Introduction to R: Basic Data Structure and the Truncated Normal

Tom Barber

R is a very useful statistical programming language. It is so useful, in fact, that it is required across a broad range of fields. This document expands on R basics and introduces probability distributions. Namely, we will be exploring normal distributions and their truncated versions. Finally, we will look at class types and syntax. Code and explanations are written in a problem and answer format which I have written myself.

{r}{r setup, include=FALSE} knitr::opts_chunk$set(echo = TRUE, eval = FALSE)

Part I

1. The normal distribution:

Explore the normal distribution in R by reading about related functions by `?rnorm`.

```
?rnorm
```

```
## starting httpd help server ... done
```

Generate $n = 100$ random draws from the standard normal distribution and assign those to the variable `Z`.

```
z <- rnorm(100, mean = 0, sd = 1)
```

What are the type and dimensions of the data that the function takes as input here?

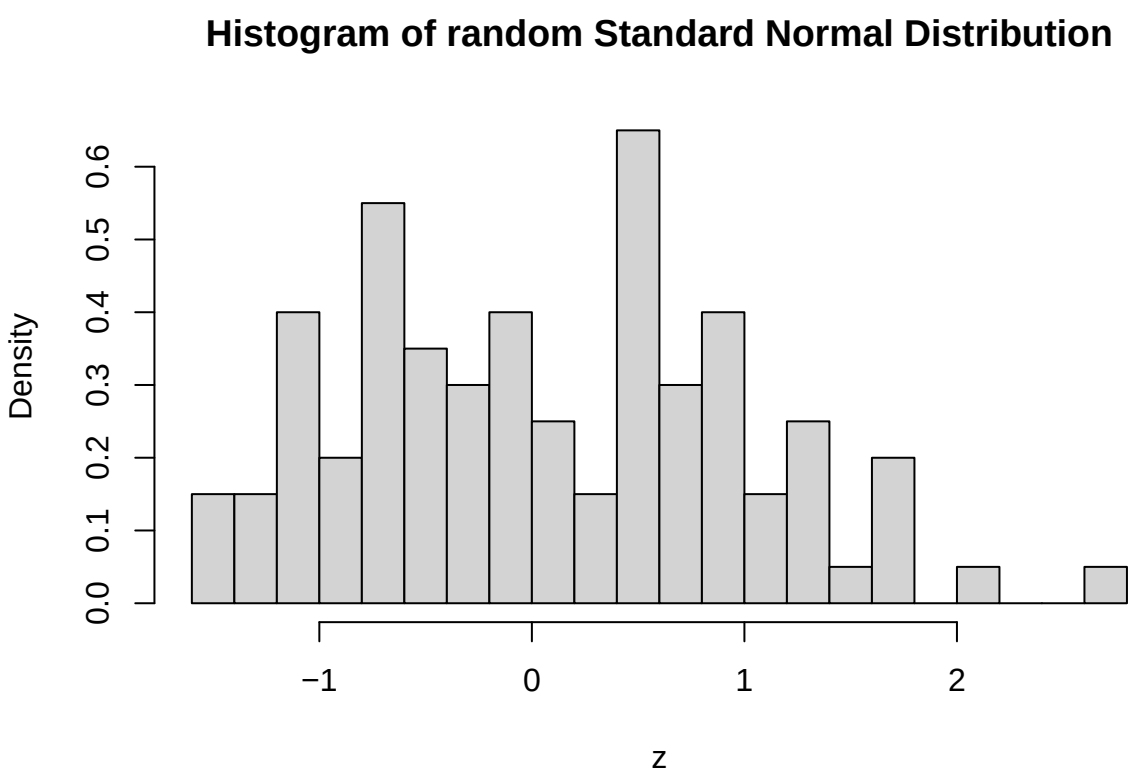
The function takes inputs that are dimension 1x1 and the type of the inputs are vectors for the mean and standard deviation, and numeric for n.

What are the type and dimensions of the data that the function returns?

The dimension of the data returned is 1x100 and the type of the data returned is numeric

Use `hist()` to plot a histogram of `Z`, normalized to be a density (so that the total area of the bars is 1, check `?hist`), make sure that the number of `breaks` (a parameter in the `hist()` function) does justice to your data, as the default value may not be representative of the underlying data. Justify your choice (you can either provide several figures, prior knowledge or any other convincing argument).

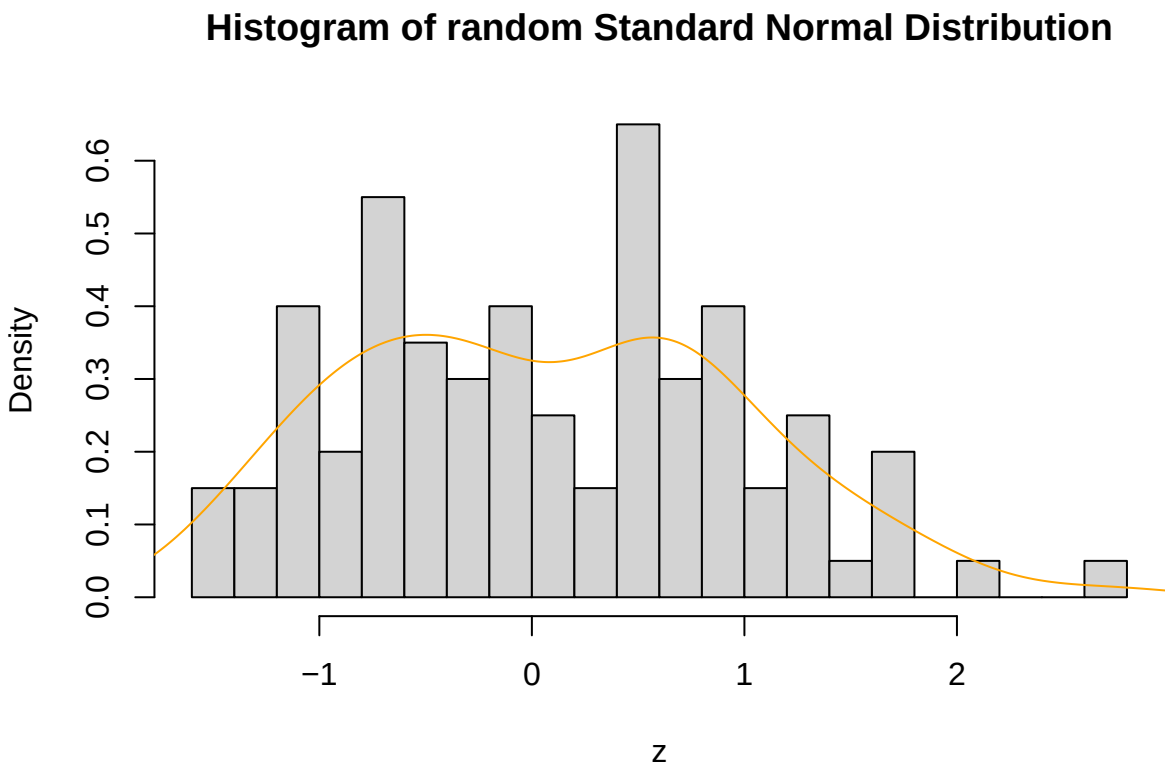
```
?hist
hist(z, breaks = 16,freq = FALSE,main = paste("Histogram of random Standard Normal Distribution"))

hist(z, breaks = 20,freq = FALSE,main = paste("Histogram of random Standard Normal Distribution"))
```



When deciding on breaks I found that 16 seems to be the max for the data. This is what allows for the histogram to best represent a normal curve. We see in the second example a breakdown in the data which makes it less representative of the distribution.

Add a line to the histogram that shows the theoretical normal density:

```
hist(z, breaks = 16,freq = FALSE,main = paste("Histogram of random Standard Normal Distribution"))
lines(density(z), col = 'orange')
```


Histogram of random Standard Normal Distribution
Density
0.0
0.1
0.2
0.3
0.4
0.5
0.6
−1
0
1
2
z

2. Sampling from the truncated normal distribution

The truncated normal distribution is the probability distribution derived from that of a normally distributed random variable by bounding the random variable from either below or above. In this exercise we will generate samples from a truncated normal random variable in a very simple way.

First, generate `n = 10000` samples from a standard normal random variable, assign those to, say, a variable named `Z`.

```
Z <- rnorm(10000, mean = 0, sd = 1)
```

Then, using a single command and line of code, assign to a new variable, say, `Z_trunc`, all the values of `Z` that are greater than or equal to 0.5.

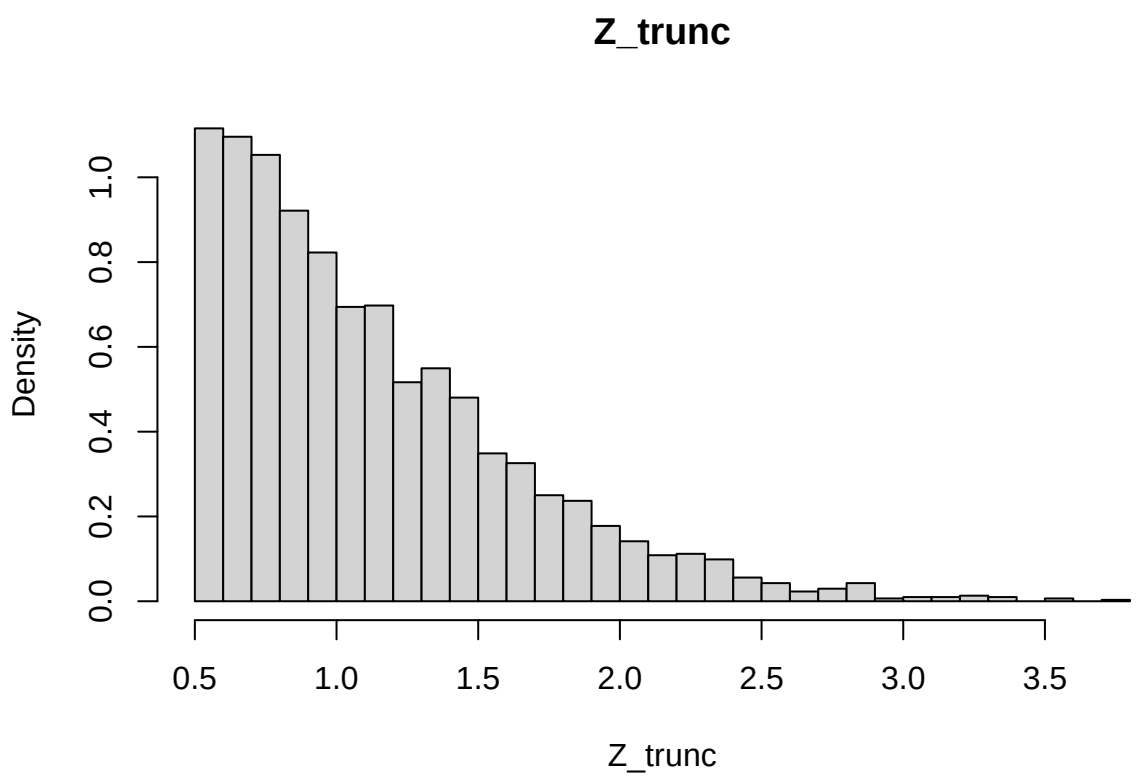
```
Z_trunc <- Z[Z >= 0.5]
```

First we create a variable Z_trunc and use the assignment command. We then index Z and set the stipulation that we are indexing Z when greater than 0.5. This then becomes Z_trunc. The input types are numeric with dimension 1x10000 and the output is also numeric with dimension 1x3077.

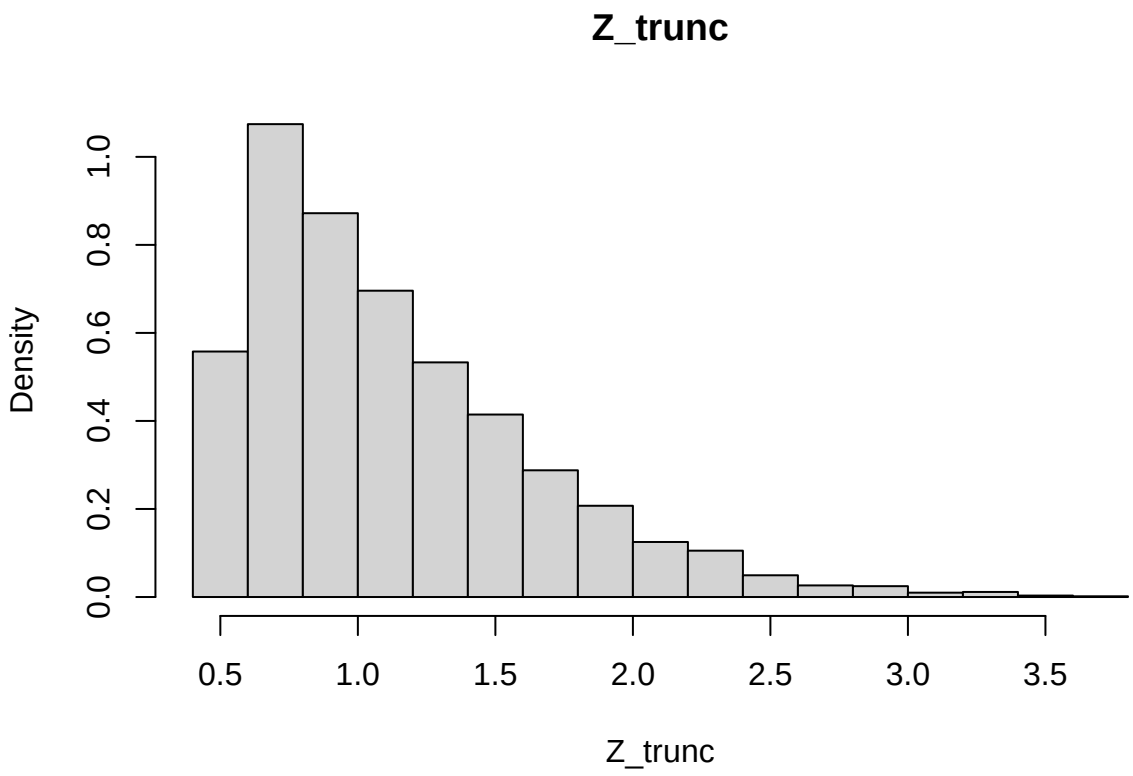
We have now generated a sample of truncated normal values.

Use `hist()` to plot a histogram of `Z_trunc`, normalized to be a density, make sure that the number of `breaks` does justice to your data. Justify your choice.

```
hist(Z_trunc, breaks = 40, freq = FALSE, main = "Z_trunc")
```



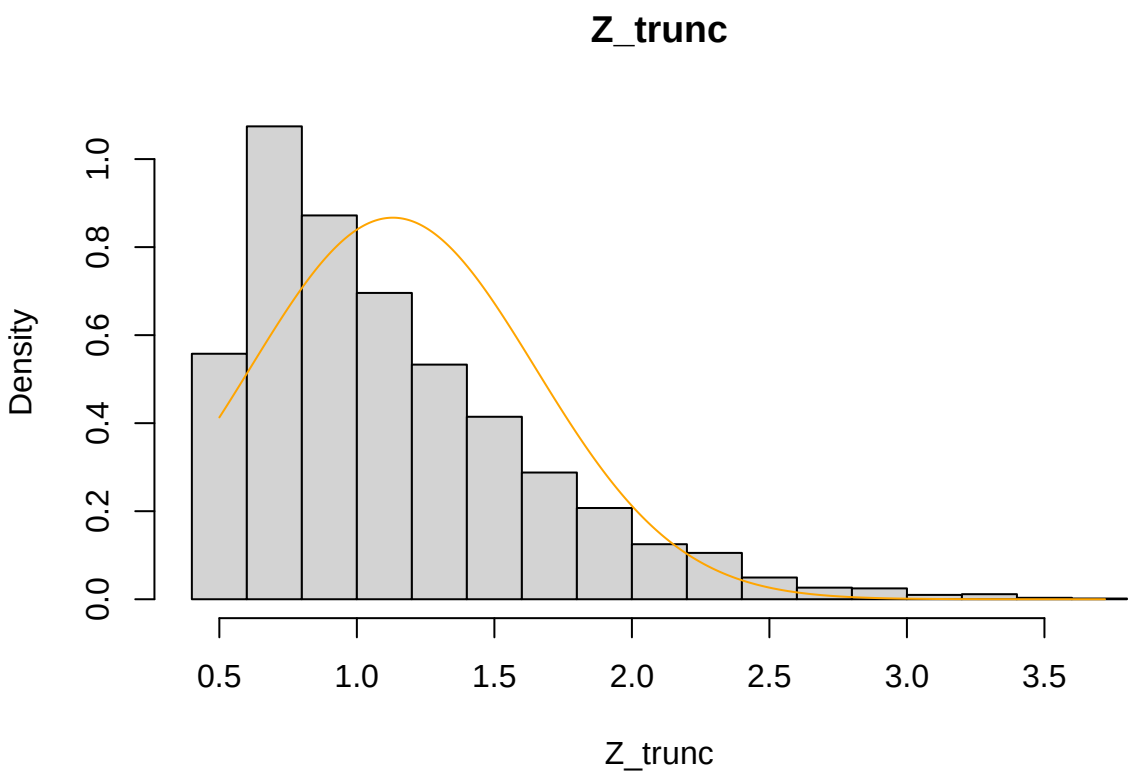
```r
hist(Z_trunc, breaks = 20, freq = FALSE, main = "Z_trunc")
```



The number of breaks used for this histogram is 20. We see that >20 breaks does not do justice to the data, whereas with 20 we can clearly see the values declining like we would expect.

Let us compare the results to the theoretical density. Choose a range for which you want to compute the density, and assign it to a vector. Using only basic operators and functions for the standard normal family (ie `dnorm`, `pnorm`, `qnorm`, `rnorm`) and the formula for the density of the truncated normal random variable from Wikipedia to compute the density of the truncated normal and add a line of your favorite color to the histogram that depicts it. That formula takes as input the range vector or other constants (note, we are using `b = Inf` here, what are the other constants?). For legibility, you must separate the computations to, say, `numerator` and `denominator`. Carefully describe each part of your computation: which functions operate on which data types? Is recycling used? Is the computation element by element?

```r
range_x <- seq(min(Z_trunc), max(Z_trunc), length.out = 10000)
num <- dnorm(range_x, mean = mean(Z_trunc), sd = sd(Z_trunc))
denom <- pnorm(max(range_x), mean = mean(Z_trunc), sd = sd(Z_trunc)) - pnorm(min(range_x),mean = mean(Z_
final_density <- num / denom
hist(Z_trunc, breaks = 20, freq = FALSE, main = "Z_trunc")
lines(range_x, final_density, col = 'orange')
```



The functions used are seq, dnorm, pnorm, division, hist, and lines. The data types they operate on are numeric vectors. Recycling is used in the division to get the final density, which computationally is element wise division.

Using the same careful descriptions as in the above, compute the expected value and standard deviation for this truncated normal distribution (based on the theoretical formulas) as well as the empirical average and empirical standard deviation of the sample. As counter-intuitively as it may sound, if you do this correctly, something should go wrong for the theoretical standard deviation. What went wrong? Fix it in a new chunk of code! Are your revised value and the empirical value similar?

```
Theoretical_stand_dev <- sqrt(((num)^2) / (denom - 1))
```

```
## Warning in sqrt(((num)^2)/(denom - 1)): NaNs produced
```

```
Theoretical_ex <- mean(Z_trunc) + ((dnorm(min(Z_trunc), mean = mean(Z_trunc), sd = sd(Z_trunc)) - dnorm

emp_mean <- mean(Z_trunc)
emp_sd <- sd(Z_trunc)
print("Theoretical Expected Value")
```

```
## [1] "Theoretical Expected Value"
```

```
print(Theoretical_ex)
```

```
## [1] 0.717826
```

```r
print("Empirical Mean")
```

```
## [1] "Empirical Mean"
```

```r
print(emp_mean)
```

```
## [1] 1.130854
```

```r
print("Empirical Standard Deviation")
```

```
## [1] "Empirical Standard Deviation"
```

```r
print(emp_sd)
```

```
## [1] 0.5180782
```

The issue with the theoretical standard deviation is that NaNs are produced due to the fact that we are taking the square root of a negative number. What I did to fix that below was by taking the max and min of Z_trunc and used a very similar method to finding the final density in the previous problem but adapted to the standard deviation formula from the Wikipedia.

```r
sd_theo <- sqrt(sd(Z_trunc)^2 - ((dnorm(min(Z_trunc), mean = mean(Z_trunc), sd = sd(Z_trunc)) - dnorm(ma

print("Theoretical Standard Deviation")
```

```
## [1] "Theoretical Standard Deviation"
```

```r
print(sd_theo)
```

```
## [1] 0.4242936
```

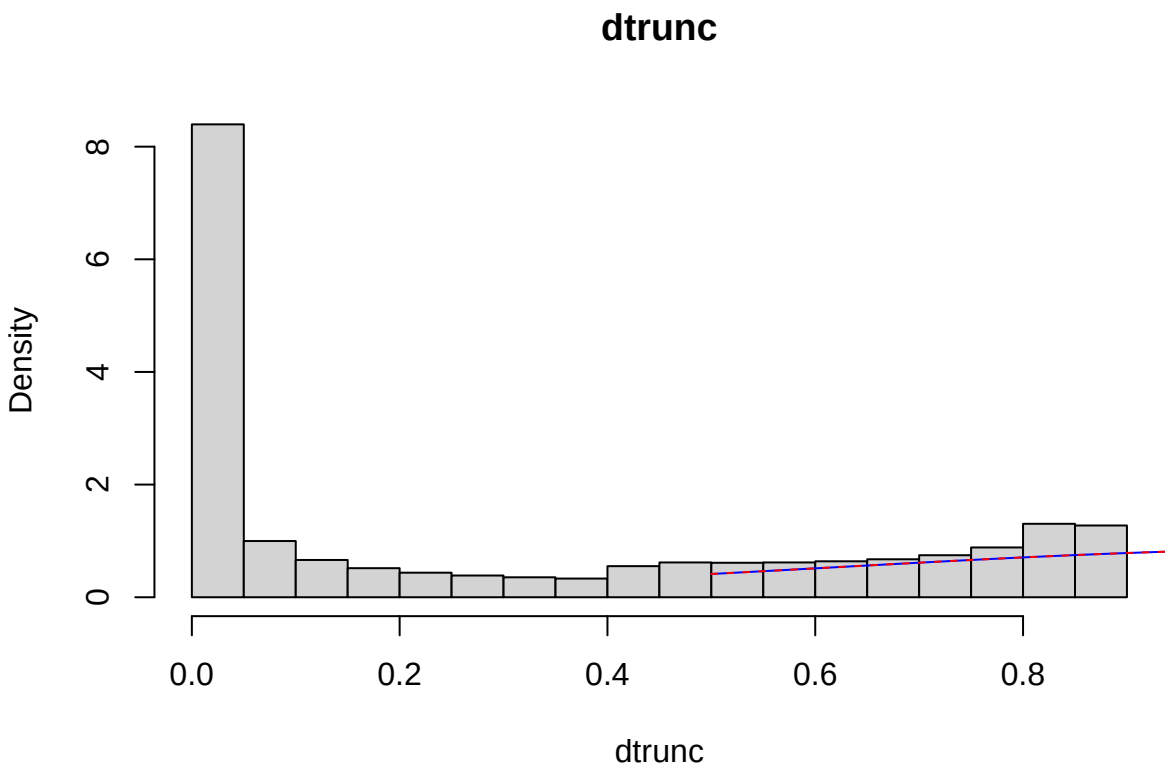
Using this correction I found that the standard deviations and expected values between the theoretical and empirical are fairly similar.

Let us compare the results to the `truncnorm` package. First, install the package and load it. Then, apply `dtruncnorm` (with the correct parameters) to the same range you used before. In this chunk of code, plot your histogram, your original curve, and another, dashed line to the histogram of your least favorite color and check whether or not the lines are similar.

```r
library(truncnorm)
```

```
## Warning: package 'truncnorm' was built under R version 4.2.3
```

```r
dtrunc <- dtruncnorm(range_x, min(Z_trunc), max(Z_trunc), mean = mean(Z_trunc), sd = sd(Z_trunc))
hist(dtrunc, breaks = 20, freq = FALSE, main = "dtrunc")
lines(range_x, final_density, col = 'blue')
lines(range_x, dtrunc, col = 'red', lty = 'dashed')
```



Can you foresee any problems with this sampling method? Consider how different this is than `rnorm` and list 3 main weaknesses.

When looking at rnorm it is a much simpler way to sample a random normal distribution. Since truncating is a bit more complex there are a few weaknesses. I would say the first is that truncating is more inefficient when the end points are farther from the mean, and so we risk truncating many samples. There is also potential bias when truncating based on which samples are within or outside the bounds. The last weakness is that we also lose the end behavior either to the right or left of the truncation, meaning we lose an end point. Such a loss can have an impact on the statistical inferences we can make.

Part II

1. Syntax and class-typing.

For each of the following commands, either explain why they should be errors, or explain the non-erroneous result.

```
vector1 <- c("5", "12", "7", "32")
max(vector1)
sort(vector1)
sum(vector1)
```

This has to do with the type of data provided. The data in vector1 are character types and so the functions treat them differently than numerics. The reason max(vector1) returns "7" because it comes last in alphabetical order. After looking at the ?sort() I found that for character vectors the elements are sorted

by ASCII or Latin-1 encodings, meaning standard numeric values are assigned to character data and then sorted. This is why we get an odd sort answer. For the sum function we get an error because the function can only take numeric, complex, or logical vectors.

For the next series of commands, either explain their results, or why they should produce errors.

```
vector2 <- c("5",7,12)
vector2[2] + vector2[3]

list4 <- list(z1="6", z2=42, z3="49", z4=126)
list4[[2]]+list4[[4]]
list4[2]+list4[4]
```

After playing around with the global environment I found that having one of the vector inputs be "5" converts the entire vector to character data types. If you were to just have 5 then it would be a numeric vector. As such, since we are trying to add two character data types with a binary operator we get an error because what we are trying to add is non numeric. When it comes to the list we created we find that the first addition yields an answer of 168. This is because the data type returned by list4[[2]] and list4[[4]] are doubles since the double brackets allow us to access the actual element in the list, which allow for addition. For the second addition command we are indexing the list itself, and so we return a data type of list that does not allow for addition to be executed. Thus, we get an error of non numeric argument to the binary operator.

For the next series of commands, either explain their results, or why they should produce errors.

```
factory <- matrix(c(40, 1, 60, 3), nrow = 2)
apply(factory, 1, function(x) {mean(x)})
apply(factory, 2, mean)
```

The first line of code creates a 2x2 matrix with rows 40, 60 and second row 1, 3. The first apply function takes the input 1, corresponding to rows, and then calculates the mean of each row. This line of code also creates a function that takes x and then calculates the mean, essentially doing the same as the third line of code with the mean input. The difference with the third line, besides what was just discussed, is the input 2. This changes the mean calculation from row wise to column wise.

For the next series of commands, either explain their results, or why they should produce errors.

```
my.distribution <- list("exponential", 7, FALSE)
is.character(my.distribution[1])
is.character(my.distribution[[1]])
my.distribution[2] ^ 2
my.distribution[[2]] ^ 2
```

After creating the list we see that the second line of code checking if the first element is a character data type returns false. This is because indexing the list like this just returns a sublist with the first element, which also has a data type of list. The double brackets actually allow us to access the element in the list, and so we see the third line of code return a true value because "exponential" is a character data type. By similar arguments in lines four and five we first get an error for trying to square a list, and we cannot perform arthimetic on this data type. The last line returns 49 as expected because we are performing the square on the actual element in the list, which is numeric.