Exercises A.4-A.10 solutions

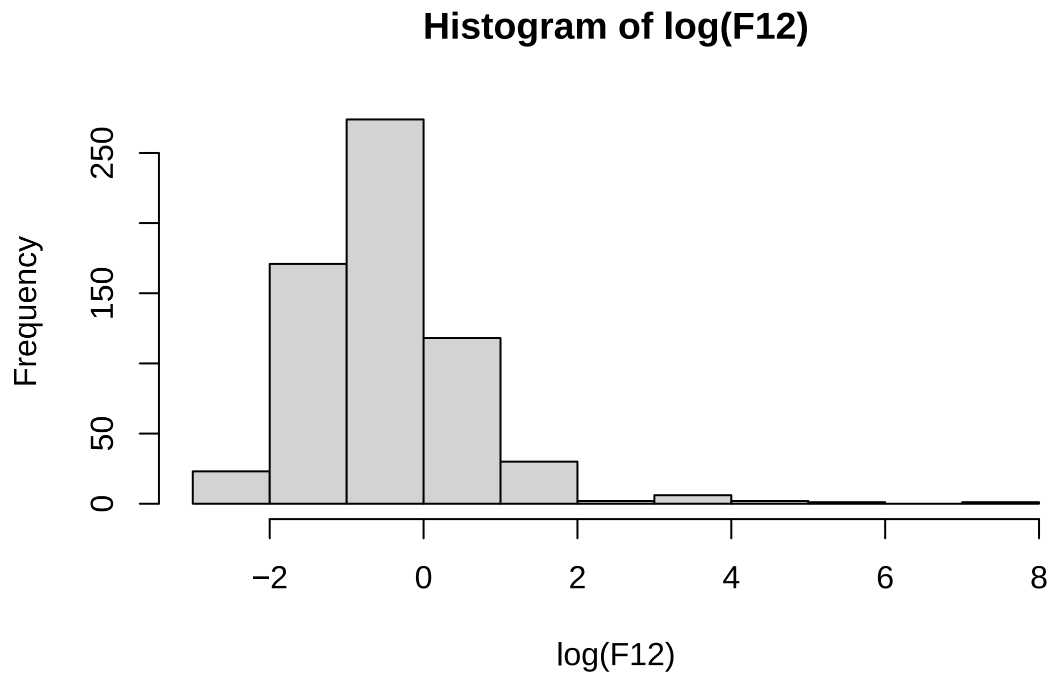
Niels Richard Hansen

September 3, 2020

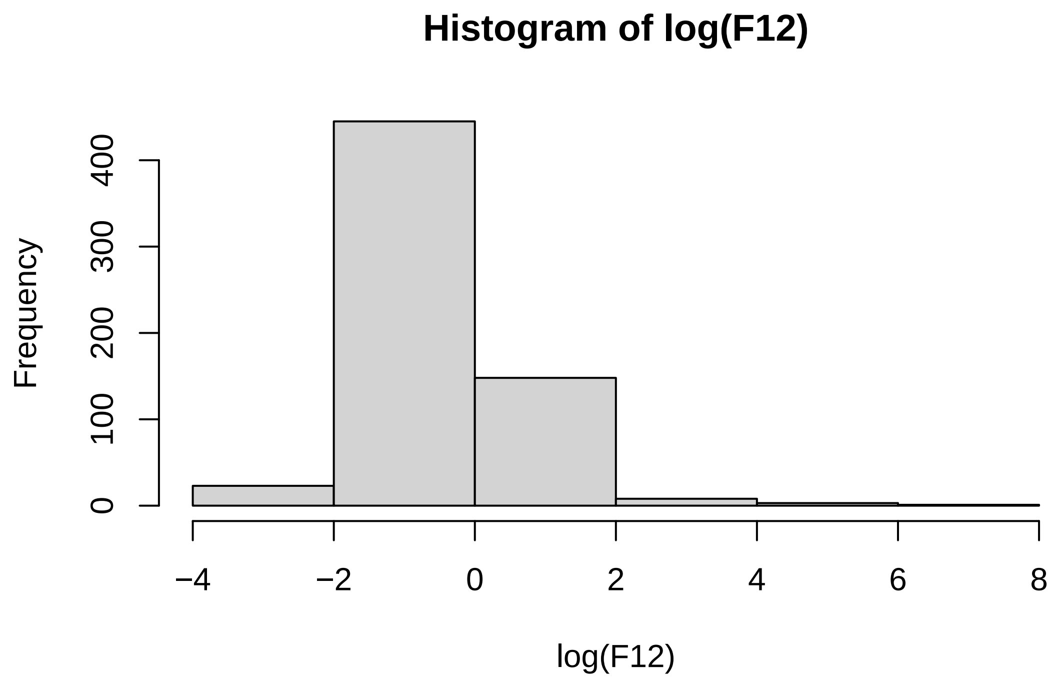
Contents

Exercise A.4

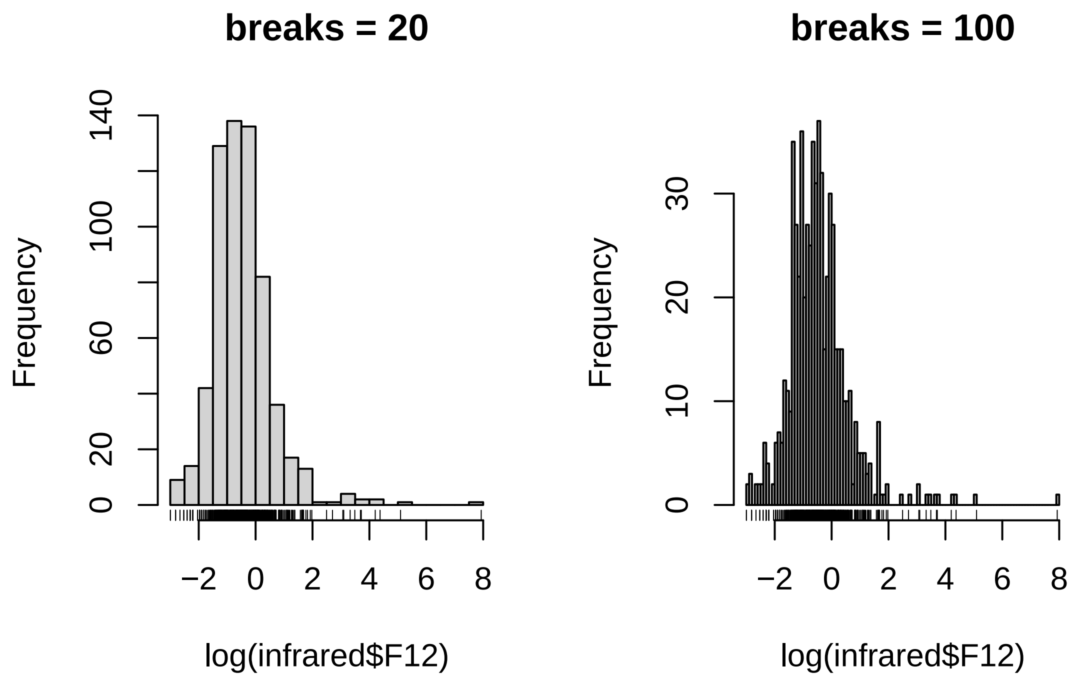
```r
infrared <- read.table("../data/infrared.txt", header = TRUE)
F12 <- infrared$F12
hist(log(F12))
```



```r
hist(log(F12), breaks = 5)
```



```r
hist(log(infrared$F12), 20, main = "breaks = 20")
rug(log(infrared$F12))
hist(log(infrared$F12), 100, main = "breaks = 100")
rug(log(infrared$F12))
```



We observe:

- The number of cells changes the impression of the distribution.
- The default gives relatively few and large cells.

- The default uses *Sturges' formula*: the number of cells for n observations is

$$\lceil \log_2(n) + 1 \rceil.$$

Exercise A.5

```
my_breaks <- function(x, h = 5) {
  x <- sort(x)
  ux <- unique(x)
  i <- seq(from = 1, to = length(ux), by = h)
  ux[i]
}
```

```
my_breaks(c(1, 3, 2, 5, 10, 11, 1, 1, 3), 2)
```

```
[1]  1  3 10
```

Note, we missed the largest value 11 in `x`.

A correction

```
my_breaks <- function(x, h = 5) {
  x <- sort(x)
  ux <- unique(x)
  i <- seq(from = 1, to = length(ux), by = h)
  if (i[length(i)] < length(ux))      # If last index not length(ux)
    i[length(i) + 1] <- length(ux)    # append that index
  ux[i]
}
```

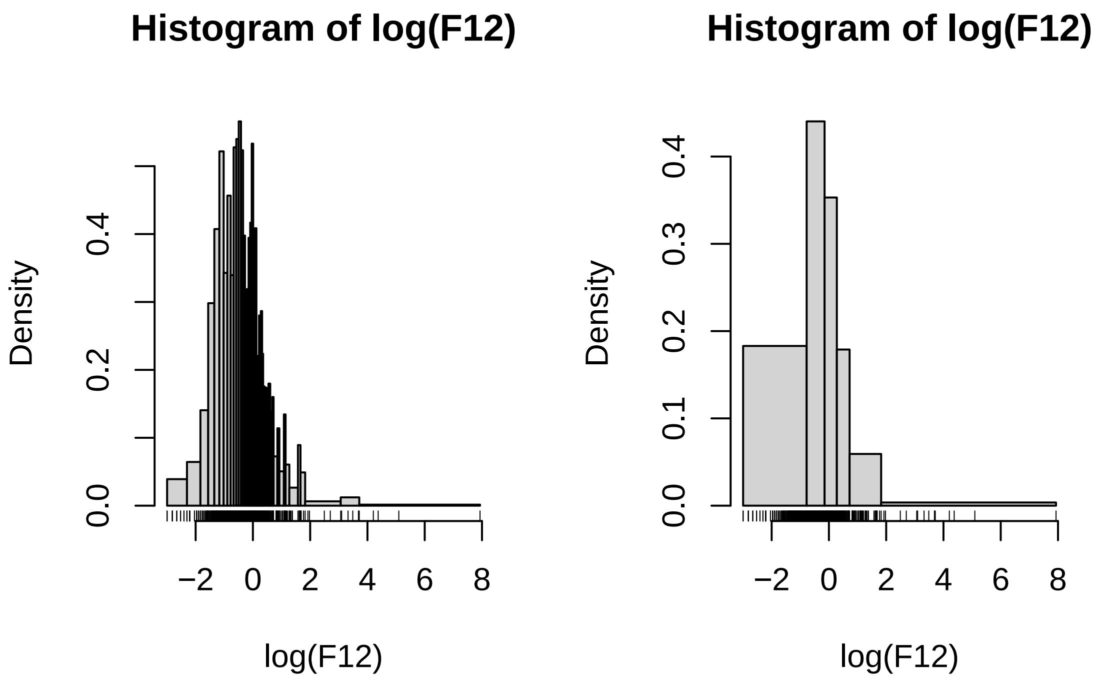
```
my_breaks(c(1, 3, 2, 5, 10, 11, 1, 1, 3), 2)
```

```
[1]  1  3 10 11
```

Now it works as expected, and we can use the function with `hist()`

```
hist(log(F12), my_breaks)
rug(log(F12))

hist(log(F12), function(x) my_breaks(x, 40))
rug(log(F12))
```



Exercise A.6

First version

```
my_breaks <- function(x, h = 5) {
  x <- sort(x)
  breaks <- xb <- x[1]
  k <- 1
  for(i in seq_along(x)[-1]) {
    if (k < h) {
      k <- k + 1
    } else {
      if (xb < x[i - 1] && x[i - 1] < x[i]) {
        xb <- x[i - 1]
        breaks <- c(breaks, xb)
        k <- 1
      }
    }
  }
  # A last breakpoint is appended to ensure coverage of the range of x
  breaks[length(breaks) + 1] <- x[length(x)]
  breaks
}
```

Testing

```
my_breaks(1:11, 1)  # Should be 1:11
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11
```

```r
my_breaks(1:2, 1)    # Should be 1, 2
```

```
[1] 1 2
```

```r
my_breaks(1:10, 5)  # Should be 1, 5, 10
```

```
[1]  1  5 10
```

```r
my_breaks(1:11)      # Should be 1, 5, 11
```

```
[1]  1  5 10 11
```

Last interval $(10, 11]$ doesn't have five elements! The way that the last breakpoint is appended doesn't ensure that the number of elements in each interval is at least `h`.

Second version

```r
my_breaks <- function(x, h = 5) {
  x <- sort(x)
  breaks <- xb <- x[1]
  k <- 1
  for(i in seq_along(x)[-1]) {
    if (k < h) {
      k <- k + 1
    } else {
      if (xb < x[i - 1] && x[i - 1] < x[i]) {
        xb <- x[i - 1]
        breaks <- c(breaks, xb)
        k <- 1
      }
    }
  }
  # If there are at least h (k == h) elements after the last
  # breakpoint (or if length(x) < h) a last breakpoint is appended
  # Otherwise the last breakpoint is changed to be the largest element of x
  last <- length(breaks)
  if (k == min(h, length(x) - 1))
    last <- last + 1
  breaks[last] <- x[length(x)]
  breaks
}
```

Testing again

```r
my_breaks(1:11, 1)  # Should be 1:11
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11
```

```r
my_breaks(1:2, 1)    # Should be 1, 2
```

```
[1] 1 2
```

```r
my_breaks(1:10, 5)  # Should be 1, 5, 10
```

```
[1]  1  5 10
```

```
my_breaks(1:11)     # Should be 1, 5, 11
```

```
[1]  1  5 11
```

```
# A test with unsorted data with ties
# Should be 1, 2, 3, 11
my_breaks(c(1, 3, 2, 5, 10, 11, 1, 1, 3), 2)
```

```
[1]  1  2  3 11
```

```
# A test with non-integer data with ties
test <- c(3, 1, 4.2, 3, 2, 4.2, 3, 1, 2, 4, 5, 3, 3.1, 3, 4.3)
# Should be A.0, 2.0, 3.0, 4.0, 4.2, 10
my_breaks(test, 2)
```

```
[1] 1.0 2.0 3.0 4.0 4.2 5.0
```

```
# Should be 1, 2, 3, 5 (why?)
my_breaks(test, 3)
```

```
[1] 1 2 3 5
```

```
# Sort data to compute breakpoints by hand
sort(test)
```

```
 [1] 1.0 1.0 2.0 2.0 3.0 3.0 3.0 3.0 3.0 3.1 4.0 4.2 4.2 4.3 5.0
```

Testing on data

```
hh <- seq(1, 100, 1)
breaks <- lapply(hh, function(h) my_breaks(log(F12), h))
counts <- lapply(breaks,
                 function(b) hist(log(F12), b, plot = FALSE)$counts)
any(sapply(breaks, function(x) any(duplicated(x))))
```

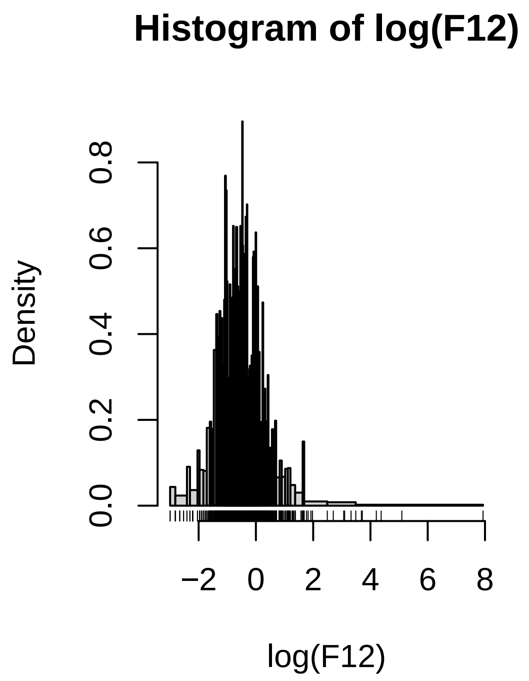
```
[1] FALSE
```

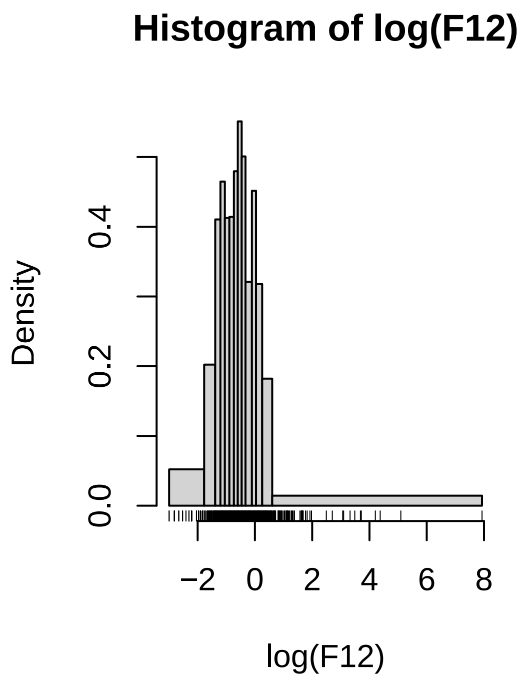
```
all(sapply(seq_along(hh), function(i) all(counts[[i]] >= hh[i])))
```

```
[1] TRUE
```

First there is a test for duplicated breaks, and second there is a test for the number of observations in each interval to be larger than h.

```
hist(log(F12), my_breaks)
rug(log(F12))
hist(log(F12), function(x) my_breaks(x, 40))
rug(log(F12))
```



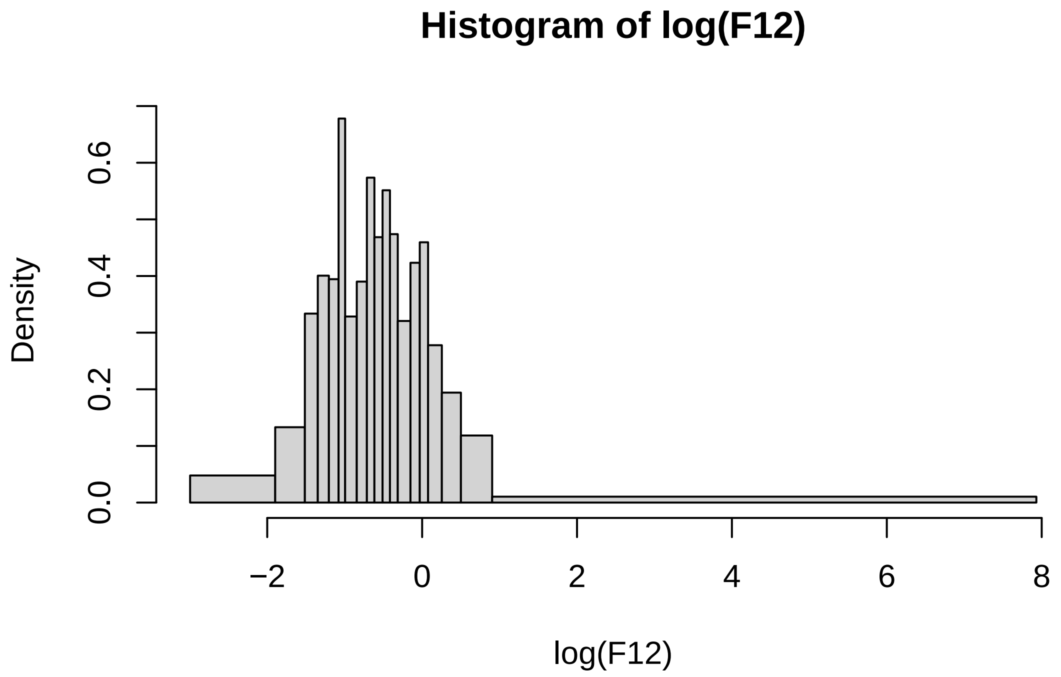


Exercise A.7

We define a `my_hist` function as requested.

```r
my_hist <- function(h, ...)
  hist(log(F12), function(x) my_breaks(x, h), ...)
```

```r
my_hist(30)
```



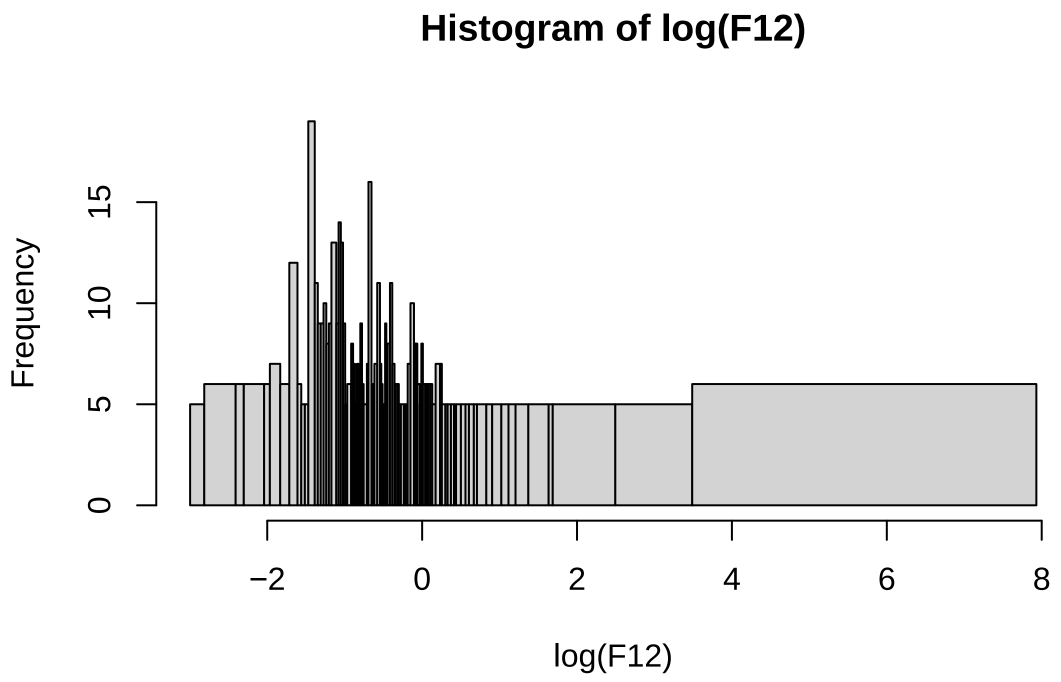
Testing

```r
my_hist()
```

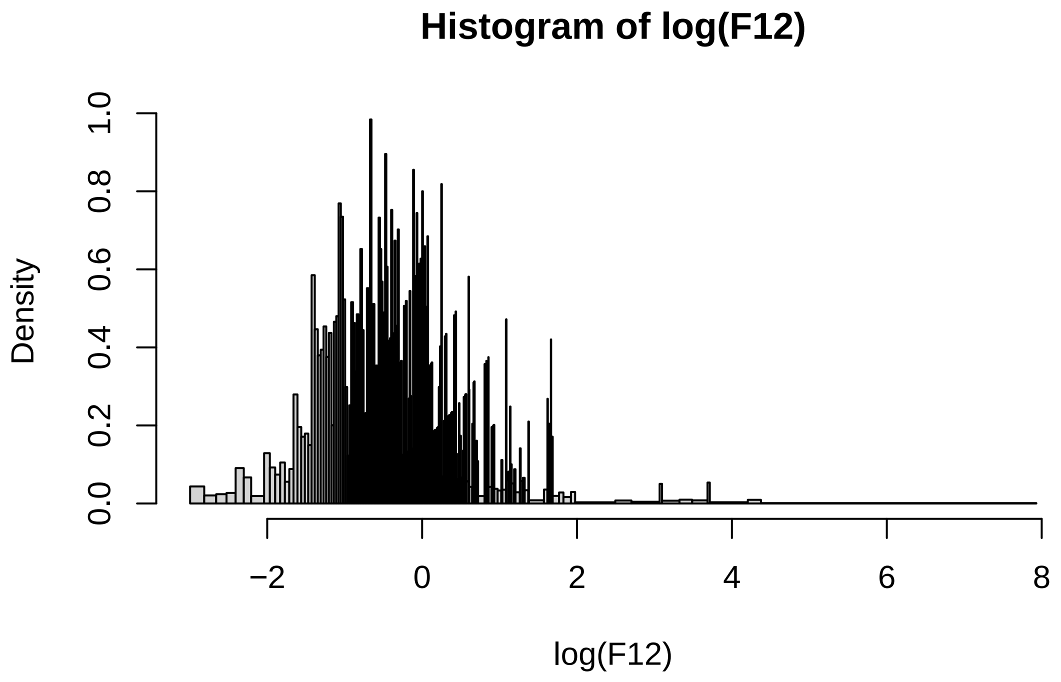
```
Error in my_breaks(x, h): argument "h" is missing, with no default
```

```r
my_hist(h = 5, freq = TRUE)
```

```
Warning in plot.histogram(r, freq = freq1, col = col, border = border, angle =
angle, : the AREAS in the plot are wrong -- rather use 'freq = FALSE'
```



```r
my_hist(h = 0) ## Result will depend on implementation of 'my_breaks'
```



Exercise A.8

```r
my_hist <- function(h, ...) {
  structure(
    hist(log(F12), function(x) my_breaks(x, h), plot = FALSE, ...),
    class = "my_histogram"
    )
}
```

Note how `...` is used to pass on arguments to `hist`.

And then we try it out:

```r
class(my_hist(40))
```

```
[1] "my_histogram"
```

```r
my_hist(40)
```

```
$breaks
 [1] -2.99573227 -1.77195684 -1.38629436 -1.20397280 -1.04982212 -0.89159812
 [7] -0.73396918 -0.59783700 -0.46203546 -0.32850407 -0.10536052  0.03922071
[13]  0.25464222  0.60431597  7.93088943

$counts
 [1] 40 49 47 45 41 41 41 47 42 45 41 43 40 66

$density
 [1] 0.05204735 0.20231545 0.41048774 0.46484421 0.41262149 0.41417916
 [7] 0.47958262 0.55110394 0.50084837 0.32112087 0.45155671 0.31784820
[13] 0.18215342 0.01434443

$mids
 [1] -2.3838446 -1.5791256 -1.2951336 -1.1268975 -0.9707101 -0.8127836
 [7] -0.6659031 -0.5299362 -0.3952698 -0.2169323 -0.0330699  0.1469315
[13]  0.4294791  4.2676027

$xname
[1] "log(F12)"

$equidist
[1] FALSE

attr(,"class")
[1] "my_histogram"
```

Next we write the print method.

```r
print.my_histogram <- function(x)
  cat(length(x$counts))
```

```r
my_hist(40)
```

```
14
```

Note that R (the graphics and base packages, to be specific) implements generic `plot`, `print` and `summary` functions. To implement a method for such generic functions, all you need is to implement a function called `print.my_histogram`, say, following the naming convention `f.classname` for the method for class `classname` for generic function `f`. Also note that you don't need to test in `print.my_histogram` whether its argument is of class my_histogram, because the method is only called for objects of this class. Finally, you will never explicitly call `print.my_histogram`, but you will call `print` with an argument of class my_histogram, and the so-called *dispatch mechanism* in R will then call `print.my_histogram`.

Note that

```
plot(my_hist(40))
```

```
Error in xy.coords(x, y, xlabel, ylabel, log): 'x' is a list, but does not have components 'x' and 'y'
```

gives an error. The error message is cryptic.

One could imagine that the call should still produce a plot of the histogram, but it doesn't. Since we have modified the class label, what happens is that R does no know that it should use `plot.histogram`, and it calls `plot.default`. This function cannot find suitable `x` and `y` components and complains.

There is a very simple way of making our class "inherit" the histogram class.

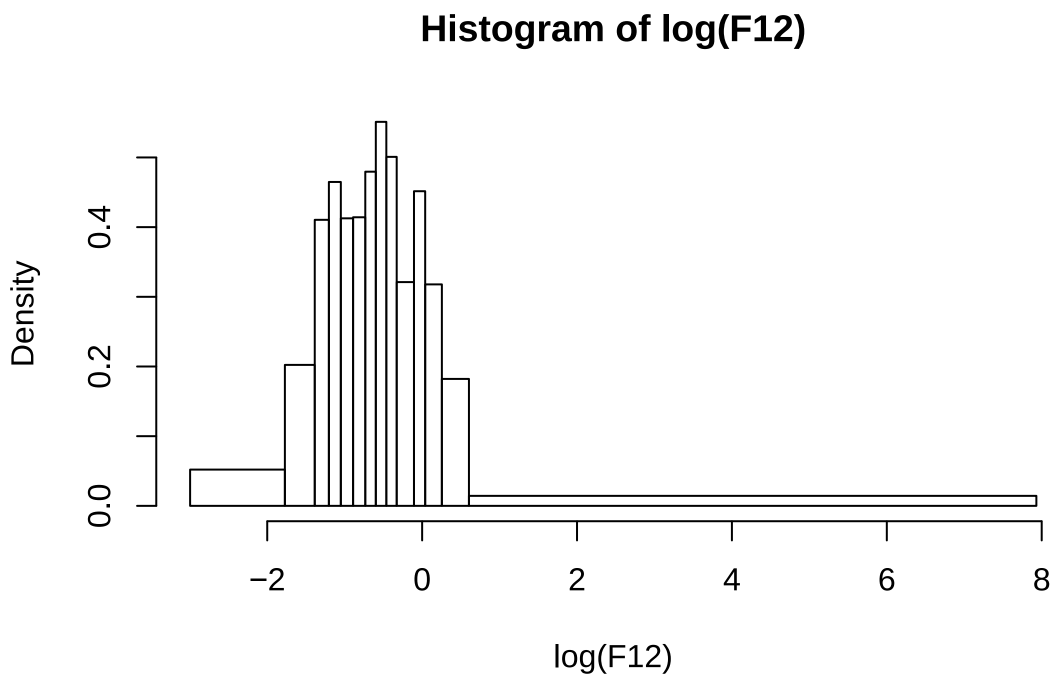
```
my_hist <- function(h, ...)
  structure(hist(log(F12), function(x) my_breaks(x, h), plot = FALSE, ...),
            class = c("my_histogram", "histogram"))
```

And now the result is printed using our method and plotted using the method for objects of class histogram.

```
my_hist(40)
```

```
14
```

```
plot(my_hist(40))
```



Exercise A.9

```
summary.my_histogram <- function(x)
  as.data.frame(x[c("mids", "counts")])
```

```r
summary(my_hist(40))
```

```
         mids counts
1  -2.3838446     40
2  -1.5791256     49
3  -1.2951336     47
4  -1.1268975     45
5  -0.9707101     41
6  -0.8127836     41
7  -0.6659031     41
8  -0.5299362     47
9  -0.3952698     42
10 -0.2169323     45
11 -0.0330699     41
12  0.1469315     43
13  0.4294791     40
14  4.2676027     66
```

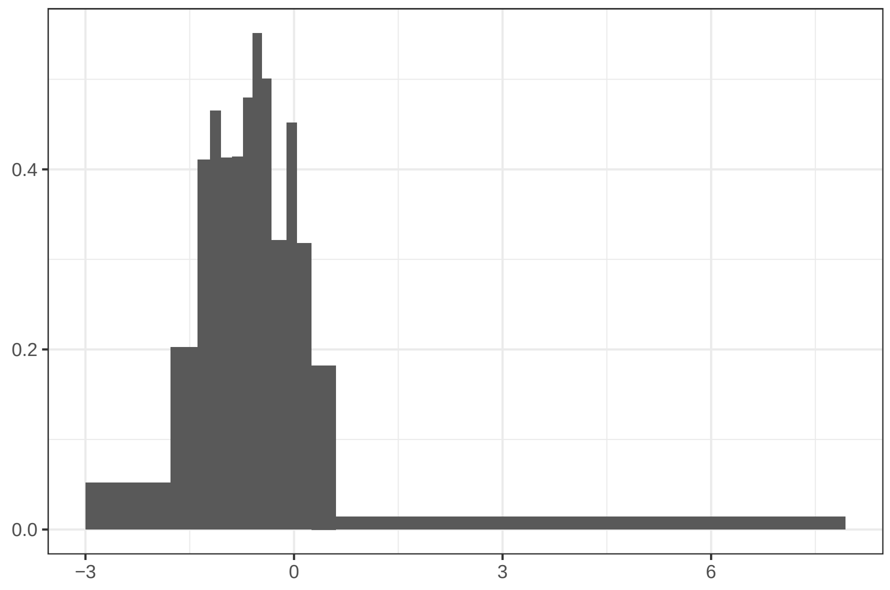
Note that in the implementation above, the entries in the list are referred to by names. This makes the implementation robust to internal changes in the number of components in the object, and is good practice. It is even better practice to use accessor functions provided by the programmer for the class. This is not widely used in R with S3 classes, but some examples include the functions `coefficients` and `residuals`, which are used together with objects of class lm or glm, say.

Exercise A.10

The plot method uses `geom_rect` from the package ggplot2 to plot the bars.

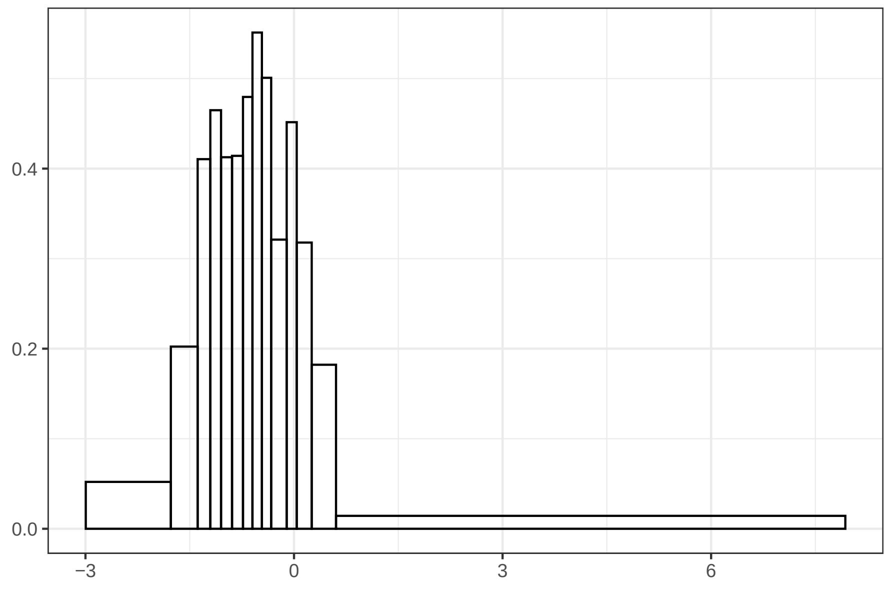
```r
library(ggplot2)
plot.my_histogram <- function(x, plot = TRUE, ...) {
  hist_data <- data.frame(
    breaksLeft = x$breaks[-length(x$breaks)],
    breaksRight = x$breaks[-1],
    density = x$density
  )
  p <- geom_rect(
    data = hist_data,
    mapping = aes(
      xmin = breaksLeft,
      xmax = breaksRight,
      ymin = 0,
      ymax = density),
    ...
  )
  if (plot)
    p <- ggplot() + p
  p
}
```

```r
plot(my_hist(40))
```



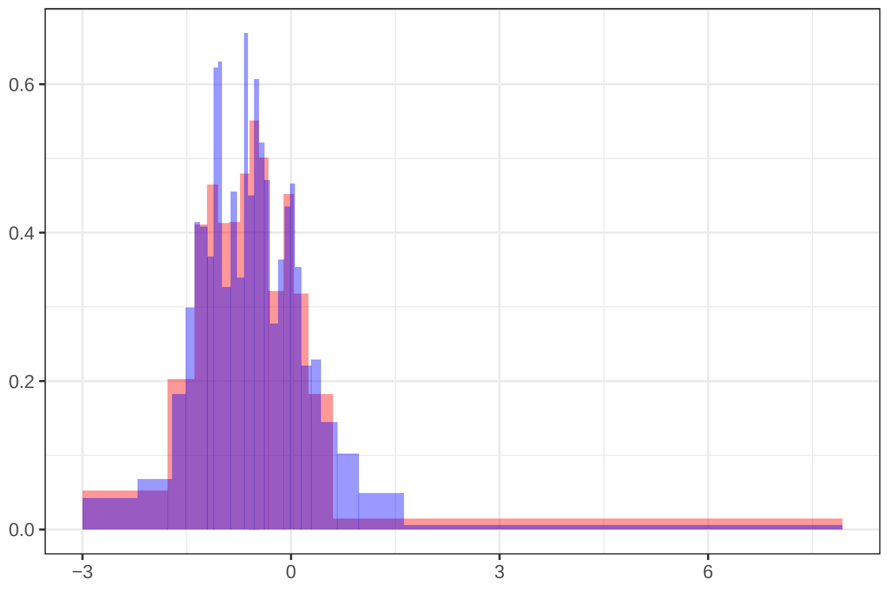
The method implements that all additional arguments are passed on to `geom_rect`, which allows us to change the colors of the lines and the fill etc.

```r
plot(my_hist(40), color = "black", fill = NA)
```



We can also make the histogram semitransparent and overplot it with another one for a different value of *h*.

```r
plot(my_hist(40), fill = "red", alpha = 0.4) +
plot(my_hist(20), plot = FALSE, fill = "blue", alpha = 0.4)
```


0.6
0.4
0.2
0.0
−3
0
3
6