R Markdown and RMD files

The most common way to use R in this class will be in R Markdown (.Rmd) files. These can combine LaTeX type setting, R code chunks, and visual outputs. For example, you can write the following true statement:

$$\forall w, b, n \in \mathbb{W} \text{ such that } w \geq n \text{ and } b \geq n, \sum_{k=0}^{n} \frac{\binom{w}{k}\binom{b}{n-k}}{\binom{w+b}{n}} = 1$$

(Why is this true?) Because it's the hypergeometric PMF.

1. Write a true statement with an integral.

$$\int_{-\infty}^{\infty} e^{-x^2/2} dx = \sqrt{2\pi}$$

R with flow control

While loops

If you doubted the first statement, you can verify it by adding an R chunk with code. To add a chunk, click on the green +C button up top and select R (or just type out the three ticks).

```r
w <- 10 # Sets w to 10
b = 20 # Sets b to 20, same as <-
n <- 15
total <- 0
k <- 0

while (k <= n) { # Run the statements inside the loop until k > n
  # Set the 'total' variable to itself plus the next term in the sum.
  total <- total + (choose(w, k) * choose(b, n-k)) / choose(w + b, n)

  # Increment k
  k <- k + 1
}

print(total)
```

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

To run the chunk, either hit `Command-Shift-Enter` (Mac) or hit the green arrow in the top right of the chunk.

2. Write a loop that prints the integers from -5 to 5 inclusive.

```r
i = -5
while (i <= 5) {
  print(i)
  i = i + 1
}
```

```
## [1] -5
## [1] -4
## [1] -3
## [1] -2
## [1] -1
## [1] 0
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

Manuals

In the earlier chunk, `choose` is an R function that calculates a binomial coefficient. You can run `?choose` to see more information about this function.

You can also type `?choose` in the "Console" at the bottom of the screen to get the same effect without adding a chunk. In the console, you can also verify that variables outside of functions are stored in memory until explicitly removed or overwritten. For example, type `total` or `print(total)` into the console to see that it is still contains the value 1. This can be useful for debugging after running code chunks.

For loops, vectorization, and sapply

We can also evaluate this sum four other ways. First we'll use a for loop:

```r
total <- 0

for (k in 0:n) { # For k = 0, then k = 1, ... finally k = n
  total <- total + (choose(w, k) * choose(b, n-k)) / choose(w + b, n)
}

print(total)
```

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

A `for` loop automatically increments `k` without an extra line to explicitly do so.

Second, we'll use a `while` loop with an `if` statement:

```r
total <- 0
k <- 0

while (TRUE) { # Run forever until broken
  total <- total + (choose(w, k) * choose(b, n-k)) / choose(w + b, n)

  k <- k + 1

  if (k > n) { # Break the while loop when k > n
    break
  }
}

print(total)
```

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

Third, we'll use vectorization. This is the best way to loop in R.

```r
k <- 0:n # Assign k to be a vector containing the elements 0 through n
print(k)
```

```
##  [1]  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
```

```
print(k[1:10]) # Print a subvector of the first 10 elements (R uses 1-based indexing)
```

```
##  [1] 0 1 2 3 4 5 6 7 8 9
```

```
# Use vectorization and sum over all the elements
total <- sum((choose(w, k) * choose(b, n-k)) / choose(w + b, n))

print(total)
```

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

Fourth, we'll create a function and apply it over a vector.

```
# Create a function called single_hyper with k, w, b, and n as parameters
single_hyper <- function(k, w, b, n) {
  value_to_return <- (choose(w, k) * choose(b, n-k)) / choose(w + b, n)
  return (value_to_return) # Return the calculated value
}

# Apply the function single_hyper to each element in k and sum the result
total <- sum(sapply(k, single_hyper, w=w, b=b, n=n))

print(total)
```

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

3. Use a for loop, vectorization, and `sapply` to calculate $\sum_{i=1}^{1000} \frac{1}{i}$.

```
total <- 0
for (i in 1:1000) {
  total <- total + 1/i
}
print(total)
```

```
## [1] 7.485471
```

```
print(sum(1/(1:1000)))
```

```
## [1] 7.485471
```

```
# Lambda function in R, you could also write out the function
print(sum(sapply(1:1000, function(x) {1/x})))
```

```
## [1] 7.485471
```

R Functions and more vectorization

There are many other functions built in for R. For example, the function we just wrote already has a built-in version:

```
k <- 0:15
total <- sum(dhyper(k, w, b, n))

print(total)
```

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

This function calculates the probability mass function of a hypergeometric with parameters w, b, and n evaluated at k. Basic operations are also vectorized in R:

```
v1 <- c(2, 4, 6) # Create vector with 3 elements by hand
v2 <- seq(5, 6, 0.5) # All real numbers from 5 to 6 spaced by 0.5
print(v1 * v2)
```

```
## [1] 10 22 36
```

A convenient (or annoying) feature of R is that vectorized operations will duplicate the smaller vector elements to match the length of the larger vector.

```
# For example,
print(3 * 1:3)
```

```
## [1] 3 6 9
```

```
# And also
v3 <- seq(5, 9, 0.5)
print(v1)
```

```
## [1] 2 4 6
```

```
print(v3)
```

```
## [1] 5.0 5.5 6.0 6.5 7.0 7.5 8.0 8.5 9.0
```

```
print(v1 * v3)
```

```
## [1] 10 22 36 13 28 45 16 34 54
```

```
# But this throws a warning
v4 <- seq(5, 8, 0.5)
print(v1 * v4)
```

```
## Warning in v1 * v4: longer object length is not a multiple of shorter object
## length
```

```
## [1] 10 22 36 13 28 45 16
```

There is some thought that vectorization in R is always faster…

```
now <- Sys.time() # Start timer
to_time <- 10^6
n <- 10
output <- vector(length = to_time)
for (i in 1:to_time) { # Multiply elements in a for loop
  output[i] <- i^2
}
print(difftime(Sys.time(), now))
```

```
## Time difference of 0.026474 secs
```

```
now <- Sys.time()
output <- (1:to_time)^2 # Do vectorized multiplication of elements (in a good way)
print(difftime(Sys.time(), now))
```

```
## Time difference of 0.0028162 secs
```

```
now <- Sys.time()
# Do vectorized multiplication of elements (in a bad way)
output <- sapply(1:to_time, `^`, 2)
print(difftime(Sys.time(), now))
```

```
## Time difference of 0.392004 secs
```

But using the `apply` functions doesn't give you much speed up. Unless there's a built-in vectorized method (which there usually is), a `for` loop is usually the cleanest way to write code.

4. Print the average of $1, 1/2, 1/3, \dots, 1/10^8$ in the fastest way you can. What is the limit of the mean of $1, 1/2, 1/3, ..., 1/n$ as $n \to \infty$?

```r
now <- Sys.time()
print(mean(1/(1:10^8)))
```

```
## [1] 1.89979e-07
```

```r
print(difftime(Sys.time(), now))
```

```
## Time difference of 0.8236809 secs
```

The limit is 0. As shown here, $\sum_{i=1}^{n} \frac{1}{i} \leq \log_2(n+1)$. Then, by L'Hospital's rule, $\lim_{n\to\infty} \frac{\log_2(n+1)}{n} = \lim_{n\to\infty} \frac{1}{(n+1)\ln(2)} \to 0$. Thus,

$$0 < \frac{1}{n}\sum_{i=1}^{n} \frac{1}{i} \leq \frac{\log_2(n+1)}{n} \to 0$$

so $\frac{1}{n}\sum_{i=1}^{n} \frac{1}{i} \to 0$ by the squeeze theorem.

Knitting

Now is a good point to knit your code. Press `Command-Shift-K` or the blue knit button at the top.

Setting headers in your chunk can have helpful effects. For example, `cache=T` stores the chunk outputs when knitting so they don't have to run again unless you change the code. The option `warning=F` prevents ugly warning messages from appearing in your output. The option `echo=F` includes the code output but not the code. The option `eval=F` includes the code but not the output. The option `include=F` skips the chunk when knitting (no code or output).

```
## [1] "Chunk was here"
```

Importing data, working with data, and plotting

Importing data

You can import data from a CSV (comma separated values) file to a `data.frame` as follows. If the data cannot be found, make sure you're in the right directory by right clicking `Nickols_R_Bootcamp.Rmd` at the top left and selecting "Set Working Directory."

```
countries <- read.csv("data/country_stats.csv", check.names = F)
```

This section will deal with a data set of country-level statistics from UNdata and Varieties of Democracy.

A few columns will be useful for the following questions:

- `GDP`: GDP per capita
- `EducExpend`: Public expenditure on education (% of GDP)
- `Doctors`: Physicians (per 1000 population)

Selecting rows and columns

You can select columns by name or by index:

```
print(countries$GDP[1:10]) # Print the first 10 GDP per capitas
```

```
##  [1]   504   544   497 40065 35748 14971  1753  3432  3607 37710
```

```
print(countries[,3][1:10]) # Same thing
```

```
##  [1]   504   544   497 40065 35748 14971  1753  3432  3607 37710
```

The comma in the second line above is important. You can select rows by putting the number before the comma.

```
print(countries[1,][1:3]) # First 3 columns of row 1
```

```
##       Country Year GDP
## 1 Afghanistan 2010 504
```

```
print(countries[1:5,1:3]) # First 3 columns of rows 1-5
```

```
##       Country Year   GDP
## 1 Afghanistan 2010   504
## 2 Afghanistan 2015   544
## 3 Afghanistan 2019   497
## 4     Andorra 2005 40065
## 5     Andorra 2015 35748
```

You can also select rows by a condition.

```
print(dim(countries)) # Original dimensions
```

```
## [1] 280   5
```

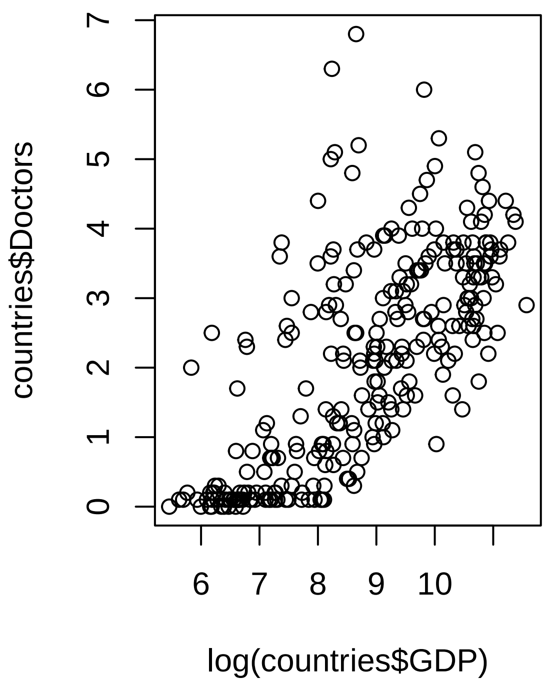
```
# Subset the data frame to only countries with GDPs per capita of over $10000
print(dim(countries[countries$GDP > 10000,]))
```

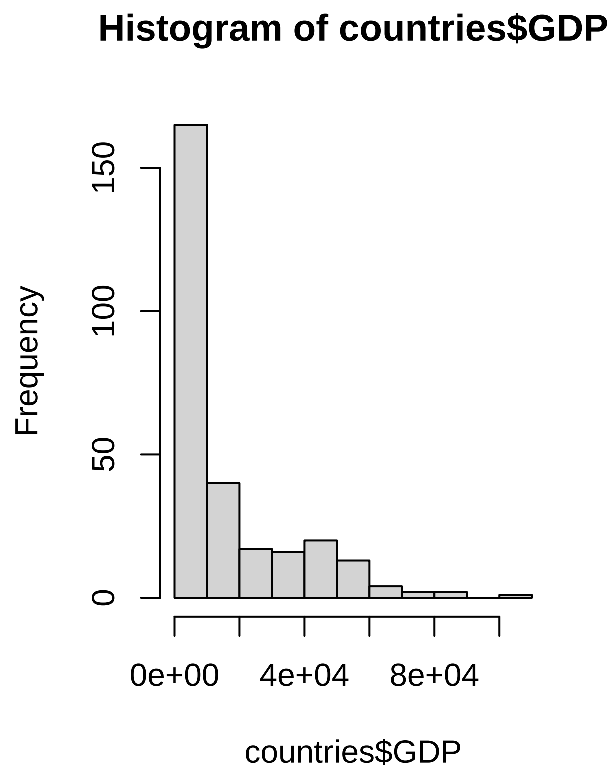
```
## [1] 115   5
```

Base R plots

The following code plots the relationship between log GDP and the number of doctors:

```r
par(mfrow = c(1,2)) # Place two plots side by side (1 row, 2 columns)
plot(log(countries$GDP), countries$Doctors)
hist(countries$GDP)
```





These are quite ugly, but we can make them prettier in `ggplot`.

GGplot

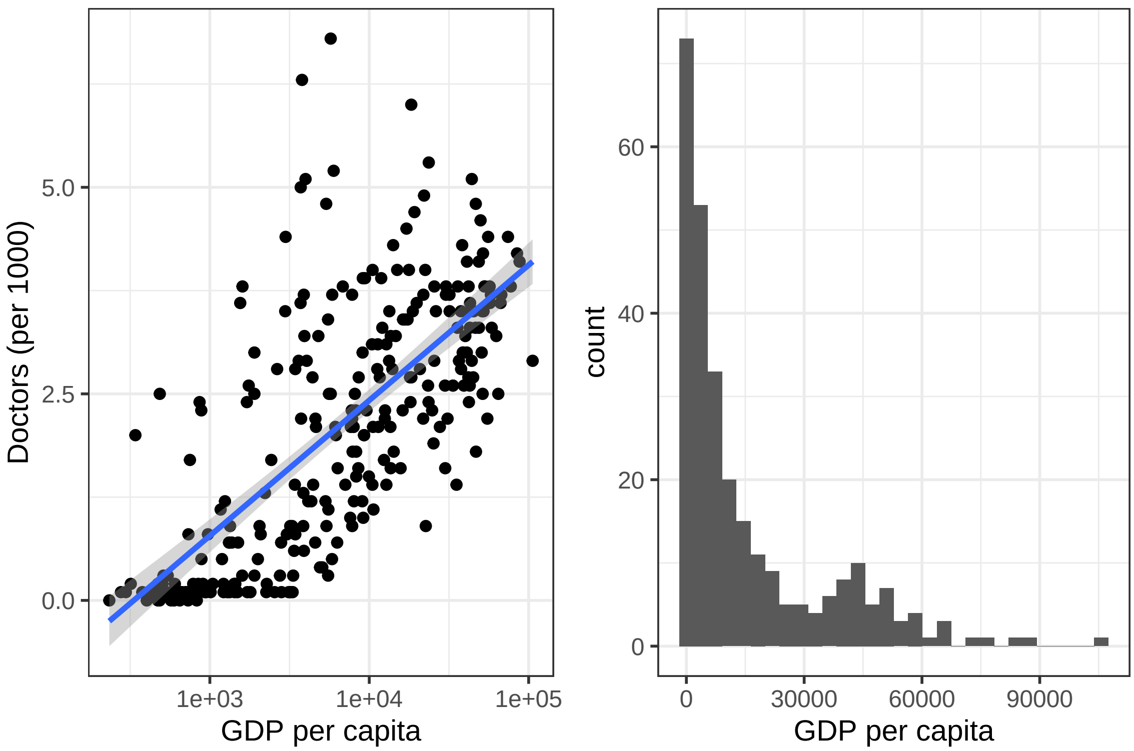
The following chunk installs and loads the package `ggplot`. You'll need to install packages only once but load them in each file.

```r
if(!"ggplot2" %in% rownames(installed.packages())) {
  install.packages("ggplot2")
}
library(ggplot2)

if(!"gridExtra" %in% rownames(installed.packages())) {
  install.packages("gridExtra")
}
library(gridExtra)
```

The following code creates the earlier plots, but prettier.

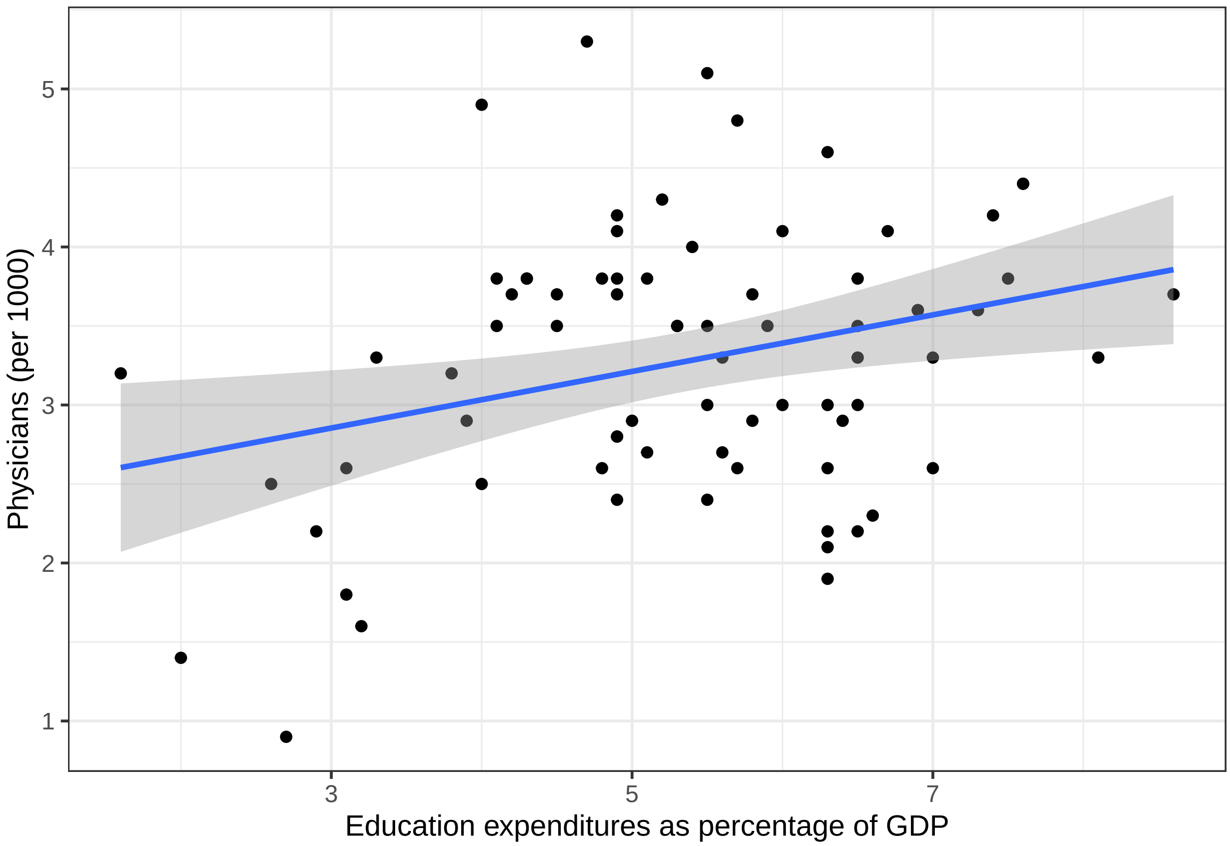
```r
# GDP per capita is x, Doctors is y
p1 <- ggplot(countries, aes(x = GDP, y = Doctors)) +
  geom_point() + # Plot points
  geom_smooth(method = 'lm', formula = "y~x") + # Plot line
  # Log transform x axis and set break points
  scale_x_continuous(trans = 'log10', breaks = c(1000, 10000, 100000)) +
  ylab("Doctors (per 1000)") + # Rename y axis
  xlab("GDP per capita") + # Rename x axis
```

```
  theme_bw() # Change background

p2 <- ggplot(countries, aes(x = GDP)) +
  geom_histogram(bins = 30) +
  xlab("GDP per capita") +
  theme_bw()

# You can display individual plots by not assigning them to p1 or p2

grid.arrange(p1, p2, ncol=2) # Put plots side by side
```



5. Plot the relationship between education expenditures and doctors in countries with GDPs per capita of at least $20,000.

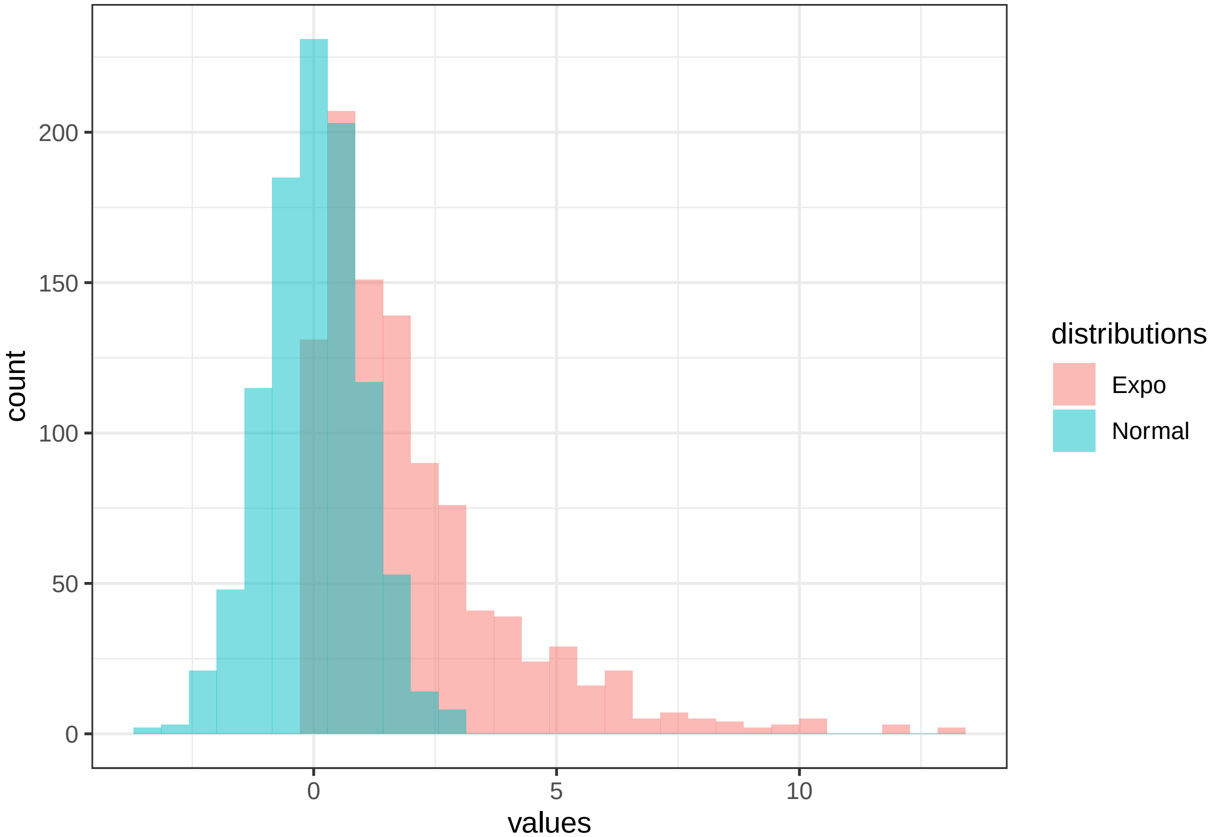
```
over_20 <- countries[countries$GDP > 20000,]
ggplot(over_20, aes(x = EducExpend, y = Doctors)) +
  geom_point() +
  geom_smooth(method = 'lm', formula = "y~x") +
  xlab("Education expenditures as percentage of GDP") +
  ylab("Physicians (per 1000)") +
  theme_bw()
```



Random number usage

One of the main benefits of R is its ability to generate and manipulate random numbers easily. Most common distributions are already ready to go. Make sure to set a seed so your code is reproducible. For example:

```
set.seed(111)

n <- 1000
normals <- rnorm(n, 0, 1) # Generate 1000 standard normals
expos <- rexp(n, 1/2) # 1000 exponentials with rate 1/2 (mean 2)

# Create a data.frame with the normal and expontials in one column
# and the names in another column
df <- data.frame("values" = c(normals, expos),
                 "distributions" = rep(c("Normal", "Expo"), each = n))

ggplot(df, aes(x = values, fill = distributions)) +
  geom_histogram(alpha = 0.5, position="identity", bins = 30) +
  theme_bw()
```



You can also calculate summary statistics for the distributions:

```
print(c(mean(normals), median(normals)))
```

```
## [1] 0.01080923 0.01967487
```

```
print(summary(normals))
```

```
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -3.32334 -0.65366  0.01968  0.01081  0.67559  2.92603
```

There are also quantile functions (they start with “q”), density functions (they start with “d”), and cumulative density functions (they start with “p”).

```
print(pnorm(-3:3))
```

```
## [1] 0.001349898 0.022750132 0.158655254 0.500000000 0.841344746 0.977249868
## [7] 0.998650102
```

```
print(qnorm(c(1 - (1-0.997)/2, 1 - (1-0.95)/2, 1 - (1-0.68)/2)))
```

```
## [1] 2.9677379 1.9599640 0.9944579
```

(Which rule does the line above show?) The 68-95-99.7 rule.

The functions `rep` is useful for repeating values, and the function `replicate` is useful for replicating computations.

```
print(rep(10, 5))
```

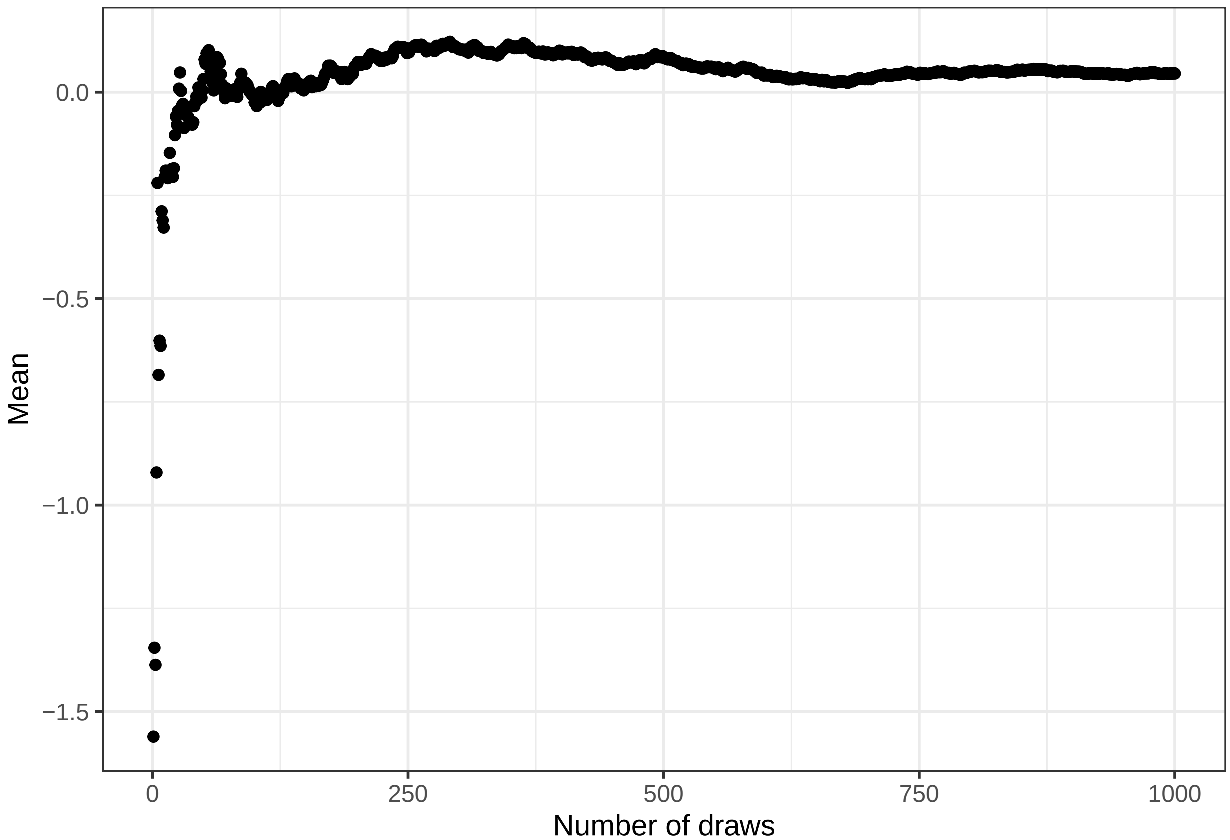
```
## [1] 10 10 10 10 10
```

```
print(replicate(10, mean(rnorm(100, mean = 1, sd = 2))))
```

```
##  [1] 0.8934618 1.0457099 0.9842224 1.2385490 1.1066039 0.5112138 1.0136824
##  [8] 1.2919768 1.2427886 0.9372874
```

6. Show visually the law of large numbers applying to draws from $\mathcal{N}(0,1)$.

```
n <- 1000
normals <- rnorm(n, 0, 1)
running_mean <- function(i, x) { # Returns the mean of elements 1 to i of x
  return (mean(x[1:i]))
}

df <- data.frame(y=sapply(1:n, running_mean, normals), x = 1:n)

ggplot(df, aes(x = x, y = y)) +
  geom_point() +
  theme_bw() +
  xlab("Number of draws") +
  ylab("Mean")
```



This concludes the instructional material. One last thing is that you can include a picture (e.g., of handwritten math) like this: