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

```
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

```
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 wurden erzeugt

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

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

## [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

```
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

```
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.

```
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.

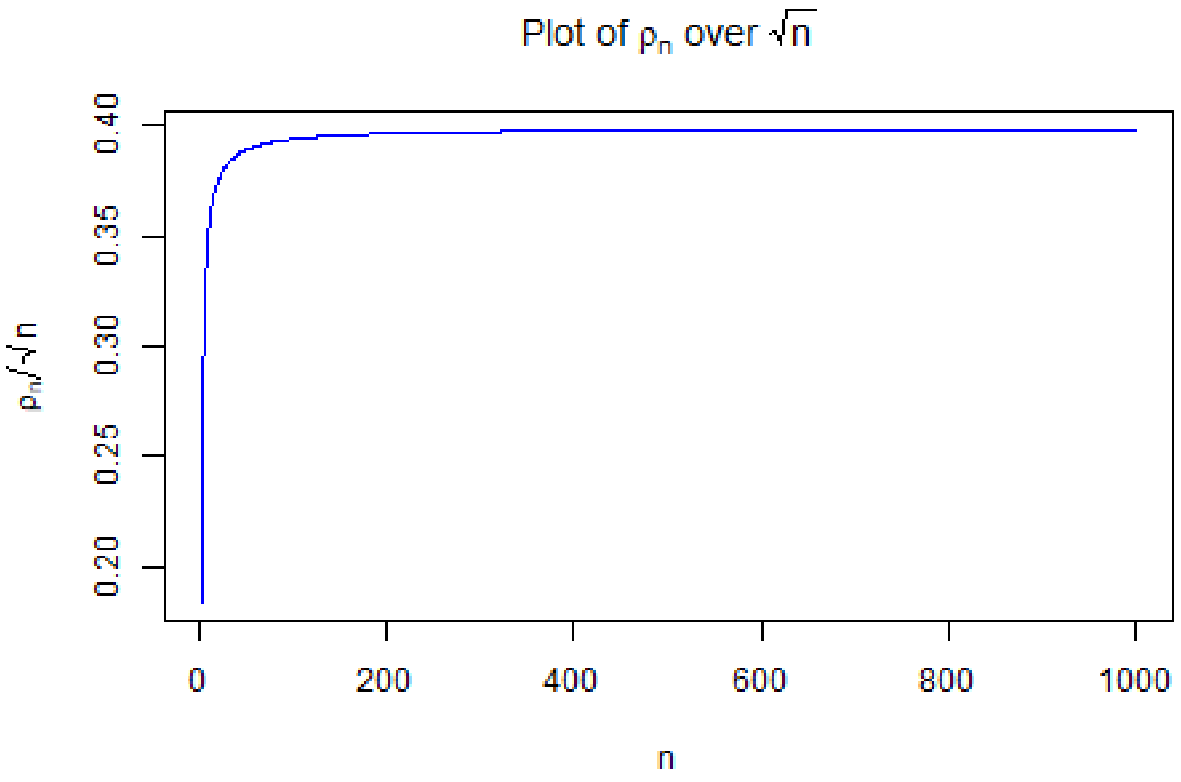
Plot $\rho_n/\sqrt{n}$ for different values of n. Can you guess the limit of $\rho_n/\sqrt{n}$ for $n \to \infty$?

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 wurden erzeugt

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

```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 $\frac{1}{\sqrt{2\pi}}$

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

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

```
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 $n = 1000$) .

```
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"
```

```
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 ^.

```
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"
```

```
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?

```
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

a.

b.

c.

6. Measuring and improving performance

a.

b.

c.

d.

e.

f.

g.