Solutions to Assignment 2: R intermediate

Dan McGlinn

2025-01-25

Examine the following for loop, and then complete the exercises

```r
data(iris)
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```r
sp_ids <- unique(iris$Species)

output <- matrix(0, nrow=length(sp_ids), ncol=ncol(iris)-1)
rownames(output) <- sp_ids
colnames(output) <- names(iris[ , -ncol(iris)])

for(i in seq_along(sp_ids)) {
    iris_sp <- subset(iris, subset=Species == sp_ids[i], select=-Species)
    for(j in 1:(ncol(iris_sp))) {
        x <- 0
        y <- 0
        if (nrow(iris_sp) > 0) {
            for(k in 1:nrow(iris_sp)) {
                x <- x + iris_sp[k, j]
                y <- y + 1
            }
            output[i, j] <- x / y
        }
    }
}
output
```

```
##            Sepal.Length Sepal.Width Petal.Length Petal.Width
## setosa            5.006       3.428        1.462       0.246
## versicolor        5.936       2.770        4.260       1.326
## virginica         6.588       2.974        5.552       2.026
```

Excercises

Iris loops

1. Describe the values stored in the object `output`. In other words what did the loops create?

These values are averages of the traits of each species.

2. Describe using pseudo-code how `output` was calculated, for example,

```
#loop through species ids
#    subset iris down to only rows associated with a particular species
#   loop through columns (i.e., species traits)
#       if their are records associated with that column then
#           loop through each observation
#               sum across the observations
#               count the number of observations
#           compute the mean across the observations
```

3. The variables in the loop were named so as to be vague. How can the objects `output`, `x`, and `y` could be renamed such that it is clearer what is occurring in the loop.

```r
# the simplest change here it to rename the vague objects
sp_mean <- matrix(0, nrow=length(sp_ids), ncol=ncol(iris)-1)
rownames(sp_mean) <- sp_ids
colnames(sp_mean) <- names(iris[ , -ncol(iris)])

for(i in seq_along(sp_ids)) {
    iris_sp <- subset(iris, subset=Species == sp_ids[i], select=-Species)
    for(j in 1:(ncol(iris_sp))) {
        trait_sum <- 0
        num_records <- 0
        if (nrow(iris_sp) > 0) {
            for(k in 1:nrow(iris_sp)) {
                trait_sum <- trait_sum + iris_sp[k, j]
                num_records <- num_records + 1
            }
            sp_mean[i, j] <- trait_sum / num_records
        }
    }
}
```

The loop above is much easier to read and understand by not using vague names.

4. It is possible to accomplish the same task using fewer lines of code? Please suggest one other way to calculate `output` that decreases the number of loops by 1.

```r
# R has a function to compute means so we can drop quite a few lines using that
sp_mean <- matrix(0, nrow=length(sp_ids), ncol=ncol(iris)-1)
rownames(sp_mean) <- sp_ids
colnames(sp_mean) <- names(iris[ , -ncol(iris)])
```

```r
for(i in seq_along(sp_ids)) {
    iris_sp <- subset(iris, subset=Species == sp_ids[i], select=-Species)
    for(j in 1:(ncol(iris_sp))) {
        sp_mean[i, j] <- mean(iris_sp[ , j])
    }
}
```

By using a function here such as mean we've made our code more readable and less prone to error

```r
# here are two other ways to simplify the code even further that both
# accomplish the exact same task

# approach 1
aggregate(subset(iris, select = -Species), list(Species = iris$Species), mean)
```

```
##      Species Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1     setosa        5.006       3.428        1.462       0.246
## 2 versicolor        5.936       2.770        4.260       1.326
## 3  virginica        6.588       2.974        5.552       2.026
```

```r
# approach 2 using aggregate's function specification
aggregate(cbind(Sepal.Length, Sepal.Width, Petal.Length, Petal.Width) ~ Species,
          data = iris, mean)
```

```
##      Species Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1     setosa        5.006       3.428        1.462       0.246
## 2 versicolor        5.936       2.770        4.260       1.326
## 3  virginica        6.588       2.974        5.552       2.026
```

```r
# approach 3 using dplyr, note the dplyr package must already be installed.
require(dplyr)
```

```
## Loading required package: dplyr
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```r
iris %>%
  group_by(Species) %>%
  summarize_all(mean)
```

```
## # A tibble: 3 x 5
##   Species    Sepal.Length Sepal.Width Petal.Length Petal.Width
##   <fct>             <dbl>       <dbl>        <dbl>       <dbl>
## 1 setosa             5.01        3.43         1.46       0.246
## 2 versicolor         5.94        2.77         4.26       1.33
## 3 virginica          6.59        2.97         5.55       2.03
```

These three approaches are pretty similar but folks differ on which one they prefer. So argue that the dplyr approach is the most 'readable', but I find that I always have to google how to specify the arguments and functions for dplyr (maybe that's just me though).

Sum of a sequence

5. You have a vector `x` with the numbers 1:10. Write a for loop that will produce a vector `y` that contains the sum of `x` up to that index of `x`. So for example the elements of `x` are 1, 2, 3, and so on and the elements of `y` would be 1, 3, 6, and so on.

```
x <- 1:10
y <-  NULL
for(i in 1:length(x)) {
    y[i] <- sum(x[1:i])
}
y
```

```
##  [1]  1  3  6 10 15 21 28 36 45 55
```

```
# alternatively we could use an sapply function
y <-  sapply(1:length(x), function(i) sum(x[1:i]))
y
```

```
##  [1]  1  3  6 10 15 21 28 36 45 55
```

```
# one other alternative that some students may have realized is to use
# the R function cumsum - definitely the easiest cheater solution ;P
?cumsum
y <- cumsum(x)
y
```

```
##  [1]  1  3  6 10 15 21 28 36 45 55
```

6. Modify your for loop so that if the sum is greater than 10 the value of `y` is set to NA

```
y <-  NULL
for(i in 1:length(x)) {
    y[i] <-  sum(x[1:i])
    if (y[i] > 10) {
        y[i] <-  NA
    }
}
y
```

```
##  [1]  1  3  6 10 NA NA NA NA NA NA
```

```
# alternatively although much more difficult to understand we could use
y <- sapply(1:length(x), function(i) ifelse(sum(x[1:i]) > 10, NA, sum(x[1:i])))
y
```

```
##  [1]  1  3  6 10 NA NA NA NA NA NA
```

```
# I definitely prefer the loop approach in this context because of readability
# the sapply approach will be no faster

# if you used the cumsum() approach for #5 then you would want to use the
# ifelse() function to replace those values that are greater than 10 with NA
y <- cumsum(x)
y <- ifelse(y > 10, NA, y)
y
```

```
##  [1]  1  3  6 10 NA NA NA NA NA NA
```

7. Place your for loop into a function that accepts as its argument any vector of arbitrary length and it will return y.

```
cum_sum_cutoff <- function(x, cutoff=10) {
    # this function computes a cumulative sum along a numeric vector up to a cutoff
    # arguments
    # x: a numeric vector
    # cutoff: a number above which sums are set to NA, defaults to 10.
    if (!is.vector(x))
        stop('x must be a vector')
    if (!is.numeric(x))
        stop('x must be numeric')
    y <- NULL
    for(i in 1:length(x)) {
        y[i] <- sum(x[1:i])
        if (y[i] > cutoff) {
            y[i] <- NA
        }
    }
    return(y)
}

cum_sum_cutoff(1:10)
```

```
##  [1]  1  3  6 10 NA NA NA NA NA NA
```

```
cum_sum_cutoff(1:10, 50)
```

```
##  [1]  1  3  6 10 15 21 28 36 45 NA
```

Notice above I have defined a new variable called `cutoff` which I use to vary what the cutoff of the cumulative sum is. Also notice that I used a fairly informative function name. This name is pushing the upper limits of how long an object name to shoot for. Text completion helps to deal with cumbersome names.

(Optional)Fibonacci numbers and Golden ratio

8. Fibonacci numbers are a sequence in which a given number is the sum of the preceding two numbers. So starting at 0 and 1 the sequence would be

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, . . .

Write and apply a simple R function that can accomplish this task with a for loop. Then write a function that computes the ratio of each sequential pair of Fibonacci numbers. Do they asymptotically approach the golden ratio (1 + sqrt(5)) / 2 = 1.618034 ?

```r
# first I define several helper functions
#' Add a value to the end of a numeric vector that is equal to the sum of the
#' last two values in the vector
#' @param x a numeric vector that has at least a length of 2
#' @returns a vector that is one longer than x which the last value a sum of
#' previous two values.
sum2c <- function(x) {
    return(c(x, x[length(x)] + x[length(x)-1]))
}

#' get a certain number of Fibonacci numbers
#' @param x a starting vector of numbers
#' @param depth how many Fibonacci numbers to mine
#' @returns a vector of Fibonacci numbers starting with x.
get_fib <- function(x, depth) {
    output <- x
    for(i in 1:depth)
       output <- sum2c(output)
    return(output)
}

#' compute the ratio of each sequential value in a vector
#' @param a vector of numbers
#' @returns a vector of ratios
get_ratios <- function(x) {
  return(x[-1] / x[-length(x)])
}

sum2c(0:1)
```

```
## [1] 0 1 1
```

```r
sum2c(3:4)
```

```
## [1] 3 4 7
```

```r
sum2c(c(0, 1, 1, 2, 3, 5))
```

```
## [1] 0 1 1 2 3 5 8
```

```r
# a recursive usage of sum2c
sum2c(sum2c(sum2c(sum2c(0:1))))
```

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

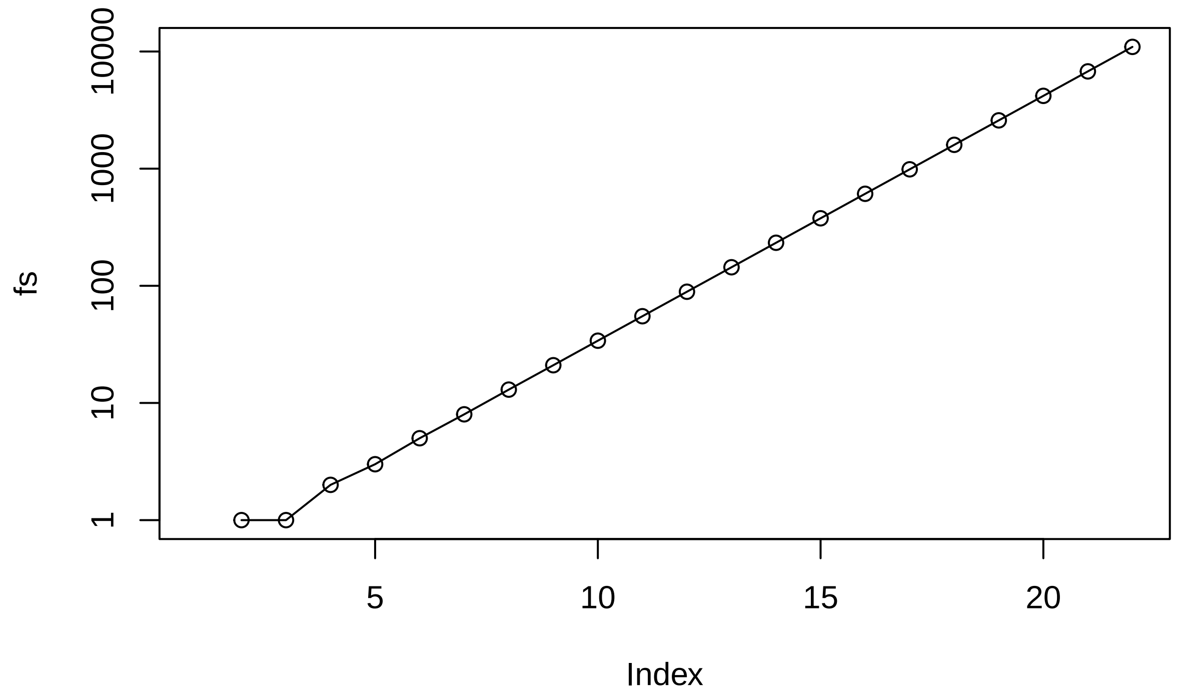
```r
fs <- get_fib(0:1, 20)

get_ratios(fs)
```

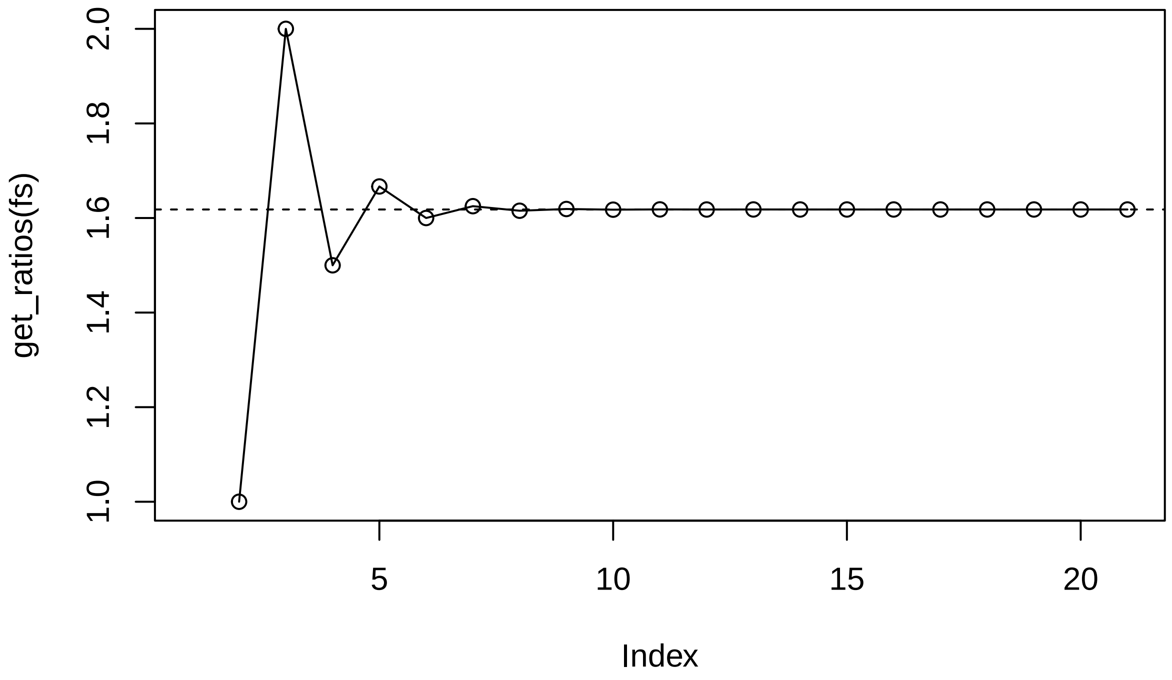
```
##  [1]      Inf 1.000000 2.000000 1.500000 1.666667 1.600000 1.625000 1.615385
##  [9] 1.619048 1.617647 1.618182 1.617978 1.618056 1.618026 1.618037 1.618033
## [17] 1.618034 1.618034 1.618034 1.618034 1.618034
```

```r
# the Fibonacci numbers follow an exponential function which is linear
# when the y-axis is log transformed.
plot(fs, type = 'o', log='y')
```

```
## Warning in xy.coords(x, y, xlabel, ylabel, log): 1 y value <= 0 omitted from
## logarithmic plot
```



```r
# notice how the ratios approach the true golden ratio
plot(get_ratios(fs), type ='o')
abline(h = (1 + sqrt(5)) / 2, lty=2) # golden ratio
```



The golden ratio pops up all throughout nature and art, and it can provide a useful design tool when developing aesthetically pleasing layout (Gendelman 2015).

Citation: Gendelman, V. 2015. How to Use the Golden Ratio to Create Gorgeous Graphic Designs https://www.companyfolders.com/blog/golden-ratio-design-examples

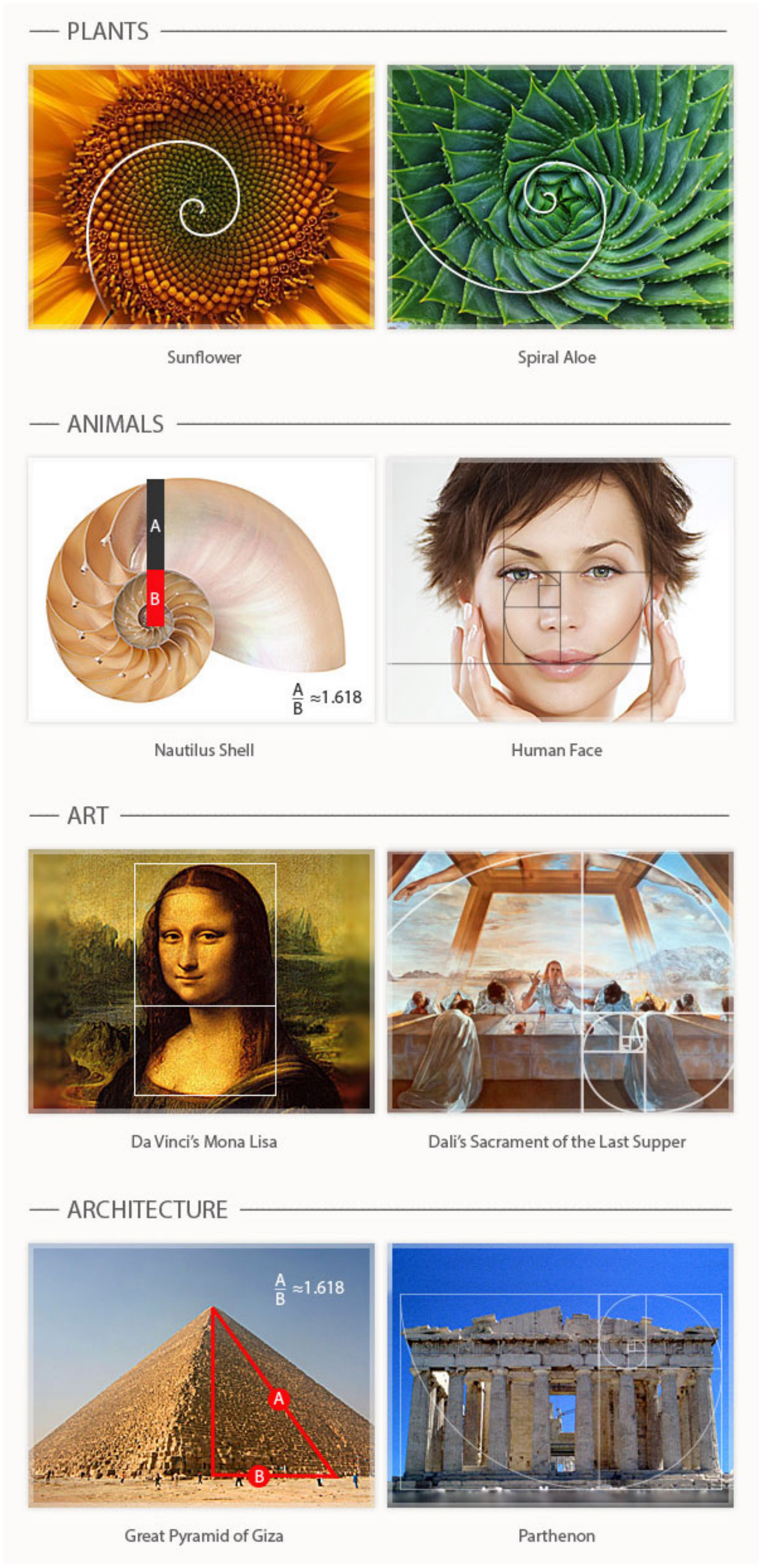


Figure 1: Examples of the golden ratio (Gendelman 2015)