Case Study 1

AKSTA Statistical Computing

Carla Salazar Veneta Grigorova Juraj Simkovic

1. Ratio of Fibonacci numbers

a.

Write two different R functions which return the sequence $r_i = F_{i+1}/F_i$ for $i = 1, \ldots, n$ where F_i is the ith Fibonacci number, once using `for` and once using `while`.

Function using a for loop

```
fib_ratio_for <- function(n) {
  fib <- numeric(n + 1)
  fib[1] <- 1
  fib[2] <- 1
  for (i in 3:(n + 1)) {
    fib[i] <- fib[i - 1] + fib[i - 2]
  }
  return(fib[2:(n + 1)] / fib[1:n])
}
print(fib_ratio_for(10))
```

```
##  [1] 1.000000 2.000000 1.500000 1.666667 1.600000 1.625000 1.615385 1.619048
##  [9] 1.617647 1.618182
```

Function using a while loop

```
fib_ratio_while <- function(n){
  fib1 <- numeric(n + 1)
  fib1[1] <- 1
  fib1[2] <- 1
  i <- 3
  while (i <= (n+1)){
    fib1[i] <- fib1[i - 1] + fib1[i - 2]
    i <- i+1
  }
  return(fib1[2:(n + 1)] / fib1[1:n])
}
print(fib_ratio_while(20))
```

```
##  [1] 1.000000 2.000000 1.500000 1.666667 1.600000 1.625000 1.615385 1.619048
##  [9] 1.617647 1.618182 1.617978 1.618056 1.618026 1.618037 1.618033 1.618034
## [17] 1.618034 1.618034 1.618034 1.618034
```

b.

Benchmark the two functions for $n = 200$ and $n = 2000$ (you can use package **microbenchmark** or package **bench** for this purpose). Which function is faster?

```
library(microbenchmark)

benchmark <- microbenchmark(
  fib_ratio_for_200 = fib_ratio_for(200),
  fib_ratio_while_200 = fib_ratio_while(200),
  fib_ratio_for_2000 = fib_ratio_for(2000),
  fib_ratio_while_2000 = fib_ratio_while(2000)
)
summary(benchmark)
```

```
##                   expr    min       lq      mean   median       uq     max
## 1    fib_ratio_for_200  7.462   8.2410   8.57310   8.4460   8.8150  12.054
## 2  fib_ratio_while_200 10.291  10.9060  11.47303  11.2545  11.6235  28.413
## 3   fib_ratio_for_2000 71.463  73.8820  76.74544  75.6245  78.1050 125.296
## 4 fib_ratio_while_2000 98.892 100.7165 104.57173 102.0490 108.4860 119.228
##   neval
## 1   100
## 2   100
## 3   100
## 4   100
```

The `for_loop` is faster because it's more efficient in R. When using a `for` loop, R automatically handles the loop index and knows how many times to repeat. This makes it run faster with less overhead.

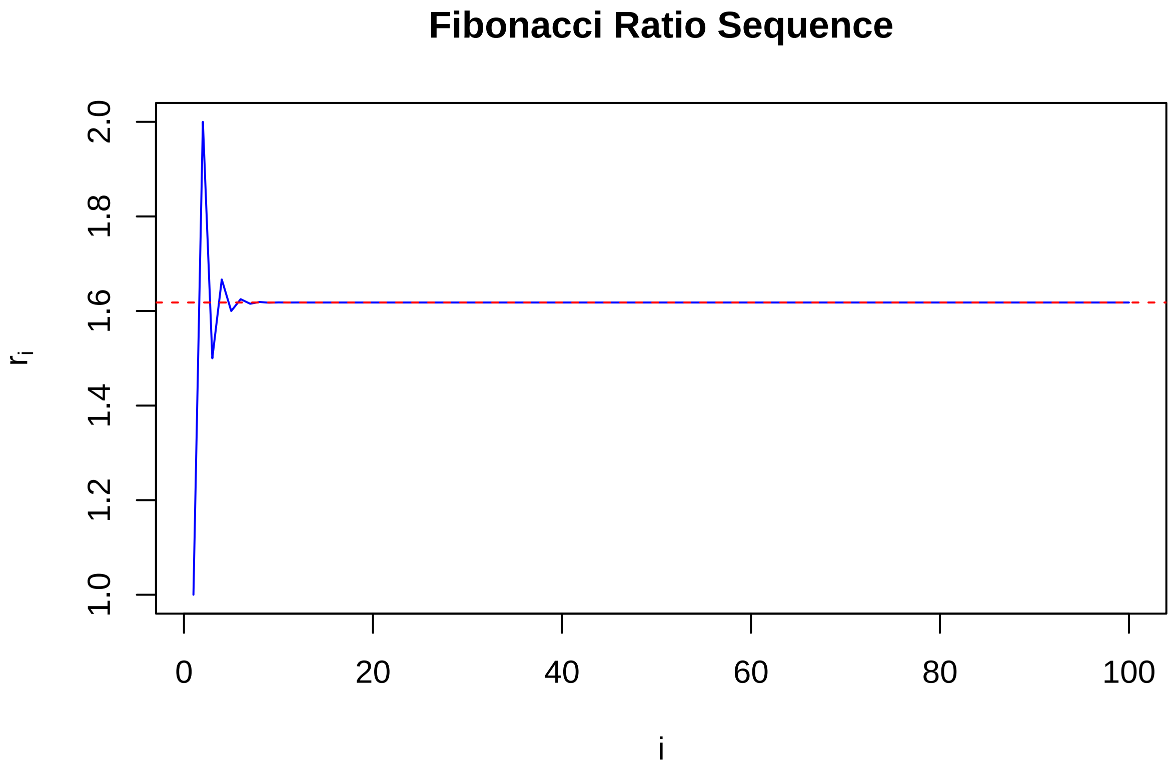
In contrast, a `while` loop checks the condition every time and needs manual updates to the counter. This adds a little extra work each time the loop runs.

So, even though both loops do the same thing, `for_loop` is faster and more consistent, especially when the number of iterations is known.

c.

Plot the sequence for $n = 100$. For which n does it start to stabilize to a value? What number does the sequence converge to?

```
ratios <- fib_ratio_for(100)
plot(ratios, type = "l", col = "blue", main = "Fibonacci Ratio Sequence",
     xlab = "i", ylab = expression(r[i]))
abline(h = (1 + sqrt(5)) / 2, col = "red", lty = 2)
```



The ratio r_i starts to stabilize around $i = 20$. It converges to the golden ratio:

$$\frac{1+\sqrt{5}}{2} \approx 1.618$$

2. Gamma function

a.

Write a function to compute the following for n a positive integer using the `gamma` function in base R.

$$\rho_n = \frac{\Gamma((n-1)/2)}{\Gamma(1/2)\Gamma((n-2)/2)}$$

```
rho_n <- function(n) {
  gamma((n - 1) / 2) / (gamma(1 / 2) * gamma((n - 2) / 2))
}
```

b.

Try $n = 2000$. What do you observe? Why do you think the observed behavior happens?

```
rho_n(2000)
```

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

The result is NaN because of numerical overflow. The the Gamma function grows very fast, and R can't handle such large values directly.

c.

Write an implementation which can also deal with large values of $n > 1000$.

```
rho_n_stable <- function(n) {
  log_rho <- lgamma((n - 1) / 2) - (lgamma(1 / 2) + lgamma((n - 2) / 2))
  exp(log_rho)
}
```

Now it works even for large n:

```
rho_n_stable(2000)
```

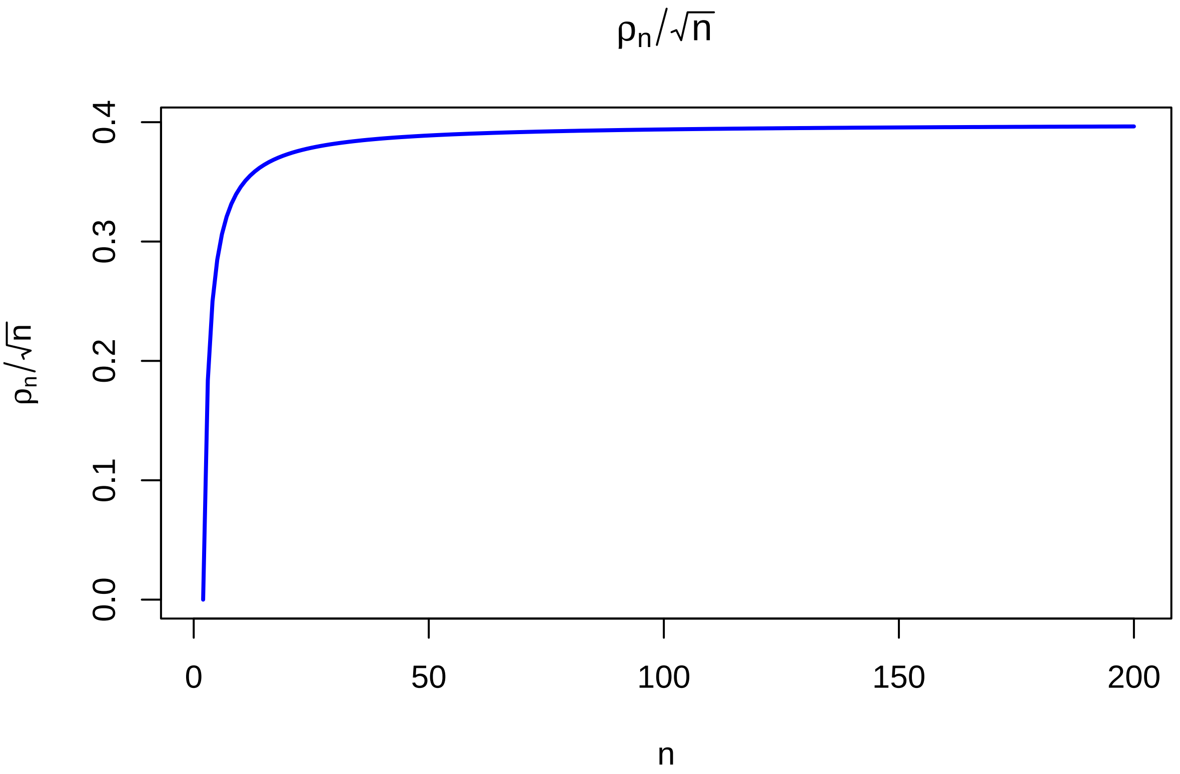
```
## [1] 17.83009
```

d.

Plot $\rho_n/\sqrt{n}$ for different values of n. Can you guess the limit of $\rho_n/\sqrt{n}$ for $n \to \infty$?

```
n_vals <- 1:200
ratios <- sapply(n_vals, function(n) rho_n_stable(n) / sqrt(n))

plot(n_vals, ratios, type = "l", col = "blue", lwd = 2,
     main = expression(rho[n] / sqrt(n)),
     xlab = "n", ylab = expression(rho[n] / sqrt(n)))
```



The plot shows that $\rho_n/\sqrt{n}$ converges to a constant as $n \to \infty$, which is consistent with asymptotic properties of the Gamma function.

3. Weak law of large numbers

The law of large numbers states that if you repeat an experiment independently a large number of times and average the result, what you obtain should be close to the expected value. Formally, the weak law of large numbers states that if $X_1, X_2, \ldots X_n$ are independently and identically distributed (iid) random variables with a finite expected value $E(X_i) = \mu < \infty$, then, for any $\epsilon > 0$ $\lim_{n\to\infty} P(|\bar{X}_n - \mu| > \epsilon) = 0$ where $\bar{X}_n = \frac{X_1+X_2+\ldots+X_n}{n}$ is the sample mean (the sample mean converges in probability to the expected value).

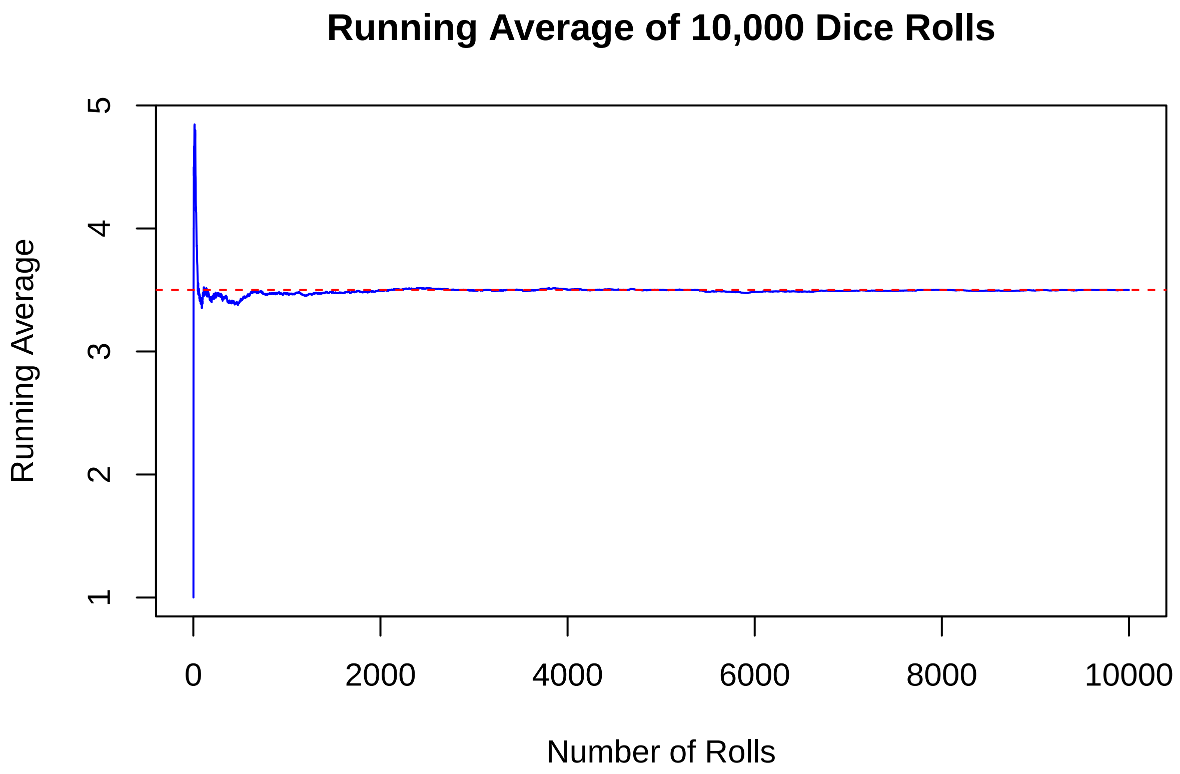
a.

- In R, simulate rolling a fair 6-sided die $n = 10\,000$ times using vectorized functions.
- After each roll, compute the cumulative mean of all previous rolls.

```
n <- 10000
rolls <- sample(1:6, n, replace = TRUE)
running_avg <- cumsum(rolls) / (1:n)
```

- Plot the running average against the number of rolls and comment on the result.

```
plot(running_avg, type = "l", col = "blue",
     main = "Running Average of 10,000 Dice Rolls",
     xlab = "Number of Rolls", ylab = "Running Average")
abline(h = 3.5, col = "red", lty = 2)
```



This plot shows how the running average changes as we roll the die more times. At the start, the average jumps around a lot because each new roll has a big effect. But after a while, it starts to settle down.

After about 3000 rolls, it hardly changes anymore. This matches what the Weak Law of Large Numbers says: the average of many die rolls should get closer to the true mean (3.5).

b.

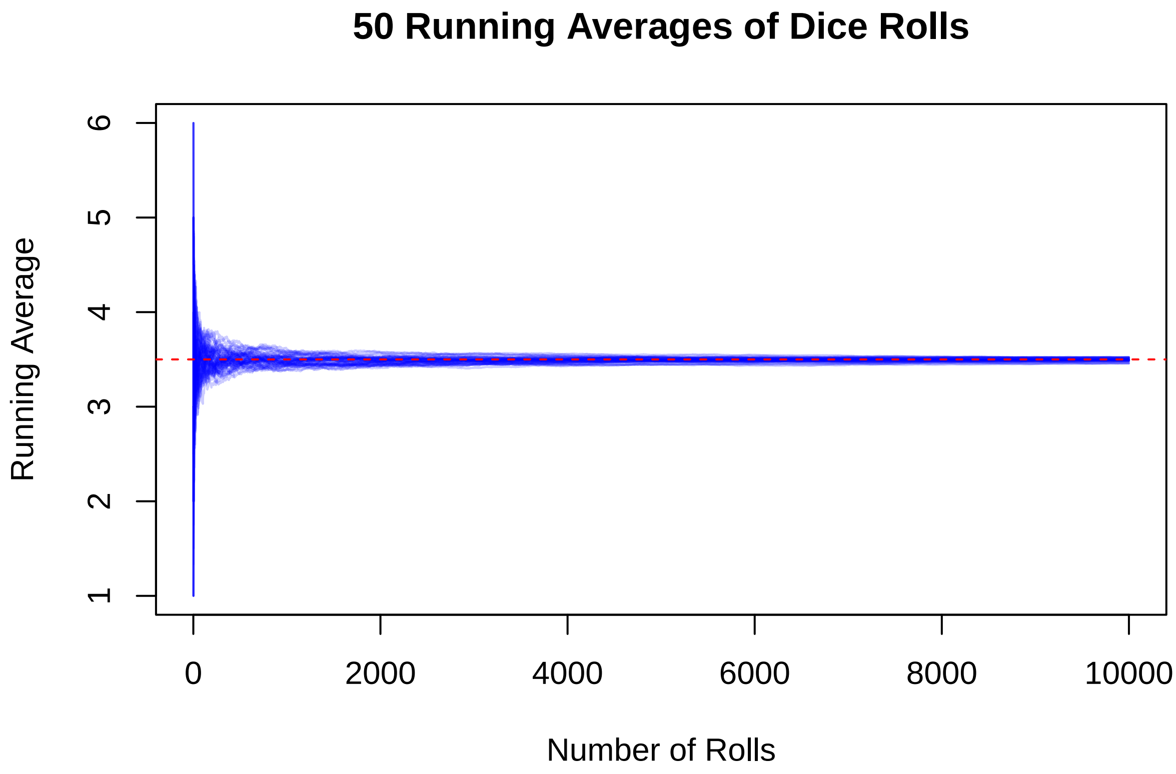
- Replicate the die-rolling experiment $M = 50$ times using a for loop. This will result in 50 paths which you can plot against the number of rolls.

```
M <- 50
paths <- matrix(NA, nrow = n, ncol = M)

for (i in 1:M) {
  rolls <- sample(1:6, n, replace = TRUE)
  paths[, i] <- cumsum(rolls) / (1:n)
}
```

- Plot all 50 paths on the same graph to observe the variation in the running averages.

```
matplot(paths, type = "l", lty = 1, col = rgb(0, 0, 1, 0.2),
        xlab = "Number of Rolls", ylab = "Running Average",
        main = "50 Running Averages of Dice Rolls")
abline(h = 3.5, col = "red", lty = 2)
```



Each line in this plot shows how the average of die rolls changes over time for a single experiment. At the beginning, the paths are all over the place because the average is very sensitive to the first few values. But as more rolls are added, the averages become more stable and start to cluster around 3.5.

By the end, almost all paths are very close to 3.5, which is exactly what we expect from the Weak Law of Large Numbers. It shows that even though each experiment is random, the average gets close to the expected value when we repeat it enough times.

c.

What is the theoretical expected value of a fair 6-sided die?

The expected value μ of a fair 6-sided die is:

$$\mu = \frac{1+2+3+4+5+6}{6} = 3.5$$

d.

Comment on how the running mean of the paths behaves relative to the expected value as n increases.

As n approaches infinity, the running average converges to the expected value 3.5, confirming the Weak Law of Large Numbers.

e.

- Modify the code in b. to use a while loop.
- Stop the loop when the proportion of replications where the difference between the running average and the expected value lies outside the interval $[-\epsilon, \epsilon]$ is less than 0.05. Use $\epsilon = 0.01$.
- How high is n at the end of the loop?

```r
epsilon <- 0.01
max_iter <- 1e6

rolls_matrix <- matrix(sample(1:6, max_iter * M, replace = TRUE), nrow = max_iter, ncol = M)
running_avg <- matrix(NA, nrow = max_iter, ncol = M)

running_avg[1, ] <- rolls_matrix[1, ]
j <- 2
prop_in <- 0

while (j <= max_iter && prop_in < 0.95) {
  running_avg[j, ] <- (running_avg[j - 1, ] * (j - 1) + rolls_matrix[j, ]) / j
  prop_in <- mean(abs(running_avg[j, ] - 3.5) <= epsilon)
  j <- j + 1
}

cat("Stopped at n =", j - 1, "with proportion in range:", round(prop_in, 3))
```

```
## Stopped at n = 82312 with proportion in range: 0.96
```

f.

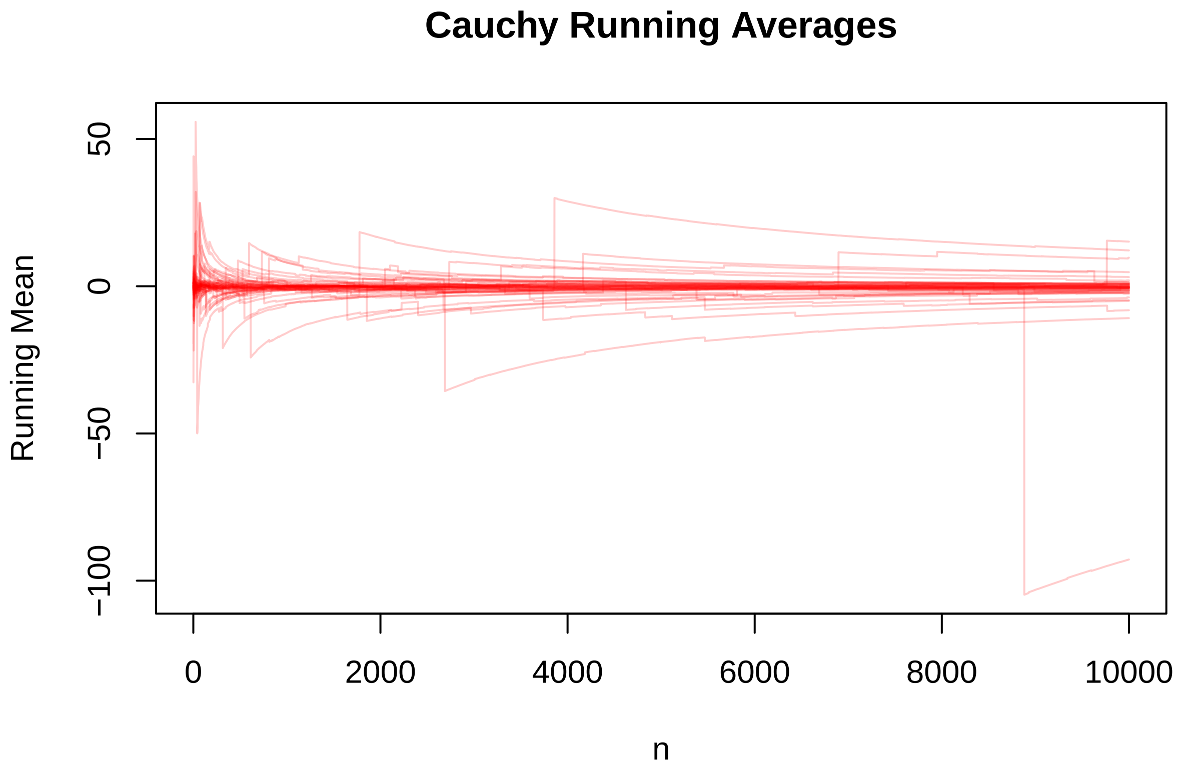
- Repeat the analysis in b, but instead of simulating from a discrete uniform distribution, simulate from a continuous standard cauchy distribution using `rcauchy()`. The (standard) cauchy distribution arises from the ratio of two independent mean zero (standard) normally distributed random variables.

```r
cauchy_paths <- matrix(NA, nrow = n, ncol = M)

for (i in 1:M) {
  x <- rcauchy(n)
  cauchy_paths[, i] <- cumsum(x) / (1:n)
}
```

- Plot the running mean against increasing n. Comment on whether the weak Law of Large Numbers seems to hold for this distribution. Explain why or why not.

```r
matplot(cauchy_paths, type = "l", lty = 1, col = rgb(1, 0, 0, 0.2),
        main = "Cauchy Running Averages",
        xlab = "n", ylab = "Running Mean")
```



This plot shows the running averages for 50 simulations of Cauchy-distributed random variables. Unlike the die rolls, these paths don't settle down or stabilize around a specific value. Some paths shoot up or down dramatically even after thousands of iterations.

This happens because the Cauchy distribution doesn't have a defined mean or variance. So the Law of Large Numbers doesn't apply here. The average doesn't converge like it did with the uniform distribution of the die rolls. Instead, it stays unstable, and a few extreme values can heavily influence the result even with many observations.

4. Central limit theorem

The central limit theorem states that, under certain conditions, the sum of a large number of random variables is approximately normal. Consider the case where $X_1, X_2, \ldots, X_n$ are i.i.d. random variables with finite expected values ($E(X_i) = \mu < \infty$) and variances ($Var(X_i) = \sigma^2 < \infty$). The central limit theorem states that the cumulative distribution function (CDF) of the normalized random variable

$$Z_n = \frac{\bar{X}_n - \mu}{\sigma/\sqrt{n}}$$

converges to the standard normal CDF as $n \to \infty$.

a.

Derive the expected value of the sample mean $\bar{X}_n = \frac{X_1+X_2+\ldots+X_n}{n}$.

Let $X_1, X_2, \ldots, X_n$ be i.i.d. random variables with expected value $\mu = E[X_i]$.

The sample mean is defined as:

$$\bar{X}_n = \frac{1}{n}\sum_{i=1}^{n} X_i$$

To find the expected value of $\bar{X}_n$, we use the linearity of expectation, which says:

$$E\left[\sum X_i\right] = \sum E[X_i]$$

Now apply it to the sample mean:

$$E[\bar{X}_n] = E\left[\frac{1}{n}\sum_{i=1}^{n} X_i\right]$$

Pull out the constant $\frac{1}{n}$:

$$= \frac{1}{n}\sum_{i=1}^{n} E[X_i]$$

Since all the variables are identically distributed, each one has the same expected value, which we call μ. So when we average n of them, we're just averaging n copies of μ, which gives us μ again.

$$= \frac{1}{n} \cdot n \cdot \mu = \mu$$

So, the expected value of the sample mean is equal to the population mean:

$$E[\bar{X}_n] = \mu$$

b.

Derive the variance of the sample mean $\bar{X}_n = \frac{X_1+X_2+\ldots+X_n}{n}$.

Let $X_1, X_2, \ldots, X_n$ be i.i.d. random variables with variance σ^2.

The sample mean is:

$$\bar{X}_n = \frac{1}{n}\sum_{i=1}^{n} X_i$$

We want to compute the variance of $\bar{X}_n$.

Use the rule:

$$\mathrm{Var}(aX) = a^2 \cdot \mathrm{Var}(X)$$

Apply it to the sample mean:

$$\mathrm{Var}(\bar{X}_n) = \mathrm{Var}\left(\frac{1}{n}\sum_{i=1}^{n} X_i\right)$$

Pull out the constant:

$$= \frac{1}{n^2} \cdot \mathrm{Var}\left(\sum_{i=1}^{n} X_i\right)$$

Since the X_i are independent, their variances add:

$$= \frac{1}{n^2} \cdot \sum_{i=1}^{n} \mathrm{Var}(X_i) = \frac{1}{n^2} \cdot n \cdot \sigma^2$$

Simplify:

$$\mathrm{Var}(\bar{X}_n) = \frac{\sigma^2}{n}$$

So the variance of the sample mean decreases as n increases, meaning the sample mean becomes more stable with larger sample sizes.

c.

The central limit theorem makes few assumption on the distribution of the random variables X_i. Consider again the die-rolling experiment where X_i is the random variable giving the number in roll i.

- Simulate Die Rolls: For each value of n (where $n = \{100, 1000, 10000\}$), simulate rolling a die n times. Record the mean of the n rolls.
- Repeat the above process $M = 100$ times for each value of n. This will give you 100 sample means for each n.

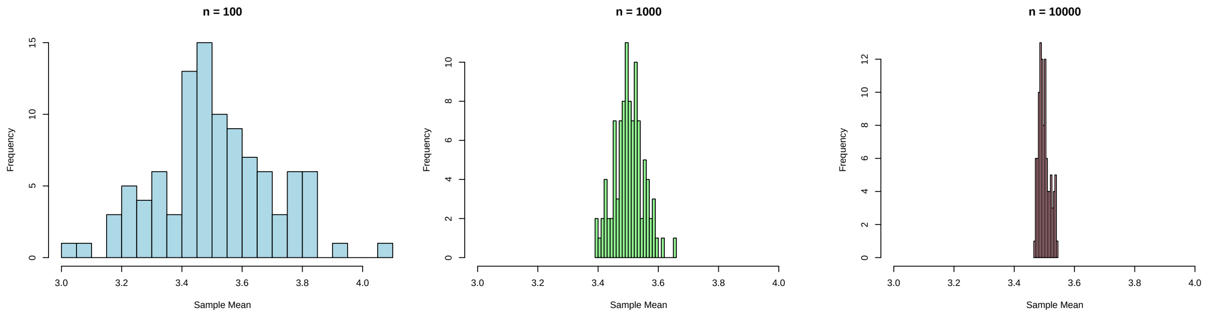
```r
get_means <- function(n, M) {
  replicate(M, mean(sample(1:6, n, replace = TRUE)))
}

means_100 <- get_means(100, 100)
means_1000 <- get_means(1000, 100)
means_10000 <- get_means(10000, 100)
```

- Plot a histogram of the 100 sample means for each value of n. Observe the shape of the histograms.

```r
x_limits <- range(c(means_100, means_1000, means_10000))

par(mfrow = c(1, 3))
hist(means_100, main = "n = 100", xlab = "Sample Mean", col = "lightblue",
     breaks = 20, xlim = x_limits)
hist(means_1000, main = "n = 1000", xlab = "Sample Mean", col = "lightgreen",
     breaks = 20, xlim = x_limits)
hist(means_10000, main = "n = 10000", xlab = "Sample Mean", col = "lightpink",
     breaks = 20, xlim = x_limits)
```



- Discuss how the distribution of the sample means changes as n increases. Comment on whether the histograms appear to approach a normal distribution as n, illustrating the Central Limit Theorem.

The distribution of the sample means changes as the number of die rolls n increases:

- **When $n = 100$:** The sample means are spread out and the shape is a bit uneven. There's still a lot of variation, but the values seem to center around 3.5.
- **When $n = 1000$:** The distribution is narrower and looks more symmetric. The sample means are more tightly grouped, and it already starts to look like a normal distribution.
- **When $n = 10000$:** The histogram is very concentrated around the mean of 3.5, and the shape looks almost perfectly normal.

As n gets larger, the sample means become less variable and more normally distributed. Even though we're rolling a fair die (which gives discrete uniform outcomes), the average of many rolls behaves like it's coming from a normal distribution when we repeat the experiment enough times.

d.

Compute expected value μ and the variance σ^2 for the random variable X_i (giving the number in die-roll i), which follows a discrete uniform distribution.

The possible outcomes are $\{1, 2, 3, 4, 5, 6\}$, each with equal probability $\frac{1}{6}$.

Expected Value μ:

$$\mu = \frac{1+2+3+4+5+6}{6} = \frac{21}{6} = 3.5$$

Variance σ^2:

$$\sigma^2 = \frac{1}{6}\sum_{i=1}^{6}(i-3.5)^2 = \frac{(6.25+2.25+0.25+0.25+2.25+6.25)}{6} = \frac{17.5}{6} = \frac{35}{12} \approx 2.9167$$

So the theoretical values are:

- $\mu = 3.5$
- $\sigma^2 = \frac{35}{12} \approx 2.9167$

e.

Now that you computed μ and σ^2, consider Z_n.

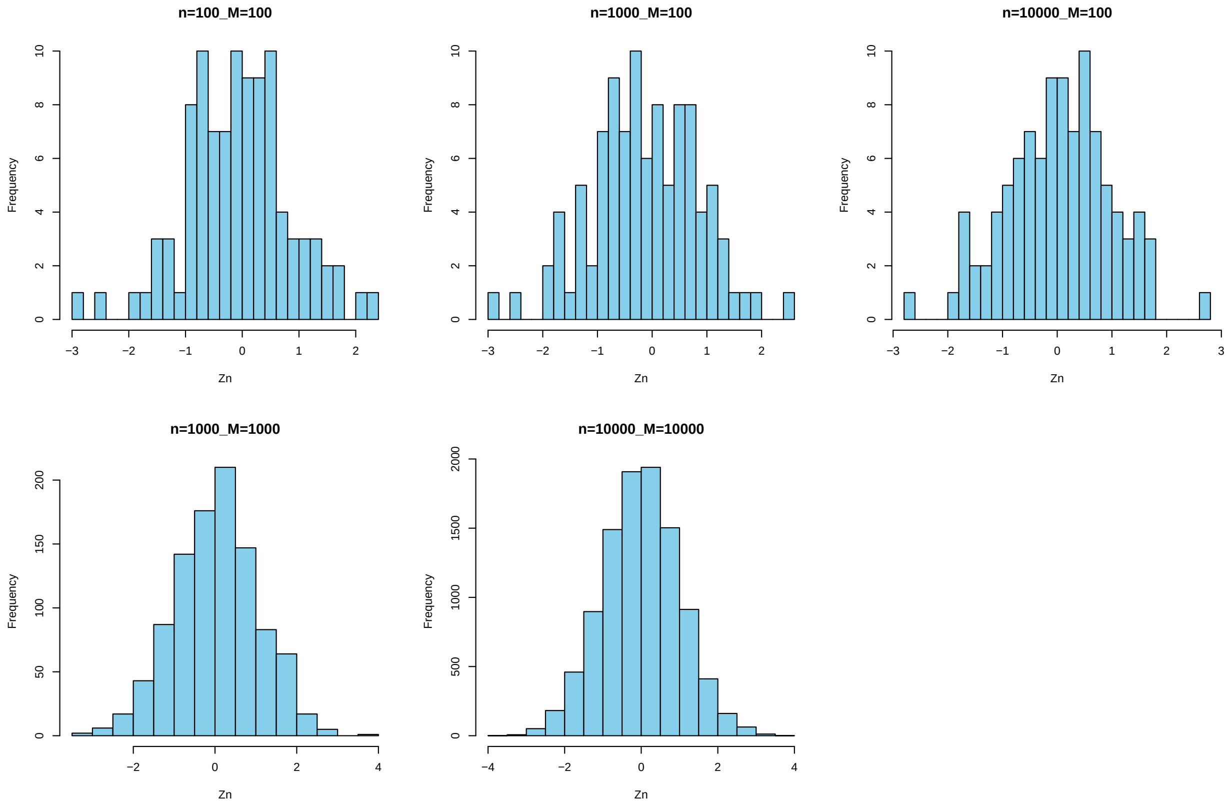
- For each value of n (where $n = \{100, 1000, 10000\}$) and for $M = \{100, 1000, 10000\}$ replications, simulate the rolling of n dice and calculate the Z_n.

```
simulate_Zn <- function(n, M, mu = 3.5, sigma = sqrt(35/12)) {
  replicate(M, {
    x <- sample(1:6, n, replace = TRUE)
    (mean(x) - mu) / (sigma / sqrt(n))
  })
}

Z_data <- list(
  "n=100_M=100" = simulate_Zn(100, 100),
  "n=1000_M=100" = simulate_Zn(1000, 100),
  "n=10000_M=100" = simulate_Zn(10000, 100),
  "n=1000_M=1000" = simulate_Zn(1000, 1000),
  "n=10000_M=10000" = simulate_Zn(10000, 10000)
)
```

- Plot histograms of Z_n for each combination of n and M.

```
par(mfrow = c(2, 3))
for (name in names(Z_data)) {
  hist(Z_data[[name]], main = name, xlab = "Zn", breaks = 20, col = "skyblue")
}
```



- Discuss how the distribution of Z_n changes as n and M increase. Comment on whether the distributions of Z_n appear to approach a normal distribution, illustrating the Central Limit Theorem.

Recall that:

$$Z_n = \frac{\bar{X}_n - \mu}{\sigma/\sqrt{n}}$$

Effect of n:

As n increases:

- The sample mean $\bar{X}_n$ becomes more concentrated around the true mean μ.
- So Z_n becomes less variable, and its distribution gets tighter and more normal-shaped.
- You can clearly see that in the bottom row (large n), where the histograms are symmetric and bell-shaped.

Effect of M:

M controls how many values of Z_n you generate.

As M increases:

- The histogram becomes smoother and more accurate, because you have more data points.
- It doesn't change the distribution's shape but makes it easier to see the bell curve.
- Compare the rough, jagged histograms when M = 100 (top row) with the very smooth ones when M = 10000 (bottom middle).

Overall, the results clearly show the Central Limit Theorem:
The distribution of the standardized sample mean Z_n approaches the standard normal distribution as n and M increase — even though the original data (rolling a fair die) comes from a discrete, non-normal distribution.

5. Readable and efficient code

Read over the code below.

a.

Explain (in text) what the code does.

This function generates 100 random values from a normal distribution using the rnorm() function for variables x and z. Afterwards it checks if a threshold for the sum of the x values are met. If not, the execution is stopped. Then it goes on to fit a linear model using these variables. It also takes the sine function of the residuals and adds a bit of noise to create the next value of x. This is repeated 4 times. At the end a dataframe is created based on the x and newly created y values, which are based on x and some noise.

In the second part of the function - a manual cross validation error estimation is calculated. The function is parted in 4 sections - 1:250, 251:500, 501:750, and 751:1000. On all of these 4 parts, the linear model is fit, predicted and the root mean squared error is calculated. The result is a vector of total 4 RMSE values of each of these 4 chunks.

b.

Explain (in text) what you would change to make the code more readable.

To make the code more readable, simple coding practices could make it better. For example, adding comments to explain what part of the code is responsible for what action (or what is intended by writing such and such code). Secondly, adding meaningful variable names - instead of x,y,z, we could write predictor, response and residuals. Thirdly, a lot of the code is repetitive. Instead of writing the same code over and over, we could wrap the code in a function to make our work more efficient. Lastly, the code is just one big chunk of code - adding spaces and lines to separate different part of the code would also make it more clearer and easier to understand.

c.

Change the code according to b. and wrap it in a function. This function should have at most 10 lines (without adding commands to more lines such as `x <- 1; y <- 2`. Such commands will count as 2 lines!). Check that the function called on the same input outputs the same as the provided code.

```r
set.seed(1)
x <- rnorm(100)
z <- rnorm(100)
if (sum(x >= .001) < 1) {
  stop("step 1 requires 1 observation(s) with value >= .001")
}
fit <- lm(x ~ z)
r <- fit$residuals
x <- sin(r) + .01
```

```r
if (sum(x >= .002) < 2) {
  stop("step 2 requires 2 observation(s) with value >= .002")
}
fit <- lm(x ~ z)
r <- fit$residuals
x <- 2 * sin(r) + .02
if (sum(x >= .003) < 3) {
  stop("step 3 requires 3 observation(s) with value >= .003")
}
fit <- lm(x ~ z)
r <- fit$residuals
x <- 3 * sin(r) + .03
if (sum(x >= .004) < 4) {
  stop("step 4 requires 4 observation(s) with value >= .004")
}
fit <- lm(x ~ z)
r <- fit$residuals
x <- 4 * sin(r) + .04
x
```

```r
set.seed(1)
x <- rnorm(1000)
y <- 2 + x + rnorm(1000)
df <- data.frame(x, y)

cat("Step", 1, "\n")
fit1 <- lm(y ~ x, data = df[-(1:250),])
p1 <- predict(fit1, newdata = df[(1:250),])
r <- sqrt(mean((p1 - df[(1:250),"y"])^2))

cat("Step", 2, "\n")
fit2 <- lm(y ~ x, data = df[-(251:500),])
p2 <- predict(fit2, newdata = df[(251:500),])
r <- c(r, sqrt(mean((p2 - df[(251:500),"y"])^2)))

cat("Step", 3, "\n")
fit3 <- lm(y ~ x, data = df[-(501:750),])
p3 <- predict(fit3, newdata = df[(501:750),])
r <- c(r, sqrt(mean((p3 - df[(501:750),"y"])^2)))

cat("Step", 4, "\n")
fit4 <- lm(y ~ x, data = df[-(751:1000),])
p4 <- predict(fit4, newdata = df[(751:1000),])
r <- c(r, sqrt(mean((p4 - df[(751:1000),"y"])^2)))
r
```

New function:

```r
cross_val_rmse <- function(df, n_splits=4) {
  test_indices <- split(1:nrow(df), cut(seq_along(1:nrow(df)), breaks=n_splits, labels=FALSE))
  rmse <- numeric(n_splits)
  for (i in seq_along(test_indices)) {
    train <- df[-test_indices[[i]], ]
    test <- df[test_indices[[i]], ]
    model <- lm(y ~ x, data=train)
    pred <- predict(model, test)
    rmse[i] <- sqrt(mean((pred - test$y)^2))}
  return(rmse)}
```

6. Measuring and improving performance

Have a look at the code of the function below. It is a function for performing a Kruskal-Wallis test, a robust non-parametric method for testing whether samples come from the same distribution. (Note: we assume no missing values are present in x).

```r
kwtest <- function (x, g, ...)
{
    if (is.list(x)) {
        if (length(x) < 2L)
            stop("'x' must be a list with at least 2 elements")
        if (!missing(g))
            warning("'x' is a list, so ignoring argument 'g'")
        if (!all(sapply(x, is.numeric)))
            warning("some elements of 'x' are not numeric and will be coerced to numeric")
        k <- length(x)
        l <- lengths(x)
        if (any(l == 0L))
            stop("all groups must contain data")
        g <- factor(rep.int(seq_len(k), l))
        x <- unlist(x)
    }
    else {
        if (length(x) != length(g))
            stop("'x' and 'g' must have the same length")
        g <- factor(g)
        k <- nlevels(g)
        if (k < 2L)
            stop("all observations are in the same group")
    }
    n <- length(x)
    if (n < 2L)
        stop("not enough observations")
    r <- rank(x)
    TIES <- table(x)
    STATISTIC <- sum(tapply(r, g, sum)^2/tapply(r, g, length))
    STATISTIC <- ((12 * STATISTIC/(n * (n + 1)) - 3 * (n + 1))/(1 -
        sum(TIES^3 - TIES)/(n^3 - n)))
    PARAMETER <- k - 1L
    PVAL <- pchisq(STATISTIC, PARAMETER, lower.tail = FALSE)
    names(STATISTIC) <- "Kruskal-Wallis chi-squared"
    names(PARAMETER) <- "df"
    RVAL <- list(statistic = STATISTIC, parameter = PARAMETER,
        p.value = PVAL, method = "Kruskal-Wallis rank sum test")
    return(RVAL)
}
```

a.

Write a pseudo code outlining what the function does.

The Kruskal-Wallis test is used in case we cannot make assumptions about the distribution of the data (non-parametric) and the assumptions about the analysis of variance are not met (reponse variable normally distributed, variance of population equal and responses for a given group are independent).

The pseudocode is outlined below:

1. Check the input:
   - x must have at least two groups with data
   - x and g must match in length (if not a list)
   - Convert g to a factor for grouping
2. Combine all observations and rank them from smallest to largest
   - Tied values get average ranks (R does that automatically)
3. Calculate rank sums and group sizes
   - For each group, count the number of values and sum their ranks
4. Compute the Kruskal-Wallis test statistic H
   - Use the formula:
     $$H = \frac{12}{N(N+1)} \sum_{j=1}^{k} \frac{R_j^2}{n_j} - 3(N+1)$$
   - Apply tie correction if needed:
     $$C = 1 - \frac{\sum(t_i^3 - t_i)}{N^3 - N}, \quad H_{\text{adj}} = \frac{H}{C}$$
5. Calculate the p-value
   - Use the chi-square distribution with df = k - 1:
     $$p = P(\chi^2_{k-1} > H_{\text{adj}})$$
6. Return the result
   - Test statistic, degrees of freedom, p-value, and method name

b.

For example data, call the function in two ways: once using `x` as a list and once using `x` as a vector with a corresponding `g` argument. Ensure that the two different function calls return the same thing by aligning the inputs.

```r
#calling kwtest with a list
list1 <- list(
  group1 = c(3, 5, 4),
  group2 = c(8, 9, 7),
  group3 = c(2, 1, 6)
)

result_list <- kwtest(list1)
result_list
```

```
## $statistic
## Kruskal-Wallis chi-squared
##                        5.6
##
## $parameter
## df
##  2
##
## $p.value
## [1] 0.06081006
##
## $method
## [1] "Kruskal-Wallis rank sum test"
```

```r
#calling it with vectors
x_vector <- c(3, 5, 4, 8, 9, 7, 2, 1, 6)
g_vector <- factor(c(1,1,1,2,2,2,3,3,3))

result_vector <- kwtest(x_vector, g_vector)
result_vector
```

```
## $statistic
## Kruskal-Wallis chi-squared
##                        5.6
##
## $parameter
## df
##  2
##
## $p.value
## [1] 0.06081006
##
## $method
## [1] "Kruskal-Wallis rank sum test"
```

c.

Make a faster version of `kwtest()` that only computes the Kruskal-Wallis **test statistic** when the input is a numeric variable `x` and a variable `g` which gives the group membership. You can try simplifying the function above or by coding from the mathematical definition (see https://en.wikipedia.org/wiki/Kruskal%E2%80%93Wallis_one-way_analysis_of_variance). This function should also perform some checks to ensure the correctness of the inputs (use `kwtest()` as inspiration).

```r
fast_kw <- function(x, g) {
  if (is.list(x)) {
    if (length(x) < 2L)
      stop("'x' must be a list with at least 2 elements")
    if (!missing(g))
      warning("'x' is a list, so ignoring argument 'g'")
    if (!all(sapply(x, is.numeric)))
      warning("some elements of 'x' are not numeric and will be coerced to numeric")
    k <- length(x)
    l <- lengths(x)
    if (any(l == 0L))
      stop("all groups must contain data")

    g <- factor(rep.int(seq_len(k), l))
    x <- unlist(x)
  }
  else {
    if (length(x) != length(g))
      stop("'x' and 'g' must have the same length")

    g <- factor(g)
    k <- nlevels(g)
    if (k < 2L)
      stop("all observations are in the same group")
  }

  n <- length(x)

  #get ranks
  r <- rank(x)

  #calculate group rank sums and sizes
  rank_sums <- tapply(r, g, sum)
  group_sizes <- tapply(r, g, length)

  #calculate H
  H <- (12/(n*(n+1))) * sum(rank_sums^2/group_sizes) - 3*(n+1)

  return(H)
}
```

d.

Consider the following scenario. You have samples available from multiple experiments $m = 1000$ where you collect the numerical values for the quantity of interest `x` and the group membership for n individuals. The first 20 individuals in each sample belong to group 1, the following 20 individuals in each sample belong to group 2, the last 10 individuals in each sample belong to group 3. Use the following code to simulate such a data structure:

```r
set.seed(1234)
m <- 1000 # number of repetitions
n <- 50   # number of individuals
X <- matrix(rt(m * n, df = 10), nrow = m)
grp <- rep(1:3, c(20, 20, 10))
```

Write a function which performs the Kruskal-Wallis test using the function `stats:::kruskal.test.default()` for m repeated experiments and returns a vector of m test statistics. The input of this function are a matrix `X` with m rows and n columns and a vector `g` which gives the grouping.

```
kw_multiple_tests <- function(X, g) {
  #check if input is correct
  if (!is.matrix(X)) stop("X must be a matrix")
  if (ncol(X) != length(g)) stop("number of columns in X must match length of g")
  if (length(unique(g)) < 2) stop("need at least 2 groups")

  #store results in the results vector
  results <- numeric(nrow(X))

  #loop over rows and create results
  for (i in 1:nrow(X)) {
    results[i] <- kruskal.test(X[i, ], g)$statistic
  }

  return(results)
}
```

e.

Write a function which performs the Kruskal-Wallis test using the function in point c. for m repeated experiments and returns a vector of m test statistics. The input of this function are a matrix `X` with m rows and n columns and a vector `g` which gives the grouping.

Basically the same thing as in 6.d), but using the original kwtest() function.

```
kwtest2 <- function(X, g) {
  #check if input is correct
  if (!is.matrix(X))
    stop("X must be a matrix")
  if (ncol(X) != length(g))
    stop("number of columns in X must match length of g")
  if (length(unique(g)) < 2)
    stop("need at least 2 groups")

  #store results in the results vector
  results <- numeric(nrow(X))

  #loop over rows and create results
  for (i in 1:nrow(X)) {
    results[i] <- kwtest(X[i, ], g)$statistic
  }

  return(results)
}
```

f.

Compare the performance of the two approaches using a benchmarking package on the data generated above. Comment on the results.

```
if (!require("microbenchmark")) install.packages("microbenchmark")
library(microbenchmark)

bench_results <- microbenchmark(
  original_kw = {
    sapply(1:nrow(X), function(i) kwtest(X[i,], grp)$statistic)
  },
  fast_kw = {
    sapply(1:nrow(X), function(i) fast_kw(X[i,], grp))
  },
  times = 3
)

bench_results
```

```
## Unit: milliseconds
##         expr       min      lq     mean    median        uq       max neval
##  original_kw 103.44136 104.428 105.8824 105.41473 107.10295 108.79116     3
##      fast_kw  37.17511  38.203  39.6225  39.23089  40.84619  42.46149     3
```

I am unsure, if I understood this part correctly, but I am comparing the original KW function with the fast KW function to see if there are differences in performance. Using R's benchmarking library, we can clearly see that the original_kw function, which was provided in the Assignment's description is a lot slower than the fast_kw. The mean runtime of the original_kw function is ~456 ms, while the mean runtime of the fast_kw is ~ 162 ms.

g.

Now consider vectorizing the function in point c. Compare this approach to the other two. Comment on the results.

```
fast_kw_vectorized <- function(x, g) {
  #check for matrix input
  if (is.matrix(x)) {
    if (missing(g)) stop("for matrix input, group vector 'g' must be provided")
    if (ncol(x) != length(g)) stop("number of columns in x must match length of g")

    #vectorization of ranks
    ranks <- t(apply(x, 1, rank))

    g <- as.factor(g)
    k <- nlevels(g)
    n <- ncol(x)

    group_sums <- matrix(0, nrow = nrow(x), ncol = k)
    group_counts <- tabulate(g)

    for (i in seq_len(k)) {
      group_cols <- which(g == levels(g)[i])
      group_sums[, i] <- rowSums(ranks[, group_cols, drop = FALSE])
    }
    H <- (12/(n*(n+1))) * rowSums(group_sums^2/group_counts) - 3*(n+1)
    return(H)
  }

  if (is.list(x)) {
    if (length(x) < 2L)
      stop("'x' must be a list with at least 2 elements")
    if (!missing(g))
      warning("'x' is a list, so ignoring argument 'g'")
    if (!all(sapply(x, is.numeric)))
      warning("some elements of 'x' are not numeric and will be coerced to numeric")

    k <- length(x)
    l <- lengths(x)
    if (any(l == 0L))
      stop("all groups must contain data")

    g <- factor(rep.int(seq_len(k), l))
    x <- unlist(x)
  } else {
    if (length(x) != length(g))
      stop("'x' and 'g' must have the same length")

    g <- factor(g)
    k <- nlevels(g)
    if (k < 2L)
      stop("all observations are in the same group")
  }

  n <- length(x)

  #get ranks
  r <- rank(x)

  #calculate group rank sums and sizes
  rank_sums <- tapply(r, g, sum)
  group_sizes <- tapply(r, g, length)

  #calculate H
  H <- (12/(n*(n+1))) * sum(rank_sums^2/group_sizes) - 3*(n+1)

  return(H)}
```

```
bench_results2 <- microbenchmark(
  original = {
    sapply(1:nrow(X), function(i) kwtest(X[i,], grp)$statistic)
  },
  fast = {
    sapply(1:nrow(X), function(i) fast_kw(X[i,], grp))
  },
  vectorized = {
    fast_kw_vectorized(X, grp)
  },
  times = 3)

bench_results2
```

```
## Unit: milliseconds
##        expr        min         lq       mean     median        uq       max
##    original 103.120699 103.745416 104.361810 104.370133 104.98237 105.59460
##        fast  36.618617  37.287409  38.114283  37.956201  38.86212  39.76803
##  vectorized   5.951191   6.612541   9.461912   7.273892  11.21727  15.16065
##  neval
##      3
##      3
##      3
```

Clearly, the vectorized function is the fastest of the three. This could be because it uses matrix operations instead of loops, avoids R's function call overhead, and processes all rows at once by computing ranks and group sums in bulk.