

Quantitative Methods in Linguistics – Lecture 4

Adrian Brasoveanu*

March 30, 2014

Contents

1	Elementary Control (of) Flow Structures	2
1.1	IF	2
1.2	FOR	2
1.3	Example: obtaining the first 12 Fibonacci numbers	3
1.4	WHILE	4
2	Taking advantage of the vectorial nature of R	5
2.1	An example problem	5
2.1.1	Solution 1	5
2.1.2	Solution 2 (better)	5
2.1.3	Solution 3 (best)	6
2.2	Another example	6
2.2.1	Solution 1	6
2.2.2	Solution 2	6
2.3	A final example	7
2.3.1	Solution 1	7
2.3.2	Solution 2	7
3	General programming tips	8
3.1	Top-down design	8
3.2	Debugging and maintenance	8
3.3	Measuring the time your program takes (very basic profiling)	9
4	Goals of quantitative modeling and analysis and types of data	10
5	Introducing the idea of a probability distribution: Frequency distributions	11
6	Becoming familiar with the normal distribution	23

This set of notes is partly based on Gries (2009), Braun and Murdoch (2007) and Johnson (2008).

*These notes have been generated with the ‘knitr’ package (Xie 2013) and are based on many sources, including but not limited to: Abelson (1995), Miles and Shevlin (2001), Faraway (2004), De Veaux et al. (2005), Braun and Murdoch (2007), Gelman and Hill (2007), Baayen (2008), Johnson (2008), Wright and London (2009), Gries (2009), Kruschke (2011), Diez et al. (2013), Gries (2013).

1 Elementary Control (of) Flow Structures

1.1 IF

(1) IF syntax:

```
if (condition) {  
    statements when TRUE  
} else {  
    statements when FALSE  
}
```

```
> a <- 2  
> if (a > 2) {  
+   cat("a is greater than 2.\n")  
+ } else if (a == 2) {  
+   cat("a is equal to 2.\n")  
+ } else {  
+   cat("a is less than 2.\n")  
+ }  
  
a is equal to 2.  
  
> a <- 3  
> a <- 1  
> b <- "car"  
> if (b == "automobile") {  
+   cat("b is 'automobile'!!!!\n")  
+ } else if (b == "car") {  
+   cat("b is 'car'!\n")  
+ } else {  
+   cat("b is neither 'automobile' nor 'car'!!!!\n")  
+ }  
  
b is 'car'!  
  
> b <- "automobile"  
> b <- "Spanish Inquisition"
```

1.2 FOR

(2) FOR syntax:

```
for (name in vector) {  
    statements  
}
```

```
> for (i in 1:3) {  
+   cat(i, "\n")  
+ }  
  
1  
2  
3
```

```

> (j <- seq(2, 1, -0.15))

[1] 2.00 1.85 1.70 1.55 1.40 1.25 1.10

> for (i in j) {
+   cat(i, "\n")
+ }

2
1.85
1.7
1.55
1.4
1.25
1.1

> for (i in 1:5) {
+   for (j in 6:10) {
+     cat(i, "times", j, "is", i * j, "\n")
+   }
+ }

1 times 6 is 6
1 times 7 is 7
1 times 8 is 8
1 times 9 is 9
1 times 10 is 10
2 times 6 is 12
2 times 7 is 14
2 times 8 is 16
2 times 9 is 18
2 times 10 is 20
3 times 6 is 18
3 times 7 is 21
3 times 8 is 24
3 times 9 is 27
3 times 10 is 30
4 times 6 is 24
4 times 7 is 28
4 times 8 is 32
4 times 9 is 36
4 times 10 is 40
5 times 6 is 30
5 times 7 is 35
5 times 8 is 40
5 times 9 is 45
5 times 10 is 50

```

1.3 Example: obtaining the first 12 Fibonacci numbers

Fibonacci sequence: the first two elements are defined as $[1, 1]$, subsequent elements are defined as the sum of the preceding two elements; the third element is $2(= 1 + 1)$, the fourth element is $3(= 1 + 2)$, the fifth element is $5(= 2 + 3)$ etc.

```

> (Fibonacci <- numeric(12)) # set up a numeric vector of length 12

[1] 0 0 0 0 0 0 0 0 0 0 0 0

> Fibonacci[1] <- Fibonacci[2] <- 1 # update the first two elements to store the value 1
> Fibonacci

[1] 1 1 0 0 0 0 0 0 0 0 0 0

> for (i in 3:12) {
+   # update the rest of the elements in the sequence
+   Fibonacci[i] <- Fibonacci[i - 2] + Fibonacci[i - 1]
+ }
> Fibonacci

[1] 1 1 2 3 5 8 13 21 34 55 89 144

```

1.4 WHILE

(3) **WHILE** syntax:

```

while (condition) {
  statements
}

```

```

> i <- 1
> while (i < 9) {
+   cat(i, "\n")
+   i <- i + 1
+ }

1
2
3
4
5
6
7
8

```

Example: list all Fibonacci numbers less than 300. We don't know how long this list is, so we don't know how to stop the `for()` loop at the right time, but a `while()` loop is just right.

```

> Fib1 <- 1
> Fib2 <- 1
> (Fibonacci <- c(Fib1))

[1] 1

> while (Fib2 < 300) {
+   Fibonacci <- c(Fibonacci, Fib2)
+   oldFib2 <- Fib2
+   Fib2 <- Fib1 + Fib2
+   Fib1 <- oldFib2
+ }

```

```
+ }
> Fibonacci

[1] 1 1 2 3 5 8 13 21 34 55 89 144 233
```

2 Taking advantage of the vectorial nature of R

2.1 An example problem

Goal: sum over TokenFrequencies for each type of class (open vs. closed):

```
> PartOfSpeech <- c("ADJ", "ADV", "N", "CONJ", "PREP")
> TokenFrequency <- c(421, 337, 1411, 458, 455)
> TypeFrequency <- c(271, 103, 735, 18, 37)
> Class <- c("open", "open", "open", "closed", "closed")
> (x <- data.frame(PartOfSpeech, TokenFrequency, TypeFrequency, Class))
```

	PartOfSpeech	TokenFrequency	TypeFrequency	Class
1	ADJ	421	271	open
2	ADV	337	103	open
3	N	1411	735	open
4	CONJ	458	18	closed
5	PREP	455	37	closed

2.1.1 Solution 1

```
> sum.closed <- 0
> sum.open <- 0
> for (i in 1:5) {
+   current.class <- Class[i] # access each word class
+   if (current.class == "closed") {
+     # test each type of class
+     sum.closed <- sum.closed + TokenFrequency[i] # add token freq to closed vector
+   } else {
+     sum.open <- sum.open + TokenFrequency[i] # add token freq to open vector
+   }
+ }
> results <- c(sum.closed, sum.open)
> names(results) <- c("sum.closed", "sum.open")
> results

sum.closed  sum.open
      913      2169
```

2.1.2 Solution 2 (better)

```
> sum.closed <- sum(TokenFrequency[which(Class == "closed")])
> sum.open <- sum(TokenFrequency[which(Class == "open")])
> results <- c(sum.closed, sum.open)
```

```
> names(results) <- c("sum.closed", "sum.open")
> results

sum.closed  sum.open
      913      2169
```

2.1.3 Solution 3 (best)

```
> tapply(TokenFrequency, Class, sum)

closed  open
   913   2169
```

2.2 Another example

Goal: determine the length of elements in lists.

```
> (another_list <- list(c(1), c(2, 3), c(4, 5, 6), c(7, 8, 9, 10)))

[[1]]
[1] 1

[[2]]
[1] 2 3

[[3]]
[1] 4 5 6

[[4]]
[1] 7 8 9 10
```

2.2.1 Solution 1

```
> lengths <- vector()
> for (i in 1:length(another_list)) {
+   lengths[i] <- length(another_list[[i]])
+ }
> lengths

[1] 1 2 3 4
```

2.2.2 Solution 2

```
> sapply(another_list, length)

[1] 1 2 3 4
```

2.3 A final example

Goal: retrieve the first element from the individual elements in a list.

2.3.1 Solution 1

```
> first_elements <- vector()
> for (i in 1:length(another_list)) {
+   first_elements[i] <- another_list[[i]][1]
+ }
> first_elements

[1] 1 2 4 7
```

2.3.2 Solution 2

```
> sapply(another_list, "[", 1)

[1] 1 2 4 7

> a_vector <- c(1:10)
> a_dataframe <- read.table("dataframe.txt", header = T, sep = "\t",
+   comment.char = "")
> another_vector <- c("This", "may", "be", "a", "sentence", "from",
+   "a", "corpus", "file", ".")
> (a_list <- list(a_vector, a_dataframe, another_vector))

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

[[2]]
  PartOfSpeech TokenFrequency TypeFrequency Class
1          ADJ           421           271  open
2          ADV           337           103  open
3           N          1411           735  open
4         CONJ           458            18 closed
5         PREP           455            37 closed

[[3]]
[1] "This"      "may"      "be"      "a"      "sentence" "from"
[7] "a"         "corpus"   "file"    "."

> sapply(a_list, "[", 1)

[[1]]
[1] 1

$PartOfSpeech
[1] ADJ ADV N CONJ PREP
Levels: ADJ ADV CONJ N PREP

[[3]]
[1] "This"
```

```

> a <- c(1, 5, 3)
> b <- c(2, 6, 4)
> ab <- list(a, b)
> ab

[[1]]
[1] 1 5 3

[[2]]
[1] 2 6 4

> lapply(ab, sort, decreasing = F)

[[1]]
[1] 1 3 5

[[2]]
[1] 2 4 6

```

3 General programming tips

Writing a computer program to solve a problem can usually be reduced to following this sequence of steps:

- (4)
 - a. Understand the problem.
 - b. Work out a general idea of how to solve it.
 - c. Translate your general idea into a detailed implementation.
 - d. Check: Does it work? Is it good enough?
 - i. If yes, you are done.
 - ii. If no, go back to step (4b).

3.1 Top-down design

Working out the detailed implementation of a program can appear to be a daunting task. The key to making it manageable is to break it down into smaller pieces which you know how to solve.

One strategy for doing that is known as ‘top-down design’. Top-down design is similar to outlining an essay before filling in the details (basically, writing a handout):

- (5) Top-down design:
 - a. Write out the whole program in a small number of steps.
 - b. Expand each step into a small number of steps.
 - c. Keep going until you have a program.

3.2 Debugging and maintenance

Computer errors are called ‘bugs’. Removing these errors from a program is called ‘debugging’. The following five steps can help you find and fix bugs in our own programs:

- (6) Finding and fixing bugs:
 - a. Recognize that a bug exists.
 - b. Make the bug reproducible.
 - c. Identify the cause of the bug.

- d. Fix the error and test.
- e. Look for similar errors.

In R, you can obtain extra information about an error message using the `traceback()` function. When an error occurs, R saves information about the current stack of active functions, and `traceback()` prints this list.

3.3 Measuring the time your program takes (very basic profiling)

`system.time()` measures the execution time of (sub)scripts. We can use it to see the benefits of vectorization in R.

```
> X <- rnorm(1e+06)
> X[1:30]

[1] -1.35200  0.32099 -0.14685 -1.23917 -1.00480  2.42491 -0.84709
[8]  0.14526 -1.68981  0.83843  0.75752  0.89119  0.48017 -0.61637
[15] -0.17039  0.47645 -0.18846  1.93962 -0.33286  1.43413  0.02970
[22] -0.07523 -0.68624  0.87679 -1.15486 -0.93678 -0.28119 -0.94176
[29]  1.49497 -0.37853

> length(X)

[1] 1000000

> Y <- rnorm(1e+06)
> Z <- rep(NA, 1e+06)
> system.time({
+   for (i in 1:1e+06) {
+     Z[i] <- X[i] + Y[i]
+   }
+ })

   user  system elapsed 
1.492    0.000   1.499 

> X[1:5]

[1] -1.3520  0.3210 -0.1468 -1.2392 -1.0048

> Y[1:5]

[1] -1.0579  0.7440  1.3776 -0.2574 -2.1486

> Z[1:5]

[1] -2.410  1.065  1.231 -1.497 -3.153

> Z <- rep(NA, 1e+06)
> system.time({
+   Z <- X + Y
+ })

   user  system elapsed 
0.000    0.000   0.002 

> X[1:5]
```

```
[1] -1.3520  0.3210 -0.1468 -1.2392 -1.0048

> Y[1:5]

[1] -1.0579  0.7440  1.3776 -0.2574 -2.1486

> Z[1:5]

[1] -2.410  1.065  1.231 -1.497 -3.153
```

4 Goals of quantitative modeling and analysis and types of data

This section and the immediately following ones are partly based on Johnson (2008), who in turn follows Kachigan (1991).

- (7) The main goals of quantitative modeling and analysis are:
 - a. data reduction: summarize trends, capture the common aspects of a set of observations such as the average, standard deviation, and correlations among variables
 - b. inference: generalize from a representative set of observations to a larger universe of possible observations using statistical models, and doing hypothesis testing on the estimated parameters of those models
 - c. more generally, discovery of relationships: find descriptive or causal patterns in data based on your interpretation of various statistical models
 - d. theory construction: proposing probabilistic theoretical models for various aspects of language competence and/or performance (e.g., probabilistic parsing, probabilistic pragmatics)
- (8) Types of data / variables:
 - a. qualitative/categorical variables:
 - i. nominal: binary/binomial, multinomial; e.g, binary data – the answer *yes* or *no* to the question *Did you skin the cat?*, multinomial data – set of possible answers to the question *How did you skin the cat?*, including but not limited to (you can google for *50 good ways to skin a cat* to see more options):
 - knife
 - can opener
 - set a series of short-term easily attainable goals, resulting in skinned cat; accomplish goals
 - peer pressure, *all the cool cats are getting skinned*
 - give cat post-hypnotic suggestion to get skinned every time it hears the question *is it hot in here?*; later, ask the question
 - rent instructional cat skinning video, study carefully, and apply what you learn
 - tell cat pleasant tale about a young boy who loves fruit; while cat is distracted by story, quietly, gently remove skin
 - centrifugal force
 - suddenly and severely frighten cat
 - marry cat; divorce cat; take cat to court for half of skin; repeat for full skin
 - allow cat to evolve beyond need for skin
 - if in a horror movie, dream about cat getting skinned; wake up to discover cat was really skinned
 - press cat's eject button

- other (more realistic) examples: what language is being observed? what dialect? which word? what is the gender of the person being observed?
- ii. ordinal: ordered outcomes (total order, i.e., antisymmetric, transitive, total), e.g., Zipf's rank frequency of words, rating scales (e.g. excellent, good, fair, poor) for acceptability, performance on a task etc., agreement scales (strongly disagree, disagree, neither agree nor disagree, agree, strongly agree), i.e., Likert scales in general
- b. quantitative/numeric variables, which can be classified along two dimensions:
- i. discrete: typically counts, e.g., the number of typos in a paper; the number of flips of a coin you have to wait until you get heads; warning: sometimes nominal and ordinal variables are coded as discrete (numeric) variables for convenience, but that doesn't make them numeric!
 - ii. continuous: temperature, height, weight, reaction time, the time you have to wait until the next bus comes
- i. interval: a property measured on a scale that does not have a true 0 value; the magnitude of differences of adjacent observations can be determined (unlike for adjacent observations on an ordinal scale), but because the 0 value is arbitrary, ratios are not really meaningful (something can be twice as hot as something else on the Celsius, but not on the Fahrenheit scale), magnitude estimation judgments
 - ii. ratio: a property that we measure on a scale that has an absolute 0 value; ratios of these measurements are meaningful; e.g., a vowel that is 100 ms long is twice as long as a 50 ms vowel, an object that weighs 2 pounds is twice as heavy as an object that weighs 1 pound, if a verb was observed 4 times in a text and another verb was observed 8 times, the second verb is twice as frequent as the first

5 Introducing the idea of a probability distribution: Frequency distributions

Suppose we ask 36 people to score the acceptability of a sentence on a five-point scale from "-2" to "2", with "-2" clearly unacceptable and 2 perfectly acceptable. Suppose that we get the following 36 ratings:

```
> y <- round(rnorm(36, 0, 1))
> y[y > 2] <- 2
> y[y < (-2)] <- -2
> y

[1]  1  1 -2  1  1  0  1  0 -1  0  0  0  2  1  2  0  1  0  0 -2  1  0 -1
[24]  0 -1 -1  0  0  0  0  2  1 -2 -1  0  0
```

Let's convert our ratings into the intended ordinal scale:

```
> y <- as.factor(y)
> levels(y)

[1] "-2" "-1" "0"  "1"  "2"

> levels(y) <- c("really unacceptable", "fairly unacceptable", "so-so",
+               "fairly acceptable", "perfectly acceptable")
> y

[1] fairly acceptable    fairly acceptable    really unacceptable
[4] fairly acceptable    fairly acceptable    so-so
[7] fairly acceptable    so-so               fairly unacceptable
```

```

[10] so-so          so-so          so-so
[13] perfectly acceptable fairly acceptable perfectly acceptable
[16] so-so          fairly acceptable so-so
[19] so-so          really unacceptable fairly acceptable
[22] so-so          fairly unacceptable so-so
[25] fairly unacceptable fairly unacceptable so-so
[28] so-so          so-so          so-so
[31] perfectly acceptable fairly acceptable really unacceptable
[34] fairly unacceptable so-so          so-so
5 Levels: really unacceptable fairly unacceptable ... perfectly acceptable

```

What can we learn about the acceptability of our sentence based on these 36 numbers? Looking at the raw data is not very revealing, but we can construct a frequency distribution based on this data:

- we put all the identical ratings in the same bin: all the "-2" ratings in the "-2" bin, all the "-1" ratings in the "-1" bin, etc.
- we count how many ratings we have in each of the 5 bins

```

> table(y)

y
really unacceptable  fairly unacceptable          so-so
                   3                5              16
  fairly acceptable perfectly acceptable
                   9                3

> xtabs(~y)

y
really unacceptable  fairly unacceptable          so-so
                   3                5              16
  fairly acceptable perfectly acceptable
                   9                3

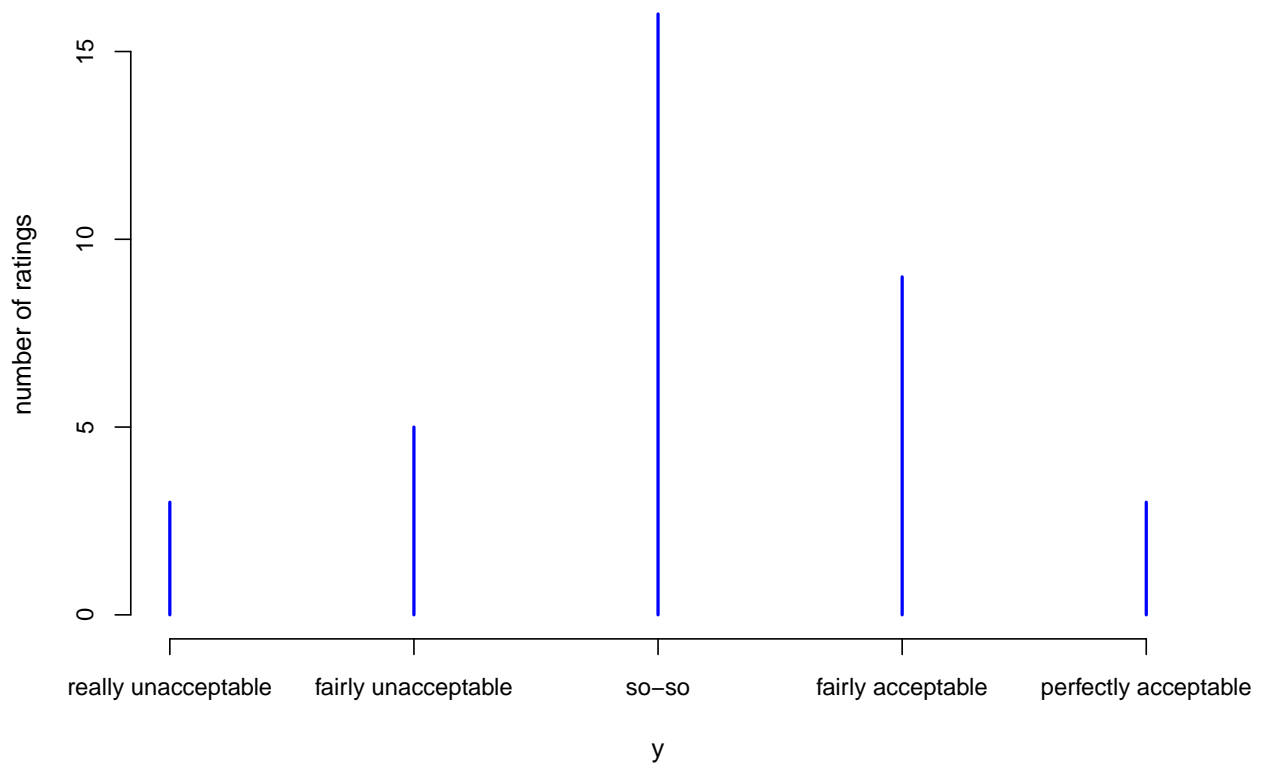
```

This frequency distribution is much more useful. And plotting it really shows how acceptable the sentence is: the sentence is so-so – as expected given that we obtained the ratings by truncating random draws from a normal distribution centered at 0.

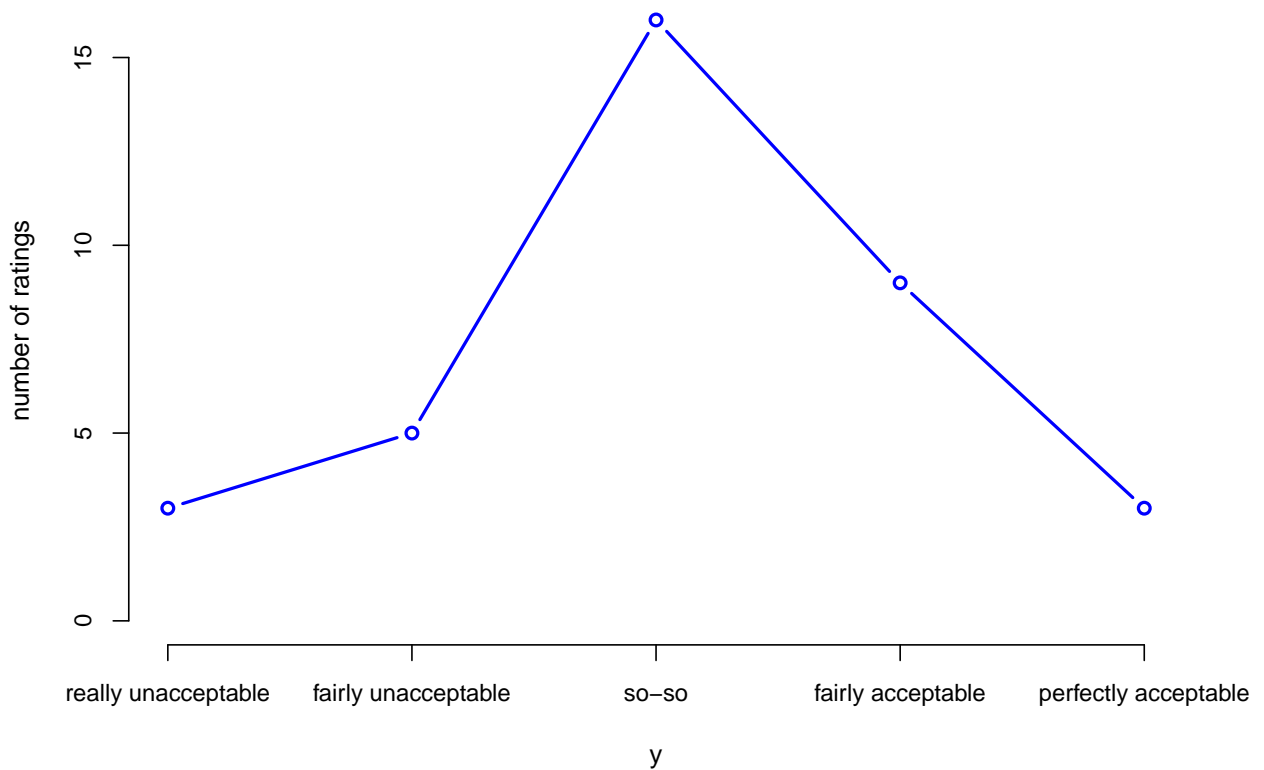
```

> plot(table(y), type = "h", col = "blue", ylab = "number of ratings",
+       cex.axis = 0.9)

```



```
> plot(table(y), type = "b", col = "blue", ylab = "number of ratings",  
+       cex.axis = 0.9)
```

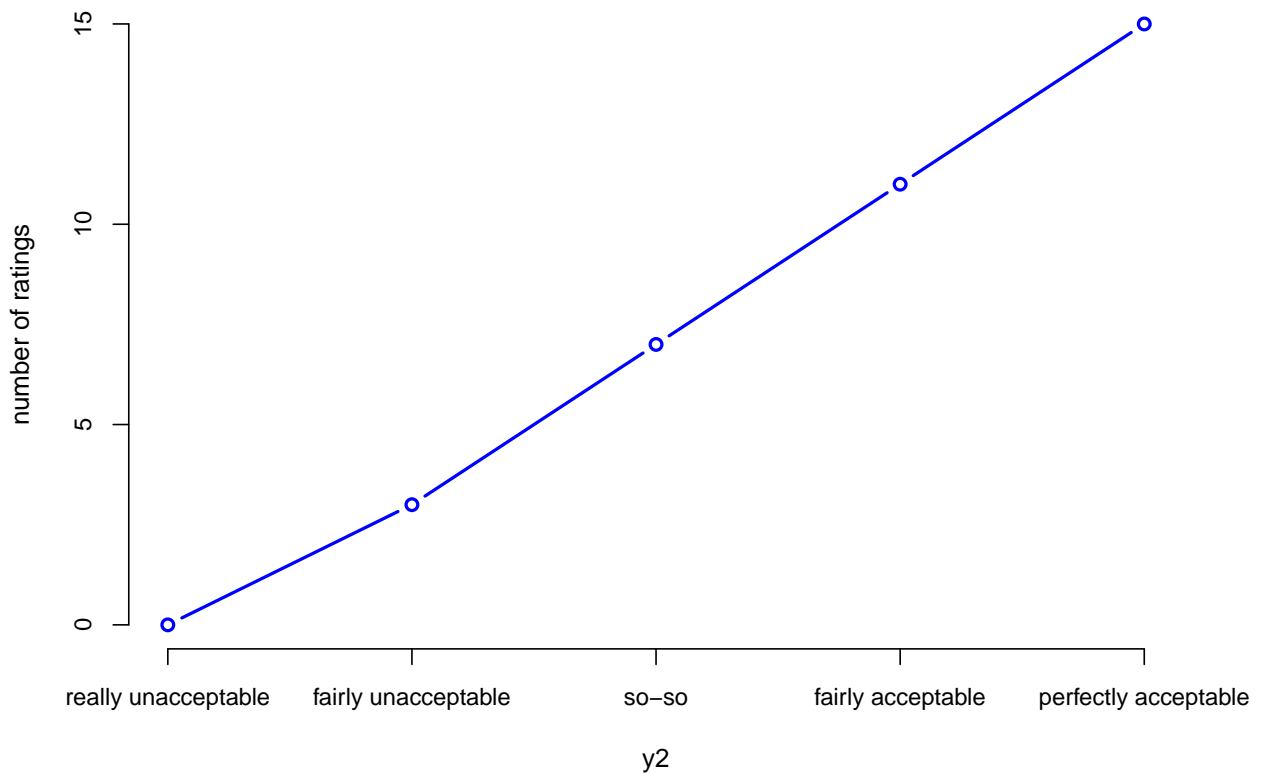


Let's compare this with the ratings for a new sentence, which is supposed to be more acceptable:

```
> y2 <- round(rnorm(36, 1, 1))
> y2[y2 > 2] <- 2
> y2[y2 < (-2)] <- -2
> y2 <- factor(y2, levels = c("-2", "-1", "0", "1", "2"))
> levels(y2) <- c("really unacceptable", "fairly unacceptable", "so-so",
+   "fairly acceptable", "perfectly acceptable")
> table(y2)
```

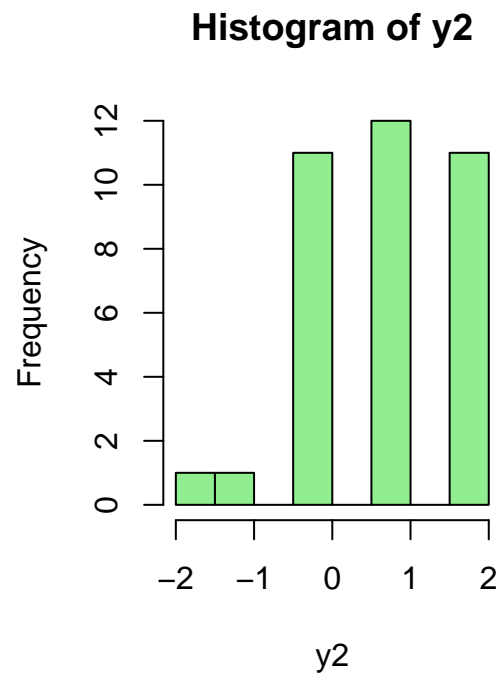
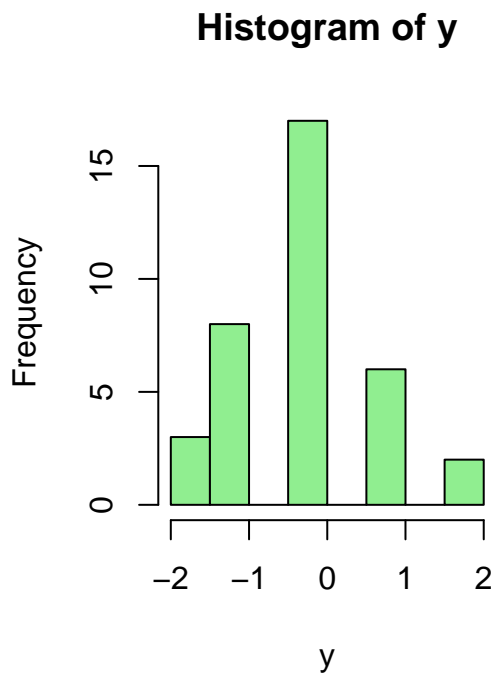
```
y2
really unacceptable  fairly unacceptable          so-so
                   0                   3              7
  fairly acceptable perfectly acceptable
                   11                   15
```

```
> plot(table(y2), type = "b", col = "blue", ylab = "number of ratings",
+   cex.axis = 0.9)
```



An alternative way of displaying the frequency distribution is with a histogram – as long as we treat the variable as numeric:

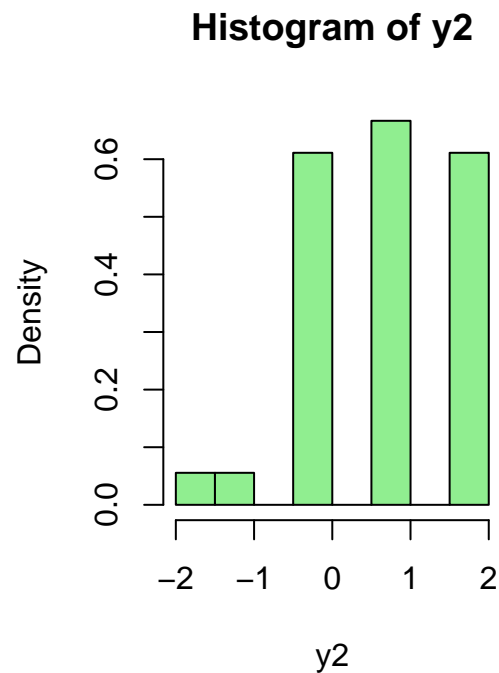
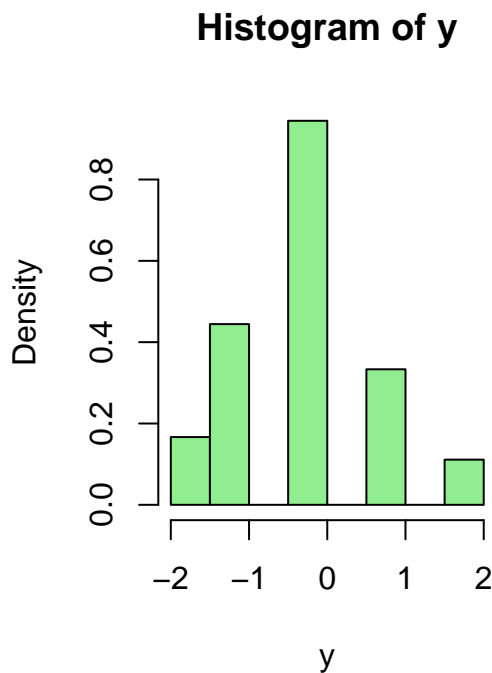
```
> y <- round(rnorm(36, 0, 1))
> y[y > 2] <- 2
> y[y < (-2)] <- -2
> y2 <- round(rnorm(36, 1, 1))
> y2[y2 > 2] <- 2
> y2[y2 < (-2)] <- -2
> par(mfrow = c(1, 2))
> hist(y, col = "lightgreen")
> hist(y2, col = "lightgreen")
```



```
> par(mfrow = c(1, 1))
```

We can actually display relative frequencies (i.e., proportions) by simply changing one of the defaults of the `hist` command:

```
> par(mfrow = c(1, 2))
> hist(y, col = "lightgreen", freq = F)
> hist(y2, col = "lightgreen", freq = F)
```

```
> par(mfrow = c(1, 1))
```

The y -axis of the histogram is now labeled ‘density’, although this is not entirely correct for our example: our variable is discrete, so the height of the bars should provide the *probability mass* – i.e., the proportion / relative frequency / percentage – associated with every one of the 5 possible ratings.

But the histogram command assumes that we gave it as input a continuous numeric variable (which we didn’t, and that’s our fault). And continuous numerical variables associate a probability mass of 0 with any particular outcome; only *sets* of outcomes have non-0 probability mass for continuous intervals, not single outcomes.

What we plot for continuous variables is their:

- (9) *Probability density*: probability mass per infinitesimal interval of outcomes, i.e., probability mass divided by the infinitesimal interval.

The name *density* is actually very intuitive. Take an object, e.g., a snowball: that object has

- a certain mass – the mass of the snow we pressed together to form the snowball, and
- a certain volume – the volume of the final ball.

The density:

- is the amount of mass per unit of volume, i.e., the ratio of mass and volume;
- measures how dense, i.e., how ‘tightly packed’, the material is in various regions of the volume we’re examining.

Take the snow that we had before we pressed it into a snowball: that snow had the same mass as the final snowball, but a much lower density because it had a much higher volume.

That is:

- the same (probability) mass can be distributed in different ways over a space (interval / region / volume) of outcomes, and
- the probability density indicates which sub-space (sub-interval / sub-region / sub-volume) is more likely depending on how ‘tightly packed’ the probability mass is over that sub-space.

Another example:

- take a balloon filled with air and squeeze it
- the density of the air inside, i.e., its mass per unit of volume, increases (and so does the pressure)
- but the mass of the air stays the same.

Whenever we deal with relative frequencies / proportions / percentages, i.e., probabilities, we *always* assume that:

- we have 1 unit of probability mass.

When the space of outcomes we distribute this probability mass over is:

- not continuous – e.g., nominal, ordinal or numeric discrete¹, each outcome is associated with a certain probability mass, namely the amount of probability that ‘sits on top of it’
- continuous
 - each particular outcome has probability 0: there is no mass associated with a point, just as there is no length, area, etc.
 - but sets / sub-spaces / sub-intervals of outcomes can have non-0 probability
 - so the best we can do is to take an infinitesimally small interval around the point of interest and look at the probability mass in that interval
 - the more probability we have on the infinitesimal interval around the point, i.e., the higher the probability density, the more likely that neighborhood is

We can approach the probability density curve of a distribution, for example the normal distribution, by taking a large sample of draws / points from that distribution and building histograms with smaller and smaller bins. That is:

- we take smaller and smaller intervals
- we count the points inside each of those intervals
- based on those counts, we obtain the relative frequency / probability mass sitting on top of each of those increasingly smaller intervals
- we divide that probability mass by the length of the intervals to approximate the density

For example:

```
> x <- rnorm(20000, 0, 1)
> grid_points <- seq(-10, 10, length.out = 1000)
> (bins_11 <- seq(-10, 10, length.out = 12))

[1] -10.0000 -8.1818 -6.3636 -4.5455 -2.7273 -0.9091  0.9091
[8]  2.7273  4.5455  6.3636  8.1818 10.0000
```

¹The geometric distribution is an example of a discrete distribution with a countably infinite set of possible outcomes, a.k.a. ‘support’. We’ll discuss this in the next lecture notes / set of slides.

```

> rel_freq_11 <- vector(length = length(bins_11) - 1)
> for (i in 2:length(bins_11)) {
+   rel_freq_11[i - 1] <- sum(x >= bins_11[i - 1] & x < bins_11[i])/length(x)
+ }
> rel_freq_11

[1] 0.0000 0.0000 0.0000 0.0034 0.1763 0.6373 0.1797 0.0032 0.0000 0.0000
[11] 0.0000

> mid_points_11 <- (bins_11[2:length(bins_11)] + bins_11[1:(length(bins_11) -
+   1)])/2
> density_11 <- rel_freq_11/(bins_11[2:length(bins_11)] - bins_11[1:(length(bins_11) -
+   1)])
> density_11

[1] 0.00000 0.00000 0.00000 0.00187 0.09697 0.35054 0.09886 0.00176
[9] 0.00000 0.00000 0.00000

> (bins_15 <- seq(-10, 10, length.out = 16))

[1] -10.0000 -8.6667 -7.3333 -6.0000 -4.6667 -3.3333 -2.0000
[8] -0.6667 0.6667 2.0000 3.3333 4.6667 6.0000 7.3333
[15] 8.6667 10.0000

> rel_freq_15 <- vector(length = length(bins_15) - 1)
> for (i in 2:length(bins_15)) {
+   rel_freq_15[i - 1] <- sum(x >= bins_15[i - 1] & x < bins_15[i])/length(x)
+ }
> rel_freq_15

[1] 0.00000 0.00000 0.00000 0.00000 0.00045 0.02145 0.23030 0.49495
[9] 0.23080 0.02180 0.00025 0.00000 0.00000 0.00000 0.00000

> mid_points_15 <- (bins_15[2:length(bins_15)] + bins_15[1:(length(bins_15) -
+   1)])/2
> density_15 <- rel_freq_15/(bins_15[2:length(bins_15)] - bins_15[1:(length(bins_15) -
+   1)])
> density_15

[1] 0.0000000 0.0000000 0.0000000 0.0000000 0.0003375 0.0160875 0.1727250
[8] 0.3712125 0.1731000 0.0163500 0.0001875 0.0000000 0.0000000 0.0000000
[15] 0.0000000

> (bins_31 <- seq(-10, 10, length.out = 32))

[1] -10.0000 -9.3548 -8.7097 -8.0645 -7.4194 -6.7742 -6.1290
[8] -5.4839 -4.8387 -4.1935 -3.5484 -2.9032 -2.2581 -1.6129
[15] -0.9677 -0.3226 0.3226 0.9677 1.6129 2.2581 2.9032
[22] 3.5484 4.1935 4.8387 5.4839 6.1290 6.7742 7.4194
[29] 8.0645 8.7097 9.3548 10.0000

> rel_freq_31 <- vector(length = length(bins_31) - 1)
> for (i in 2:length(bins_31)) {
+   rel_freq_31[i - 1] <- sum(x >= bins_31[i - 1] & x < bins_31[i])/length(x)
+ }
> rel_freq_31

```

```

[1] 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
[9] 0.00000 0.00010 0.00180 0.01030 0.04080 0.11150 0.20825 0.25375
[17] 0.20485 0.11515 0.04190 0.01005 0.00140 0.00015 0.00000 0.00000
[25] 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000

> mid_points_31 <- (bins_31[2:length(bins_31)] + bins_31[1:(length(bins_31) -
+ 1)))/2
> density_31 <- rel_freq_31/(bins_31[2:length(bins_31)] - bins_31[1:(length(bins_31) -
+ 1)])
> density_31

[1] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
[8] 0.0000000 0.0000000 0.0001550 0.0027900 0.0159650 0.0632400 0.1728250
[15] 0.3227875 0.3933125 0.3175175 0.1784825 0.0649450 0.0155775 0.0021700
[22] 0.0002325 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
[29] 0.0000000 0.0000000 0.0000000

> (bins_51 <- seq(-10, 10, length.out = 52))

[1] -10.0000 -9.6078 -9.2157 -8.8235 -8.4314 -8.0392 -7.6471
[8] -7.2549 -6.8627 -6.4706 -6.0784 -5.6863 -5.2941 -4.9020
[15] -4.5098 -4.1176 -3.7255 -3.3333 -2.9412 -2.5490 -2.1569
[22] -1.7647 -1.3725 -0.9804 -0.5882 -0.1961 0.1961 0.5882
[29] 0.9804 1.3725 1.7647 2.1569 2.5490 2.9412 3.3333
[36] 3.7255 4.1176 4.5098 4.9020 5.2941 5.6863 6.0784
[43] 6.4706 6.8627 7.2549 7.6471 8.0392 8.4314 8.8235
[50] 9.2157 9.6078 10.0000

> rel_freq_51 <- vector(length = length(bins_51) - 1)
> for (i in 2:length(bins_51)) {
+ rel_freq_51[i - 1] <- sum(x >= bins_51[i - 1] & x < bins_51[i])/length(x)
+ }
> rel_freq_51

[1] 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
[9] 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
[17] 0.00045 0.00125 0.00430 0.00975 0.02235 0.04775 0.07605 0.11750
[25] 0.14100 0.15660 0.14435 0.11345 0.07850 0.04825 0.02305 0.01005
[33] 0.00400 0.00110 0.00025 0.00000 0.00000 0.00000 0.00000 0.00000
[41] 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
[49] 0.00000 0.00000 0.00000

> mid_points_51 <- (bins_51[2:length(bins_51)] + bins_51[1:(length(bins_51) -
+ 1)))/2
> density_51 <- rel_freq_51/(bins_51[2:length(bins_51)] - bins_51[1:(length(bins_51) -
+ 1)])
> density_51

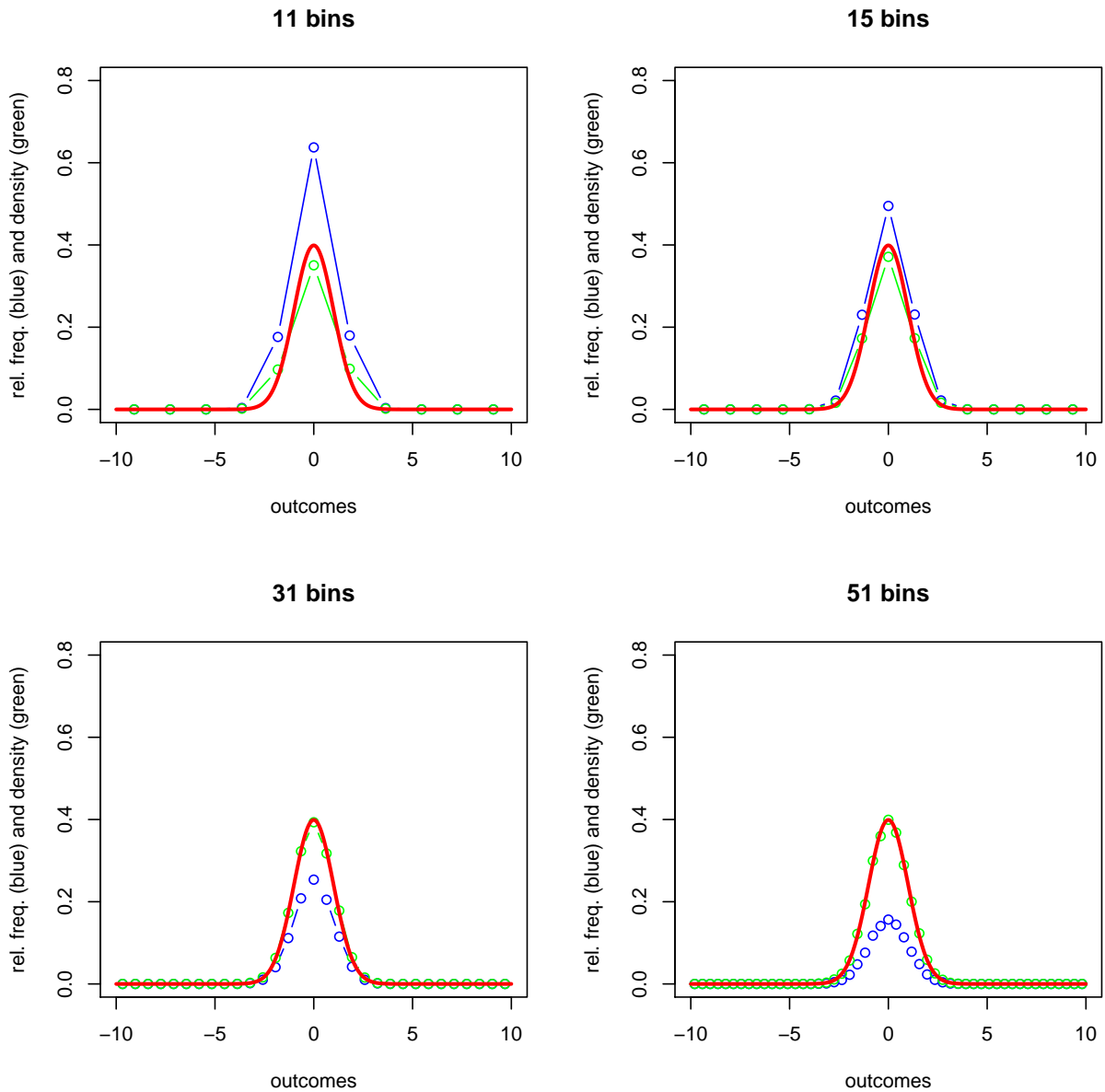
[1] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
[8] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
[15] 0.0000000 0.0000000 0.0011475 0.0031875 0.0109650 0.0248625 0.0569925
[22] 0.1217625 0.1939275 0.2996250 0.3595500 0.3993300 0.3680925 0.2892975
[29] 0.2001750 0.1230375 0.0587775 0.0256275 0.0102000 0.0028050 0.0006375
[36] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
[43] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
[50] 0.0000000 0.0000000

```

```

> par(mfrow = c(2, 2))
> plot(mid_points_11, rel_freq_11, xlim = c(-10, 10), ylim = c(0, 0.8),
+      type = "b", col = "blue", xlab = "outcomes", ylab = "rel. freq. (blue) and density (green)",
+      main = "11 bins")
> lines(mid_points_11, density_11, col = "green", type = "b")
> lines(grid_points, dnorm(grid_points, 0, 1), col = "red", lwd = 2.5)
> plot(mid_points_15, rel_freq_15, xlim = c(-10, 10), ylim = c(0, 0.8),
+      type = "b", col = "blue", xlab = "outcomes", ylab = "rel. freq. (blue) and density (green)",
+      main = "15 bins")
> lines(mid_points_15, density_15, col = "green", type = "b")
> lines(grid_points, dnorm(grid_points, 0, 1), col = "red", lwd = 2.5)
> plot(mid_points_31, rel_freq_31, xlim = c(-10, 10), ylim = c(0, 0.8),
+      type = "b", col = "blue", xlab = "outcomes", ylab = "rel. freq. (blue) and density (green)",
+      main = "31 bins")
> lines(mid_points_31, density_31, col = "green", type = "b")
> lines(grid_points, dnorm(grid_points, 0, 1), col = "red", lwd = 2.5)
> plot(mid_points_51, rel_freq_51, xlim = c(-10, 10), ylim = c(0, 0.8),
+      type = "b", col = "blue", xlab = "outcomes", ylab = "rel. freq. (blue) and density (green)",
+      main = "51 bins")
> lines(mid_points_51, density_51, col = "green", type = "b")
> lines(grid_points, dnorm(grid_points, 0, 1), col = "red", lwd = 2.5)

```

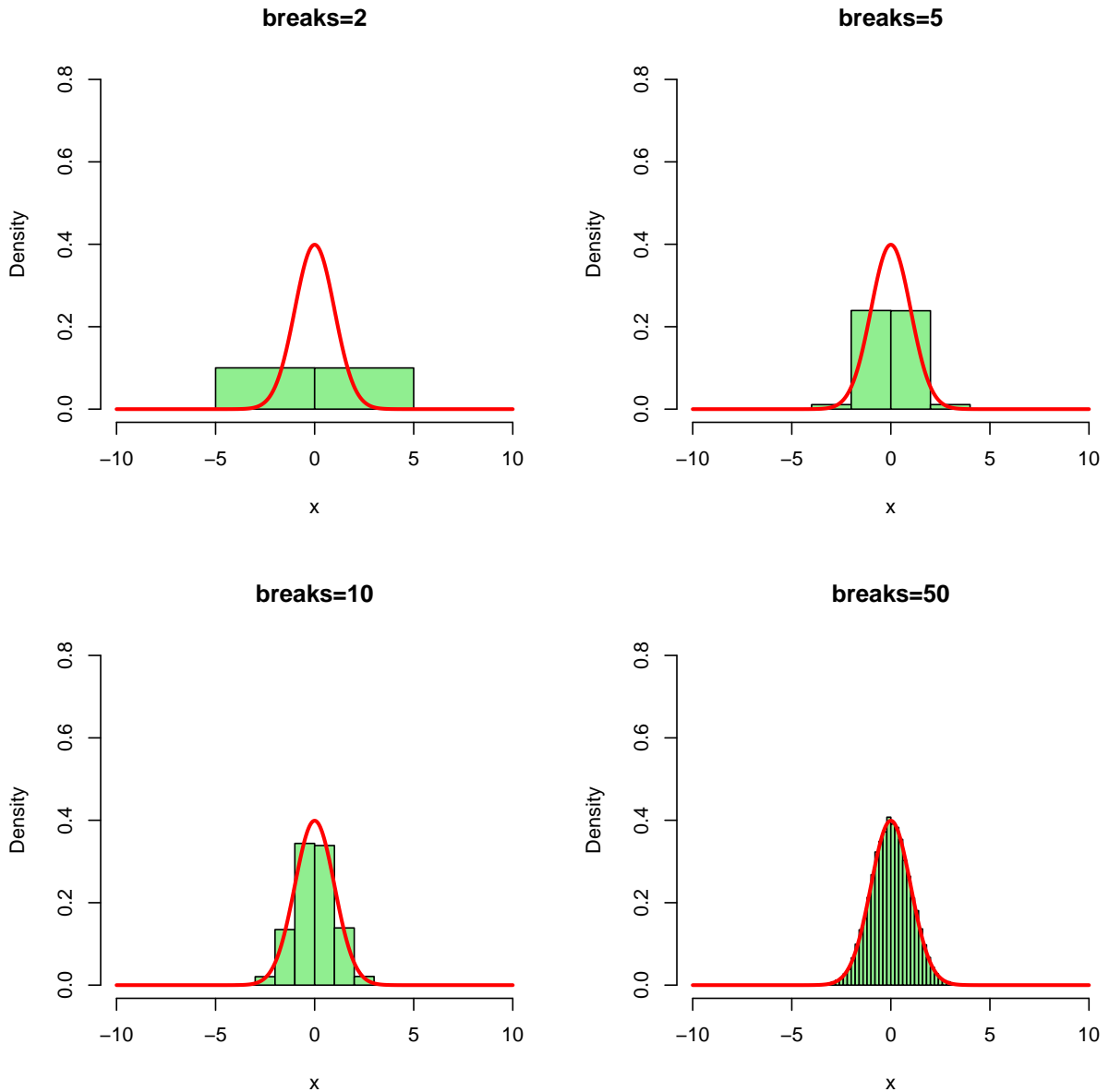


```
> par(mfrow = c(1, 1))
```

Histograms do the same density approximation for us, only more easily:

```
> par(mfrow = c(2, 2))
> hist(x, col = "lightgreen", xlim = c(-10, 10), ylim = c(0, 0.8), freq = F,
+      breaks = 2, main = "breaks=2")
> lines(grid_points, dnorm(grid_points, 0, 1), col = "red", lwd = 2.5)
> hist(x, col = "lightgreen", xlim = c(-10, 10), ylim = c(0, 0.8), freq = F,
+      breaks = 5, main = "breaks=5")
> lines(grid_points, dnorm(grid_points, 0, 1), col = "red", lwd = 2.5)
> hist(x, col = "lightgreen", xlim = c(-10, 10), ylim = c(0, 0.8), freq = F,
+      breaks = 10, main = "breaks=10")
> lines(grid_points, dnorm(grid_points, 0, 1), col = "red", lwd = 2.5)
```

```
> hist(x, col = "lightgreen", xlim = c(-10, 10), ylim = c(0, 0.8), freq = F,
+       breaks = 50, main = "breaks=50")
> lines(grid_points, dnorm(grid_points, 0, 1), col = "red", lwd = 2.5)
```



```
> par(mfrow = c(1, 1))
```

6 Becoming familiar with the normal distribution

The family of normal distributions is parametrized by their:

- mean: the central tendency, i.e., (literally) the center of the probability mass (the mass is 'balanced' is we place it on that point)

- standard deviation: the dispersion around the central tendency; we'll talk more about this, for now just think of it as a measure of dispersion around the center of the distribution

That is, for each mean (which can be any positive or negative real number) and any standard deviation (which can be any strictly positive real number), we get a different normal distribution, i.e., a different way of spreading our 1 unit of probability mass over the entire set of real numbers.

For example, the standard normal distribution is a normal distribution with mean 0 and standard deviation 1.

Normal distributions are also known as Gaussian distributions.

- (10) The probability density function of the standard normal:

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$$

- (11) The probability density of a $Normal(\mu, \sigma^2)$ distribution, i.e., a normal distribution with mean μ and standard deviation σ :

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Note that the standard way of writing up how a normal distribution is parametrized is in terms of its mean and *variance*, which is the square of the standard deviation. This is because certain properties of the distribution are more easily understood / talked about in terms of its variance rather than its standard deviation.

But be careful: in writing, we parametrize Gaussians in terms of means and variances, but R parametrizes normal distributions in terms of means and standard deviations.

Let's take a closer look at the probability density function (pdf) of the standard normal and try to understand it.

We start with the innermost function: x^2 .

- the values of this function are always positive, as shown in the first (top leftmost panel) below

Now the function $-x^2$ is:

- its mirror image – mirrored across the x -axis, i.e., all its values are negative and the function has a global maximum at 0 (see the second panel)

When we take any number a and raise it to a negative power $-z$, the result is:

- $a^{-z} = \frac{1}{a^z}$

And when we raise a number to the 0th power, we get 1.

We can take Euler's number e and raise it to the $-x^2$ power, as shown in the third panel:

- the values are always positive
- they asymptotically approach 0 as x goes to $+\infty$ / $-\infty$
- the function reaches its global maximum of 1 at $x = 0$

The fourth panel shows what happens when we actually raise e to the $-\frac{x^2}{2}$ power:

- the values decay a little bit slower towards 0 from their maximum at 1.

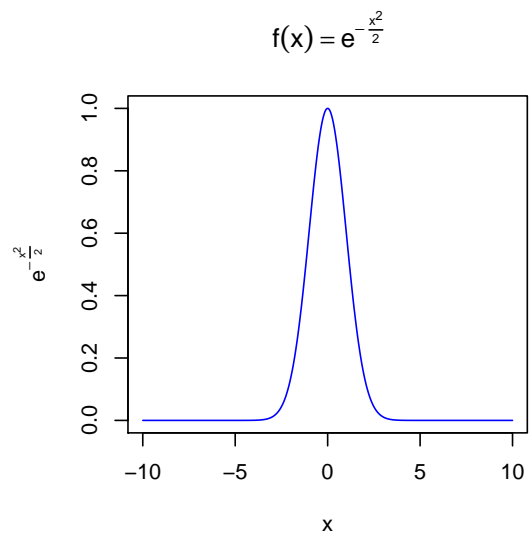
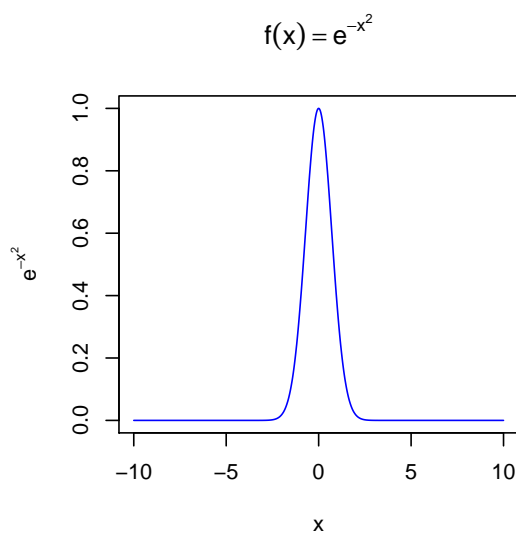
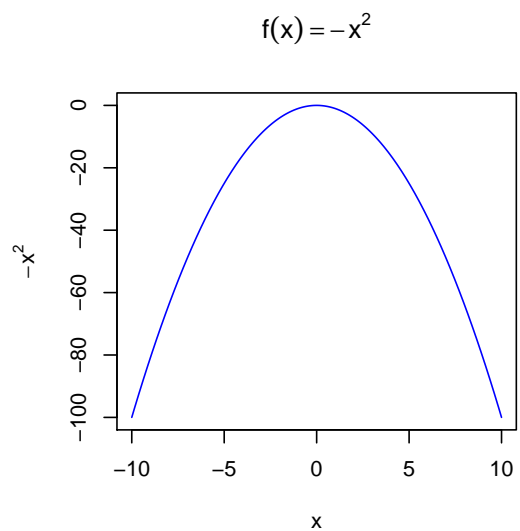
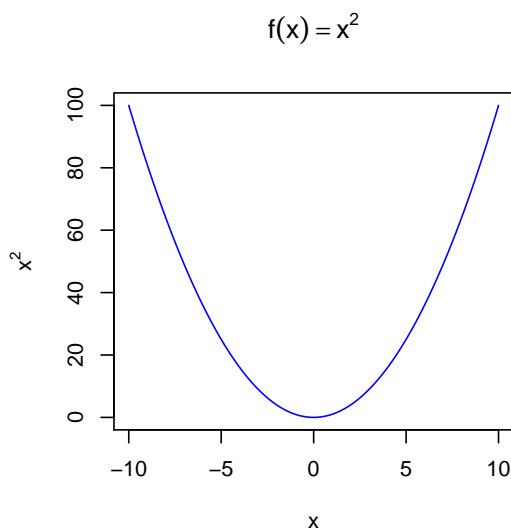
```
> x <- seq(-10, 10, length.out = 1000)
> exp(1) # Euler's number
[1] 2.718
```



```

> par(mfrow = c(2, 2), mai = c(1.02, 1.02, 0.82, 0.42))
> plot(x, x^2, type = "l", col = "blue", xlab = "x", ylab = expression(x^2),
+      main = expression(f(x) == x^2))
> plot(x, -x^2, type = "l", col = "blue", xlab = "x", ylab = expression(-x^2),
+      main = expression(f(x) == -x^2))
> plot(x, exp(-x^2), type = "l", col = "blue", xlab = "x", ylab = expression(e^{
+      -x^2
+ }), main = expression(f(x) == e^{
+      -x^2
+ }), ylim = c(0, 1))
> plot(x, exp(-x^2/2), type = "l", col = "blue", xlab = "x", ylab = expression(e^{
+      -~frac(x^2, 2)
+ }), main = expression(f(x) == e^{
+      -~frac(x^2, 2)
+ }), ylim = c(0, 1))

```



```
> par(mfrow = c(1, 1), mai = c(1.02, 0.82, 0.82, 0.42))
```

The initial fraction $\frac{1}{\sqrt{2\pi}}$ is simply a *normalizing constant*:

- we want the area under the probability density curve to be 1, since we have 1 unit of probability to spread over the x values
- but the area under the $e^{-\frac{x^2}{2}}$ curve is greater than that

```
> f <- function(x) {
+   exp(-x^2/2)
+ }
> integrate(f, -Inf, Inf)
```

```
2.507 with absolute error < 0.00023
```

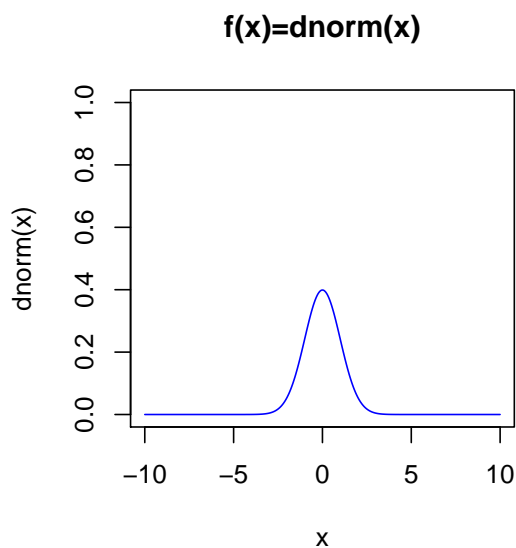
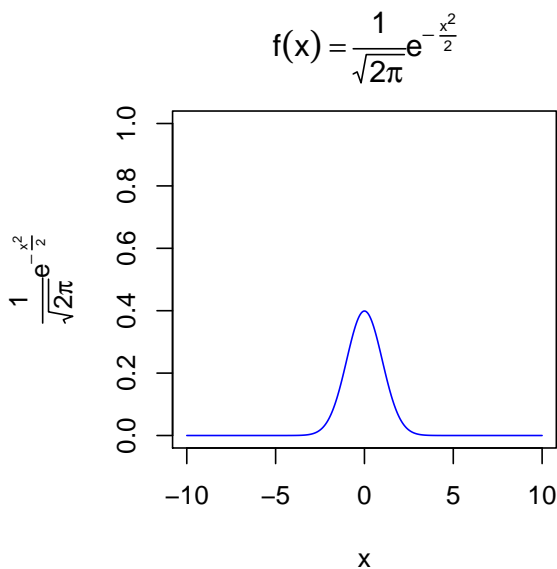
- so we need to ‘normalize’ that area, i.e., scale it down so that it is 1 unit
- it turns out that scaling down (dividing) the values of $e^{-\frac{x^2}{2}}$ by $\sqrt{2\pi}$ yields an area of 1
- this can be proved, but we will only show this numerically here

```
> 1/sqrt(2 * pi)
[1] 0.3989

> f <- function(x) {
+   (1/sqrt(2 * pi)) * exp(-x^2/2)
+ }
> integrate(f, -Inf, Inf)
1 with absolute error < 9.4e-05
```

When we put all this together, the resulting function is the pdf of the standard normal.
In R, we can easily access it with the function `dnorm`:

```
> par(mfrow = c(1, 2), mai = c(1.02, 1.12, 0.82, 0.42))
> plot(x, (1/sqrt(2 * pi)) * exp(-x^2/2), type = "l", col = "blue",
+   xlab = "x", ylab = expression(frac(1, sqrt(2 * pi)) * e^{
+     -frac(x^2, 2)
+   })), main = expression(f(x) == frac(1, sqrt(2 * pi)) * e^{
+     -frac(x^2, 2)
+   })), ylim = c(0, 1))
> plot(x, dnorm(x), type = "l", col = "blue", xlab = "x", ylab = "dnorm(x)",
+   main = "f(x)=dnorm(x)", ylim = c(0, 1))
```



```
> par(mfrow = c(1, 1), mai = c(1.02, 0.82, 0.82, 0.42))
```

References

- Abelson, R.P. (1995). *Statistics as Principled Argument*. L. Erlbaum Associates.
- Baayen, R. Harald (2008). *Analyzing Linguistic Data: A Practical Introduction to Statistics Using R*. Cambridge University Press.
- Braun, J. and D.J. Murdoch (2007). *A First Course in Statistical Programming with R*. Cambridge University Press.
- De Veaux, R.D. et al. (2005). *Stats: Data and Models*. Pearson Education, Limited.
- Diez, D. et al. (2013). *OpenIntro Statistics: Second Edition*. CreateSpace Independent Publishing Platform. URL: <http://www.openintro.org/stat/textbook.php>.
- Faraway, J.J. (2004). *Linear Models With R*. Chapman & Hall Texts in Statistical Science Series. Chapman & Hall/CRC.
- Gelman, A. and J. Hill (2007). *Data Analysis Using Regression and Multilevel/Hierarchical Models*. Analytical Methods for Social Research. Cambridge University Press.
- Gries, S.T. (2009). *Quantitative Corpus Linguistics with R: A Practical Introduction*. Taylor & Francis.
- (2013). *Statistics for Linguistics with R: A Practical Introduction, 2nd Edition*. Mouton De Gruyter.
- Johnson, K. (2008). *Quantitative methods in linguistics*. Blackwell Pub.
- Kachigan, S.K. (1991). *Multivariate Statistical Analysis: A Conceptual Introduction*. Radius Press.
- Kruschke, John K. (2011). *Doing Bayesian Data Analysis: A Tutorial with R and BUGS*. Academic Press/Elsevier.
- Miles, J. and M. Shevlin (2001). *Applying Regression and Correlation: A Guide for Students and Researchers*. SAGE Publications.
- Wright, D.B. and K. London (2009). *Modern regression techniques using R: A practical guide for students and researchers*. SAGE.
- Xie, Yihui (2013). *Dynamic Documents with R and knitr*. Chapman and Hall/CRC.