Case Study 1

AKSTA Statistical Computing

Tatzberger Jonas, Rasser Thomas, Grübling Alexander

03.04.2024

Exercises

1. Ratio of Fibonacci numbers

a.

b.

c.

2. Gamma function

a.

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

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

Answer

```r
rho <- function(n) {
  return (
    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?

Answer

```r
testcases <- c(1, 2, 3, 4, 5, 10, 50, 100, 500, 1000, 2000)
for (case in testcases) {
  print(paste("Rho(", case, ") = ", rho(case), sep=""))
}
```

```
## Warning in gamma((n - 1)/2): NaNs produced

## [1] "Rho(1) = NaN"

## Warning in gamma((n - 2)/2): NaNs produced

## [1] "Rho(2) = NaN"
## [1] "Rho(3) = 0.318309886183791"
## [1] "Rho(4) = 0.5"
## [1] "Rho(5) = 0.636619772367581"
## [1] "Rho(10) = 1.09375"
## [1] "Rho(50) = 2.74959606512371"
## [1] "Rho(100) = 3.93926528482005"
## [1] "Rho(500) = NaN"
## [1] "Rho(1000) = NaN"
## [1] "Rho(2000) = NaN"
```

This function does not work for 1 and 2, since gamma returns NaN for values less or equal to 0

```r
testcases <- 100:500
for (case in testcases) {
  if (is.nan(rho(case))) {
    print(paste("Rho breaks at ", case, sep=""))
    break
  }
}
```

```
## [1] "Rho breaks at 346"
```

If the number gets to big, the inbuilt our functions breaks, since it relies on the inbuilt gamma function, which returns Inf if the input is to big, which breaks our division

```r
testcases <- 1:500
for (case in testcases) {
  if (is.infinite(gamma(case))) {
    print(paste("Gamma returns Inf at ", case, sep=""))
    break
  }
}
```

```
## [1] "Gamma returns Inf at 172"
```

c.

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

Answer

The given function can be rewritten using logarithms as:

$$\log\left(\frac{\Gamma\left(\frac{n-1}{2}\right)}{\Gamma\left(\frac{1}{2}\right)\cdot\Gamma\left(\frac{n-2}{2}\right)}\right) = \log\Gamma\left(\frac{n-1}{2}\right) - \left(\log\Gamma\left(\frac{1}{2}\right) + \log\Gamma\left(\frac{n-2}{2}\right)\right)$$

which turns the multiplication into additions.

```r
rho_log <- function(n) {
  if (n <= 5) {
    return (gamma((n-1)/2) / (gamma(1/2) * gamma((n-2)/2)))
  } else {
    log_gamma_expr <- lgamma((n - 1) / 2) - (lgamma(1 / 2) + lgamma((n - 2) / 2))
    return (exp(log_gamma_expr))
  }
}
```

d.

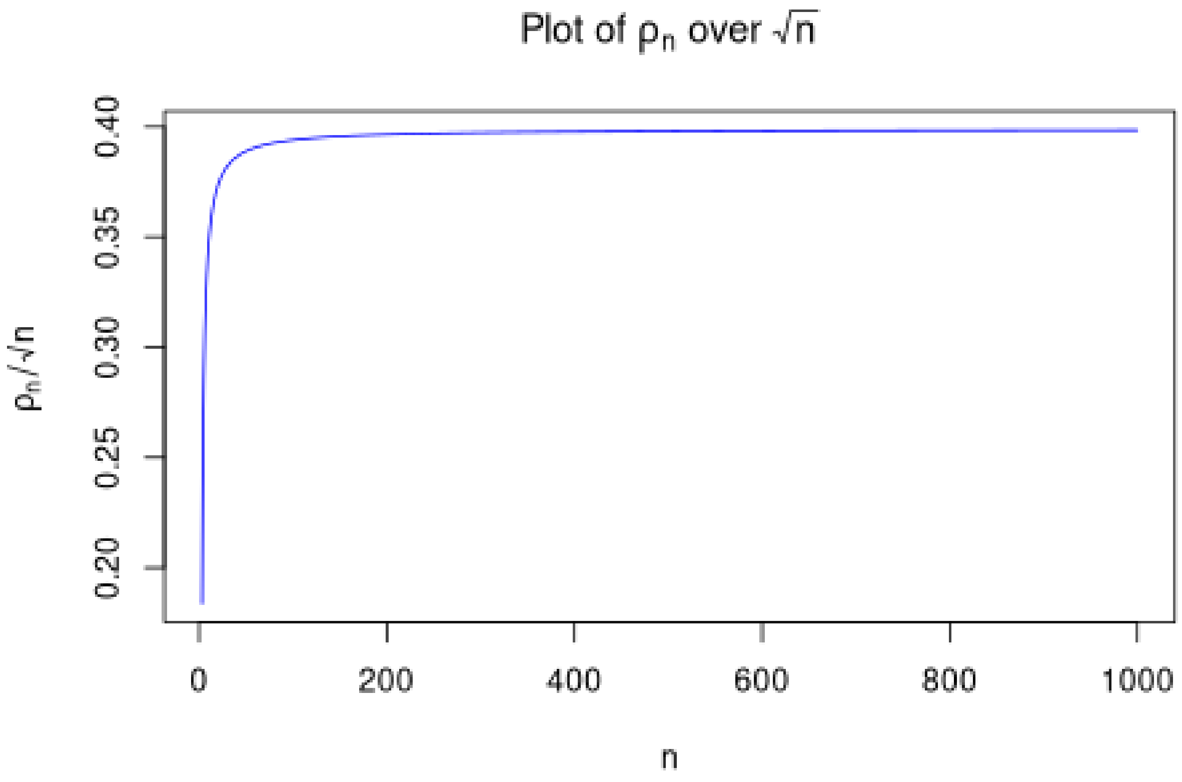
Plot $\rho_n/\sqrt{n}$ for different values of n. Can you guess the limit of $\rho_n/\sqrt{n}$ for $n \to \infty$?

Answer

```r
rhoOverSqrtN <- function(n) {
  if (!is.numeric(n)) {
    stop("n must be numeric.")
  }

  return (rho_log(n)/sqrt(n))
}

x <- 1:1000
y <- sapply(x, rhoOverSqrtN)
```

```
## Warning in gamma((n - 1)/2): NaNs produced

## Warning in gamma((n - 2)/2): NaNs produced
```

```r
# Filter out invalid values (NA, NaN, Inf)
valid_indices <- !is.na(y) & !is.nan(y) & !is.infinite(y)
x <- x[valid_indices]
y <- y[valid_indices]

plot(x, y, type = 'l', col = 'blue', xlab = "n", ylab = expression(rho[n] / sqrt(n)), main=expression("
```



```r
convergence <- tail(y)
for (c in convergence) {
  print(c)
}
```

```
## [1] 0.3984408
## [1] 0.3984413
## [1] 0.3984418
## [1] 0.3984423
## [1] 0.3984428
## [1] 0.3984433
```

The function approaches 3.984, which is approximately $\frac{1}{\sqrt{2\pi}}$

```r
print(paste("Approx. limit:   ", y[length(y)-1]))
```

```
## [1] "Approx. limit:    0.398442815762513"
```

```r
expr <- expression(1/sqrt(2*pi))
value <- eval(expr)
print(paste(expr, ": ", value))
```

```
## [1] "1/sqrt(2 * pi) :  0.398942280401433"
```

3. The golden ratio

Two positive numbers x and y with $x > y$ are said to be in the golden ratio if the ratio between the larger number and the smaller number is the same as the ratio between their sum and the larger number:

$$\frac{x}{y} = \frac{x+y}{x}$$

The golden ratio $\Phi = x/y$ can be computed as a solution to $\Phi^2 - \Phi - 1 = 0$ and is

$$\Phi = \frac{\sqrt{5}+1}{2}$$

and it satisfies the Fibonacci-like relationship:

$$\Phi^{n+1} = \Phi^n + \Phi^{n-1}$$

a.

Write an R function which computes Φ^{n+1} using the recursion above (go up to $n = 1000$) .

```r
phi_recursive <- function(n) {
  phi_0 <- (sqrt(5) - 1) / 2  # This is not used directly but initialized for completeness
  phi_1 <- (sqrt(5) + 1) / 2  # This is Phi
  phi_2 <- phi_1 + 1          # This is Phi^2, which equals Phi + 1 due to the quadratic equation

  # Base cases
  if (n == 0) {
    return(phi_1)
  } else if (n == 1) {
    return(phi_2)
  }

  # Iteratively compute Phi^n+1 for n > 1
  for (i in 2:n) {
    # Update the sequence values
    temp <- phi_2 + phi_1
    phi_1 <- phi_2
    phi_2 <- temp
  }

  return(phi_2)
}

testcases <- seq(0,4)
for (case in testcases) {
  print(paste("Phi^", case+1, ": ", phi_recursive(case), sep=""))
}
```

```
## [1] "Phi^1: 1.61803398874989"
## [1] "Phi^2: 2.61803398874989"
## [1] "Phi^3: 4.23606797749979"
## [1] "Phi^4: 6.85410196624968"
## [1] "Phi^5: 11.0901699437495"
```

```r
print(paste("Phi^1000: ", phi_recursive(999), sep=""))
```

```
## [1] "Phi^1000: 9.71941777359082e+208"
```

b.

Write a function which computes this Φ^{n+1} by simply using the power operator ^.

```r
phi_power <- function(n) {
  return (((sqrt(5) + 1) / 2)^(n+1))
}

testcases <- seq(0,4)
for (case in testcases) {
  print(paste("Phi^", case+1, ": ", phi_power(case), sep=""))
}
```

```
## [1] "Phi^1: 1.61803398874989"
## [1] "Phi^2: 2.61803398874989"
## [1] "Phi^3: 4.23606797749979"
## [1] "Phi^4: 6.85410196624969"
## [1] "Phi^5: 11.0901699437495"
```

```r
print(paste("Phi^1000: ", phi_power(999), sep=""))
```

```
## [1] "Phi^1000: 9.71941777359114e+208"
```

c.

Use once == and once all. equal to compare Φ^{n+1} obtained in a. vs the one obtained in b. for $n = 12, 60, 120, 300$. What do you observe? If there are any differences, what can be the reason?

```r
n_values <- c(12, 60, 120, 300)
comparison_results <- sapply(n_values, function(n) {
  recursive_computed <- phi_recursive(n)
  power_computed <- phi_power(n)

  # Using == for exact comparison (might not always be true due to floating-point precision)
  exact_match <- recursive_computed == power_computed

  # Using all.equal for a tolerance-based comparison
  tolerant_match <- all.equal(recursive_computed, power_computed)

  list(ExactMatch = exact_match, TolerantMatch = tolerant_match)
})

print(comparison_results)
```

```
##               [,1]  [,2]  [,3]  [,4]
## ExactMatch    FALSE FALSE FALSE FALSE
## TolerantMatch TRUE  TRUE  TRUE  TRUE
```

As written in the R documentation, the *all.equal* function uses a tolerance:

“Differences smaller than tolerance are not reported. The default value is close to 1.5e-8”

4. Game of craps

5. Readable and efficient code

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

cat("Step", 1, "\n")
```

```
## Step 1
```

```
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")
```

```
## Step 2
```

```
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")
```

```
## Step 3
```

```
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")
```

```
## Step 4
```

```
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
```

```
## [1] 1.0249956 0.9952113 1.0685500 1.0707264
```

a.

This code first takes samples of two normally distributed random variables.Then, multiple linear models are calculated, based on different portions of the input data. Afterwards, those models are used to predict that portion of the data, that has not been used to train the models. Finally, the mean squared error is appended to a list and printed out.

b.

While achieving its purpose, the code is not structured in an optimal way. First, there is a lot of repetition, which violates the DRY (Don't Repeat Yourself) Principle. To mitigate this, I would suggest refactoring by putting repeated code segments in a dedicated function. Then, the variable representing the mean squared errors is named "r". While this might not be an explicit problem, I would prefer to rename this to a more clear name.

c.

```r
getLinearModelMeanSquaredError <- function(data, testingRange) {
  fit <- lm(y ~ x, data = data[-testingRange,])
  prediction <- predict(fit, newdata = data[testingRange,])
  return(sqrt(mean((prediction - data[testingRange, "y"])^2)))
}

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

indices <- list(c(1:250), c(251:500), c(501:750), c(751:1000))
errors <- unlist(lapply(indices, function(i) getLinearModelMeanSquaredError(df, i)))
print(errors)
```

```
## [1] 1.0249956 0.9952113 1.0685500 1.0707264
```

6. Measuring and improving performance

```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.

```
Function kwtest(x, g, ...)
    If x is a list Then
        If length of x is less than 2 Then
            Stop with error "'x' must be a list with at least 2 elements"
        End If
        If g is not missing Then
            Issue warning "'x' is a list, so ignoring argument 'g'"
        End If
        If not all elements of x are numeric Then
            Issue warning "some elements of 'x' are not numeric and will be coerced to numeric"
        End If
        Set k to length of x
        Set l to lengths of x
        If any element of l is 0 Then
            Stop with error "all groups must contain data"
        End If
        Set g to factor with levels as sequence from 1 to k, repeated l times
        Unlist x
    Else
        If length of x is not equal to length of g Then
            Stop with error "'x' and 'g' must have the same length"
        End If
        Set g to factor of g
        Set k to number of levels in g
        If k is less than 2 Then
            Stop with error "all observations are in the same group"
        End If
    End If
    Set n to length of x
    If n is less than 2 Then
        Stop with error "not enough observations"
    End If
    Rank x and store in r
    Create a table of frequencies of x and store in TIES
    Calculate STATISTIC as sum of squares of ranks in each group divided by the number of ranks in each
    Adjust STATISTIC based on the formula provided
    Set PARAMETER to k - 1
    Calculate PVAL as the p-value of the chi-squared distribution with PARAMETER degrees of freedom and
    Set names of STATISTIC and PARAMETER to "Kruskal-Wallis chi-squared" and "df" respectively
    Create a list RVAL with statistic, parameter, p.value, and method
    Return RVAL
End Function
```

b.

```r
data("PlantGrowth")
group1 <- PlantGrowth$weight[PlantGrowth$group == "ctrl"]
group2 <- PlantGrowth$weight[PlantGrowth$group == "trt1"]
group3 <- PlantGrowth$weight[PlantGrowth$group == "trt2"]

result_list <- kwtest(list(group1, group2, group3))
print(result_list)
```

```
## $statistic
## Kruskal-Wallis chi-squared
##                   7.988229
##
## $parameter
## df
##  2
##
## $p.value
## [1] 0.01842376
##
## $method
## [1] "Kruskal-Wallis rank sum test"
```

```r
x_vector <- PlantGrowth$weight
g_vector <- factor(PlantGrowth$group)

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

```
## $statistic
## Kruskal-Wallis chi-squared
##                   7.988229
##
```

```
## $parameter
## df
##  2
##
## $p.value
## [1] 0.01842376
##
## $method
## [1] "Kruskal-Wallis rank sum test"
```

c.

```r
kwtest_fast <- function(x, g) {
  if (!is.numeric(x) || !is.numeric(g) || length(x) != length(g)) {
    stop("'x' and 'g' must be numeric vectors of the same length")
  }
  if (length(unique(g)) < 2) {
    stop("At least two unique groups are required for Kruskal-Wallis test")
  }
  ranks <- rank(x)
  group_sums <- tapply(ranks, g, sum)
  group_sizes <- tapply(ranks, g, length)

  n <- length(x)
  k <- length(unique(g))
  H <- (12 / (n * (n + 1))) * sum(group_sums^2 / group_sizes) - 3 * (n + 1)

  df <- k - 1

  p_value <- pchisq(H, df, lower.tail = FALSE)

  result <- list(
    statistic = H,
    parameter = df,
    p.value = p_value,
    method = "Kruskal-Wallis rank sum test"
  )

  return(result)
}
```

d.

```r
perform_kruskal_wallis_tests <- function(X, g, fun) {
  split_X <- split(X, seq_len(nrow(X)))

  test_results <- lapply(split_X, function(experiment_data) {
    fun(experiment_data, g)
  })
  test_statistics <- sapply(test_results, function(result) result$statistic)
  return(unname(test_statistics))
}

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))

test_statistics <- perform_kruskal_wallis_tests(X, grp, stats:::kruskal.test.default)
head(test_statistics)
```

```
## [1] 3.8463529 0.5463529 3.3352941 1.8783529 1.5087059 2.6771765
```

e.

```
test_statistics <- perform_kruskal_wallis_tests(X, grp, kwtest_fast)
head(test_statistics)
```

```
## [1] 3.8463529 0.5463529 3.3352941 1.8783529 1.5087059 2.6771765
```

f.

```
library(bench)
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))
mark(perform_kruskal_wallis_tests(X, grp, stats:::kruskal.test.default), perform_kruskal_wallis_tests(X
```

```
## Warning: Some expressions had a GC in every iteration; so filtering is
## disabled.
```

```
## # A tibble: 2 x 6
##   expression                                  min median `itr/sec` mem_alloc `gc/sec`
##   <bch:expr>                               <bch> <bch:>     <dbl> <bch:byt>    <dbl>
## 1 perform_kruskal_wallis_tests(X, grp~ 551ms  551ms      1.81   19.82MB     5.44
## 2 perform_kruskal_wallis_tests(X, grp~ 178ms  178ms      5.56    8.58MB    11.1
```

g.

```
kwtest_vectorized <- function(x, g) {
  if (!is.numeric(x) || !is.numeric(g) || length(x) != length(g)) {
    stop("'x' and 'g' must be numeric vectors of the same length")
```

```r
  }
  if (length(unique(g)) < 2) {
    stop("At least two unique groups are required for Kruskal-Wallis test")
  }
  ranks <- rank(x)

  n <- length(x)
  k <- length(unique(g))
  H <- (12 / (n * (n + 1))) * sum(tapply(ranks, g, sum)^2 / tapply(ranks, g, length)) - 3 * (n + 1)

  df <- k - 1

  p_value <- pchisq(H, df, lower.tail = FALSE)

  result <- list(
    statistic = H,
    parameter = df,
    p.value = p_value,
    method = "Kruskal-Wallis rank sum test"
  )

  return(result)
}

library(bench)
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))
mark(
  perform_kruskal_wallis_tests(X, grp, stats:::kruskal.test.default),
  perform_kruskal_wallis_tests(X, grp, kwtest_fast),
  perform_kruskal_wallis_tests(X, grp, kwtest_vectorized)
  )
```

```
## Warning: Some expressions had a GC in every iteration; so filtering is
## disabled.
```

```
## # A tibble: 3 x 6
##   expression                            min median `itr/sec` mem_alloc `gc/sec`
##   <bch:expr>                          <bch> <bch:>     <dbl> <bch:byt>    <dbl>
## 1 perform_kruskal_wallis_tests(X, grp~ 578ms  578ms      1.73   19.82MB     5.19
## 2 perform_kruskal_wallis_tests(X, grp~ 179ms  184ms      5.43    8.58MB     7.24
## 3 perform_kruskal_wallis_tests(X, grp~ 189ms  217ms      4.74    8.74MB     7.90
```