations where we ask, "What would happen if. . . " we can get an answer from R.

Two Uses for R

- Today we'll introduce the R statistical program that is a primary tool both for statistical *analysis* and for statistical *investigation* in Psychology 310.
- By that, I mean that we will use R to perform statistical analyses, but we will also use it to investigate statistical theory.
- We can use R to perform a huge variety of analyses, because many procedures have been programmed in R, and are available in libraries.
- We can use R to investigate theoretical questions and follow statistical hunches, because R is extremely powerful *and* easily extensible. In a few lines of code, we can get R to create data, analyze it, perform the operation thousands of times, store all the results, and tell us what happened.
- So in many situations where we ask, "What would happen if. . . " we can get an answer from R.

Two Uses for R

- Today we'll introduce the R statistical program that is a primary tool both for statistical *analysis* and for statistical *investigation* in Psychology 310.
- By that, I mean that we will use R to perform statistical analyses, but we will also use it to investigate statistical theory.
- We can use R to perform a huge variety of analyses, because many procedures have been programmed in R, and are available in libraries.
- We can use R to investigate theoretical questions and follow statistical hunches, because R is extremely powerful *and* easily extensible. In a few lines of code, we can get R to create data, analyze it, perform the operation thousands of times, store all the results, and tell us what happened.
- So in many situations where we ask, "What would happen if. . . " we can get an answer from R.

Interpreted vs. Compiled Languages

- R is an *interpreted* language.
- Each syntactically valid line of code is executed immediately, and the results are added to an active intellectual structure called a *workspace*.
- Compiled languages are different. With compiled languages, you create an entire program, then submit the program to a compiler. The compiler takes your higher level language and translates it into ultra-fast *machine code*.
- The compiler only needs to read your high-level code once. From then on, your program runs as ultra-fast machine code.
- An interpreter reads and interprets each line of code every time it executes. In general, it will run an order of magnitude slower than compiler code.
- On the other hand, you get your results faster when the operation is simple.

Interpreted vs. Compiled Languages

- R is an *interpreted* language.
- Each syntactically valid line of code is executed immediately, and the results are added to an active intellectual structure called a *workspace*.
- Compiled languages are different. With compiled languages, you create an entire program, then submit the program to a compiler. The compiler takes your higher level language and translates it into ultra-fast *machine code*.
- The compiler only needs to read your high-level code once. From then on, your program runs as ultra-fast machine code.
- An interpreter reads and interprets each line of code every time it executes. In general, it will run an order of magnitude slower than compiler code.
- On the other hand, you get your results faster when the operation is simple.

Interpreted vs. Compiled Languages

- R is an *interpreted* language.
- Each syntactically valid line of code is executed immediately, and the results are added to an active intellectual structure called a *workspace*.
- Compiled languages are different. With compiled languages, you create an entire program, then submit the program to a compiler. The compiler takes your higher level language and translates it into ultra-fast *machine code*.
- The compiler only needs to read your high-level code once. From then on, your program runs as ultra-fast machine code.
- An interpreter reads and interprets each line of code every time it executes. In general, it will run an order of magnitude slower than compiler code.
- On the other hand, you get your results faster when the operation is simple.

Interpreted vs. Compiled Languages

- R is an *interpreted* language.
- Each syntactically valid line of code is executed immediately, and the results are added to an active intellectual structure called a *workspace*.
- Compiled languages are different. With compiled languages, you create an entire program, then submit the program to a compiler. The compiler takes your higher level language and translates it into ultra-fast *machine code*.
- The compiler only needs to read your high-level code once. From then on, your program runs as ultra-fast machine code.
- An interpreter reads and interprets each line of code every time it executes. In general, it will run an order of magnitude slower than compiler code.
- On the other hand, you get your results faster when the operation is simple.

Interpreted vs. Compiled Languages

- R is an *interpreted* language.
- Each syntactically valid line of code is executed immediately, and the results are added to an active intellectual structure called a *workspace*.
- Compiled languages are different. With compiled languages, you create an entire program, then submit the program to a compiler. The compiler takes your higher level language and translates it into ultra-fast *machine code*.
- The compiler only needs to read your high-level code once. From then on, your program runs as ultra-fast machine code.
- An interpreter reads and interprets each line of code every time it executes. In general, it will run an order of magnitude slower than compiler code.
- On the other hand, you get your results faster when the operation is simple.

Interpreted vs. Compiled Languages

- R is an *interpreted* language.
- Each syntactically valid line of code is executed immediately, and the results are added to an active intellectual structure called a *workspace*.
- Compiled languages are different. With compiled languages, you create an entire program, then submit the program to a compiler. The compiler takes your higher level language and translates it into ultra-fast *machine code*.
- The compiler only needs to read your high-level code once. From then on, your program runs as ultra-fast machine code.
- An interpreter reads and interprets each line of code every time it executes. In general, it will run an order of magnitude slower than compiler code.
- On the other hand, you get your results faster when the operation is simple.

Simple Calculations in R

Exponentiation and Square Roots

- After you get used to R, you'll start using it to do simple calculations.
- Try the following:

```
> 2 + 2
[1] 4
> 2 * 12
[1] 24
> (3+5)/(2+2)
[1] 2
```

Simple Calculations in R

Exponentiation and Square Roots

- Try the following:

```
> ## Exponentiation
>
> 3^3
[1] 27
> ## Square root
>
> sqrt(64)
[1] 8
>
```

Defining a Variable in R

- As our first simple example, suppose we enter the command:

```
> x <- 5
```

- The symbol `x` has now been assigned the value 5. If we simply type the name of the symbol and press the Enter key, R will respond with the current value of `x`.

```
> x
[1] 5
```

Error Messages in R

- When you make an error in R, it will often signal you with an error message.
- Often these messages are cryptic to the beginner. Learning to interpret them and figure out what you really did wrong is a skill that takes some time to acquire.

Error Messages in R

- When you make an error in R, it will often signal you with an error message.
- Often these messages are cryptic to the beginner. Learning to interpret them and figure out what you really did wrong is a skill that takes some time to acquire.

Error Messages in R

- For example, one thing that will cause you to make errors when you first start using R is that it is *case sensitive*. It distinguishes between upper and lower case letters.
- So the variable `x` is not the same as the variable `X`.
- So, if we make a mistake and, in our current session, ask the value of `X` which has not yet been defined, R will inform us.

```
> X

Error: object 'X' not found
```

- The object `x` has been defined, but `X` has not.

Defining a Vector of Numbers

- We can define an entire list with a simple syntax. Here, we define `x` to be the list 70,80,90.

```
> x <- c(70,80,90)
```

- We can access the entire list

```
> x
```

```
[1] 70 80 90
```

- We can access a single element of the list

```
> x[2]
```

```
[1] 80
```

- Indeed, we can specify a range of elements.

```
> x[2:3]
```

```
[1] 80 90
```

Creating a Sequence of Numbers

- Suppose we wanted to know the sum of all the numbers from 1 to 100.
- It is a snap in R.

```
> 1:100

  [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
 [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
 [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
 [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
 [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
 [91]  91  92  93  94  95  96  97  98  99 100

> sum(1:100)

[1] 5050
```

Simple Listwise Operations

- Calculating the mean is as simple as this:

```
> x <- c(70,80,90)
> mean(x)
[1] 80
```

- Likewise, the standard deviation is

```
> sd(x)
[1] 10
```

Simple Listwise Operations

- Listwise operations are simple, and natural, in the R syntax. If we wish to transform x into y with the formula $y = 2x + 5$, we simply write

```
> y <- 2*x + 5
```

- Notice that *multiplication has to be indicated explicitly* with the * operator. A common error is to forget the * character in front of a parenthetical expression and write something like

```
> 2(x+5)
```

in which case you will get a cryptic error message like this

```
Error: attempt to apply non-function
```

This message occurs because when R sees a an expression in front of parentheses, it interprets it as a *function call*, and checks its list of defined functions for a function called 2! Of course, there is no such function, and so R denounces you for attempting to apply a non-function with functional notation!

Using Functions within Expressions

- One of the beauties of R is that you can employ functions within expressions in a natural way.
- For example, we can convert the `x` scores to z-score form with the formula

```
> (x - mean(x))/sd(x)
[1] -1  0  1
```

- Not surprisingly, we find that the `y` scores have the same z-score equivalents as their `x` counterparts.

```
> (y - mean(y))/sd(y)
[1] -1  0  1
```

A Simple Z-Score Function

- Something that we do as often as computing z-scores can be automated by defining a function.
- Here is code that will do the trick.

```
> z.score <- function(x){(x - mean(x))/sd(x)}
```

- Once we have defined the function, using it is very simple.

```
> z.score(x)
[1] -1  0  1
> z.score(y)
[1] -1  0  1
```

Parsing the Z-Score Function

- The first part of the command, "`z.score <- function`" declares that the name `z.score` is to be assigned a function.
- Once you assign a function, you have to describe the input required by the function. In this case, we list a single input `x` within the parentheses.
- It is important to realize that, inside the function, `x` is a generic descriptor. It is not the same as the variable `x` that has already been described as a list of numbers.
- Next, inside braces, you write code to operate on the input variable `x` to produce the desired result.
- The last output produced by the commands inside braces is returned to R by the function anytime it is executed.
- Many programmers consider it good programming style to have explicit return statements, so that it is easy to identify points at which a function returns values.
- R has a `return` command. We could use it in a modified version of the `z.score` function as follows:

```
> z.score <- function(x){
+  return((x-mean(x))/sd(x))
+ }
```

A Rescaling Function

- We can use the Z-scoring function we just created in future functions.
- For example, here is a simple function to rescale an input list of numbers to have desired mean and standard deviation.
- The function first Z-scores the numbers, then multiplies by the desired standard deviation and adds the desired mean.

```
> rescale.numbers<- function(x,mean,sd){
+ z <- z.score(x)
+ return(z*sd + mean)
+ }
> rescale.numbers(x,40,14)
[1] 26 40 54
```

Generating a Random Sample

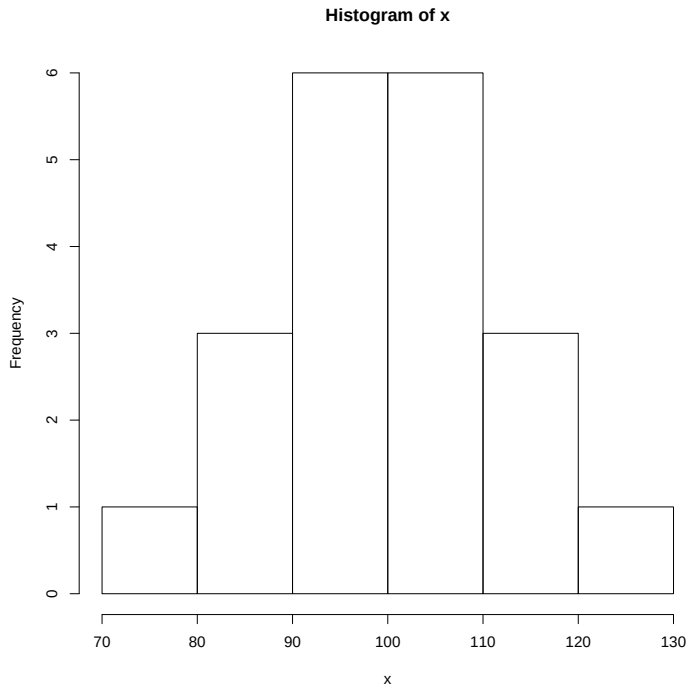
- R has built-in distribution functions.
- One of the standard capabilities is random number generation.
- R simulates random numbers. Actually, the system is completely deterministic. To start the sequence from a known point, use the `set.seed` function. This will allow you to replicate any investigation that uses random numbers.
- Here is the code to generate a simulated random sample of size $n = 20$ from a normal distribution with a mean of 100 and a standard deviation of 15.

```
> set.seed(12345)
> x <- rnorm(20,100,15)
> mean(x)
[1] 101.1478
> sd(x)
[1] 12.50903
```

Statistical Graphics

- R has many sophisticated graphics capabilities.
- Let's draw a histogram:

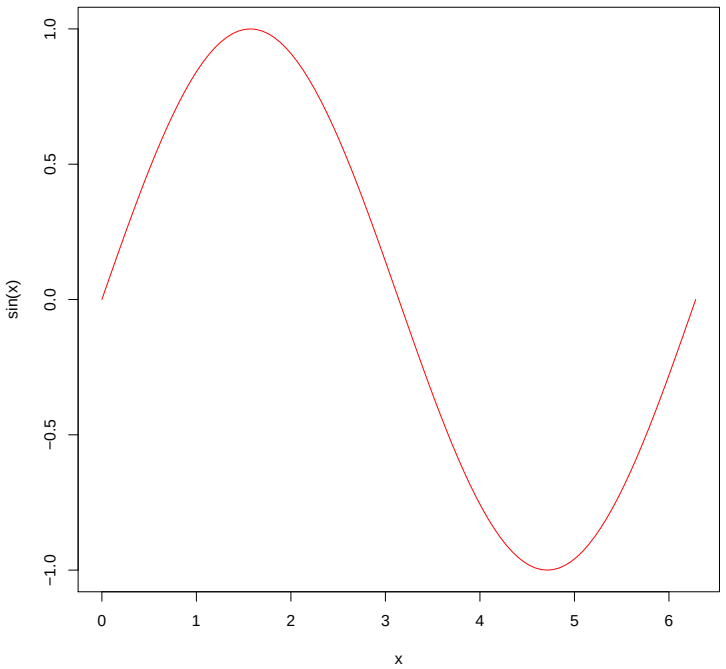
```
> hist(x)
```



Curve Drawing

- R can plot curves for you. Here is a graph of $\sin(x)$ in red.

```
> curve(sin(x),0,2*pi,col="red")
```



Generating Data with Exact Properties

- Notice that the data on the previous slide did not have a sample mean of 100 or a sample standard deviation of 15. This is because the data were sampled from a population with that metric. Samples almost never have the same mean and standard deviation as the population from which they were taken, due to random variation.
- Here are some "random-looking" data that have a mean of exactly 100 and a standard deviation of exactly 15. Note how I create a function that takes a random sample, then scales it exactly to the desired metric.

```
> make.exact.data <- function(n, mean, sd){
+ x <- rnorm(n)
+ return(rescale.numbers(x,mean,sd))
+ }
> x <- make.exact.data(10,100,15)
> x
 [1] 108.92832 117.59925  90.66797  79.01366  78.44208 122.07874  92.75415
 [8] 106.88624 106.78037  96.84923
> mean(x)
[1] 100
> sd(x)
[1] 15
```

The Maximum Possible Z-Score

- Here is a theoretical question with an answer that might strike you as surprising.
- Suppose you take a course with 4 other students, so that the class size is $n = 5$.
- What is the maximum Z-score you can receive in the course?

The Maximum Possible Z-Score

- Here is a theoretical question with an answer that might strike you as surprising.
- Suppose you take a course with 4 other students, so that the class size is $n = 5$.
- What is the maximum Z-score you can receive in the course?

The Maximum Possible Z-Score

- Here is a theoretical question with an answer that might strike you as surprising.
- Suppose you take a course with 4 other students, so that the class size is $n = 5$.
- What is the maximum Z-score you can receive in the course?

The Maximum Possible Z-Score

- At first, the answer might seem to be infinity, depending on how extremely you outperform the other students.
- However, a bit of reflection might convince you that this is not necessarily true.
- When scores are standardized, no matter how far out one score is, it must be "brought back into the field of view" in order to standardize the variability of the scores. So, ironically, as your "raw score" increases without bound, your Z-score may not increase.

The Maximum Possible Z-Score

- Here are some data followed by their $Z-$scores.

```
> z.score(c(1,2,3,4,100))
[1] -0.4814564 -0.4585299 -0.4356034 -0.4126769  1.7882666
> z.score(c(1,2,3,4,100000))
[1] -0.4472471 -0.4472248 -0.4472024 -0.4471801  1.7888544
```

- You can see that, as the raw score of the best performer increases without bound, the Z-score for the best performer stabilizes at 1.7888.
- With some work, you can prove that the maximum Z-score is $\frac{n-1}{\sqrt{n}}$. Here is another example with $n = 10$.

```
> z.score(c(1,2,3,4,5,6,7,8,9,10000000))
 [1] -0.3162290 -0.3162287 -0.3162284 -0.3162281 -0.3162278 -0.3162274
 [7] -0.3162271 -0.3162268 -0.3162265  2.8460499
> 9/sqrt(10)
[1] 2.84605
```

Expected Value of the Maximum

- An important branch of statistics is "Order Statistic Theory," which deals with the distribution of quantities after they are observed and ranked.
- For example, in a sample of size n, the nth order statistic is the value of the largest number in the sample.
- Over repeated samples, the nth order statistic will have a distribution. In general, this distribution will not have the same location, spread, or shape as the distribution of the population from which the samples are taken.
- We can use R to investigate these points.

Distribution of the Maximum

- Not only can R generate random samples, it can effortlessly replicate the process while storing the results.
- Below is a statement that, in one line, takes a sample of 100 observations from a normal distribution with a mean of 70 and a standard deviation of 2.5, calculates the maximum (the n-th order statistic), and repeats the process 100 times.

```
> replicate(100,max(rnorm(100,70,2.5)))
  [1] 76.19278 76.63947 76.86851 75.64943 78.32683 75.31866 75.79306 77.73424
  [9] 75.81059 74.71965 76.14269 75.15458 75.73470 78.22256 77.62422 75.86363
 [17] 76.75039 76.90250 74.88691 76.57215 74.56204 75.55808 76.06466 76.49287
 [25] 75.91715 76.89264 75.49782 75.25300 76.75532 75.82443 75.48602 75.41367
 [33] 75.98224 76.23405 75.19356 76.48001 76.07191 76.05485 75.19499 75.50697
 [41] 74.77912 75.92430 78.16012 75.61860 76.51255 77.99303 77.10926 76.74191
 [49] 77.11422 75.36821 75.69930 74.46599 74.04653 76.51092 75.84704 74.57591
 [57] 76.01357 75.29074 76.89521 76.66205 75.42587 75.60124 76.33356 76.60389
 [65] 76.90107 75.88839 74.91978 75.50928 77.64556 76.31904 75.98597 76.54471
 [73] 78.39023 75.04108 76.31743 75.21772 76.70969 75.92105 75.61413 77.49126
 [81] 76.83390 76.70824 76.07758 76.32378 76.80724 75.25670 76.99410 75.99128
 [89] 76.91233 75.96028 77.29332 77.05173 76.42490 75.94495 76.65877 76.86872
 [97] 75.18656 76.02243 76.90028 75.56453
```

- R took 100 samples of size 100, got the maximum value in each, and returned the result.
- Imagine these are high schools, and the height of male students in the population is 70, the standard deviation 2.5.
- These results represent the height of the tallest boys in each class over 100 years. Note that they show some variation!

Distribution of the Maximum

- The "Expected Value" of a random variable is the long run average.
- As the size of the sample increases, you get an increasingly accurate indication of the expected value by just calculating the mean of the sample.
- So to get a highly refined estimate of the expected value of the tallest boy in a class of 100, I let the number of simulated "years" get very large.

```
> set.seed(12345)
> mean(replicate(10000,max(rnorm(100,70,2.5))))
[1] 76.282
```

- R took 10000 samples of size 100, got the maximum value in each, and calculated the mean of the result.