Lecture: R Math & Programming Refresher

Denis Cohen*

Outline

Thing we are going to cover

Given our time constraints, we will be fairly selective in what we cover today.

Specifically, we will focus on those aspects of the R language that will be immediately relevant in the pursuit of or our learning objectives:

- data types
- object types
- algebraic operators
- probability distributions
- logical and numerical object slicing
- control structures
  - `for` loops
  - `apply`, `lapply`, `sapply`
- functions

Things we are not going to cover: Best practices for reproducible data projects

- R Projects
- Version control (e.g., Git)
- Dependency management (e.g., `renv`)

*Mannheim Centre for European Social Research, University of Mannheim, 68131 Mannheim, Germany. denis.cohen@uni-mannheim.de.

- Start-up customization (`.Renviron`)
- Custom project initialization (`.Rprofile`)
- R Markdown
  - `rmarkdown`
  - `bookdown`
  - `pagedown`
  - `blogdown`
  - ...

Things we are not going to cover: Base R vs tidyverse

- Base R: Collection of native commands
  - historical legacies: not necessarily a "clean" language/syntax
- `tidyverse`:
  - an "opinionated collection of R packages" for data visualization, transformation, tidying, and import
  - more or less uniform syntax
  - Packages: `dplyr`, `ggplot2`, `tibble`, `readr`, `tidyr`, `purrr`
- both are valid
- both are fully compatible with one another
- both are fully compatible with external packages
- you can do amazing things by relying on one, on the other, or on both
- we will use whichever best suits our purposes (here: mostly base R)

Further reading

To learn more about the above, check out my Methods Bites Tutorial Efficient Data Management in R, co-authored with Cosima Meyer, Marcel Neunhoeffer, and Oliver Rittmann.

Structure of this session

Very quick refreshers

1. Data types
2. Arithmetic operators and transformations
3. Object types
4. Object indexing

Moderately quick refreshers

1. R objects vs mathematical objects
2. Probability distributions
3. Control structures and apply functions
4. Programming

Data types

```
# Real numbers ("double")
x <- 9
z <- 13.2

# Integers
y <- 4L

# Characters
start <- "R is so much"
end <- "fun!"
sentence <- paste(start, end, sep = " ")

# Logicals & Boolean operators
2 + 5 == 6              # Does 2 + 5 equal 6?
isFALSE(3 + 4 != 2 + 5) # Is it false that 3 + 4 is not equal to 2 + 5?
3 | 4 >= 2 + 1          # Is 3 OR 4 greater than/equal to 2 + 1?
3 & 7 %in% c(1:5)       # Are 3 AND 7 included in an integer sequence from 1 to 5?
```

Arithmetic operators and transformations

```
x + y     # addition
x - y     # subtraction
x * y     # multiplication
x / y     # division
x ^ y     # exponentiation
log(x)    # natural logarithm
exp(x)    # exponential
sqrt(x)   # square root
a %*% b   # vector/matrix multiplication
t(a)      # vector/matrix transposition
solve(A)  # matrix inversion
```

Object types

Vectors

A vector is a serial listing of data elements of the same type (e.g. `integer`, `double`, `logical`, or `character`).

```
item <- c("flour", "sugar", "potatoes", "tomatoes", "bananas")
price <- c(.99, 1.49, 1.99, 2.79, 1.89)
pricey <- price > 1.5
```

Matrices

A matrix is a rectangular arrangement of data elements of the same type.

```
mat1 <- matrix(seq(-.35, .35, .1), nrow = 2, ncol = 4)
mat1
```

```
##       [,1]  [,2] [,3] [,4]
```

```
## [1,] -0.35 -0.15 0.05 0.25
## [2,] -0.25 -0.05 0.15 0.35
```

Arrays

An array is a multidimensional arrangement of data elements of the same type.

You may think of an array as a generalization of a matrix to any number of dimensions:

- all elements are of the same type
- fixed dimensional structure
- any vector or matrix can be stored as an array using `as.array()`

Lists

Lists allow you to store objects of various classes, various storage types, and various sizes. This can be very useful, e.g., for returning various outputs from a function.

```
my_list <- list()
my_list$char_mat <- matrix(LETTERS, nrow = 2, ncol = 13)
my_list$vectors <- list()
my_list$vectors$num_vec1 <- c(1:2)
my_list$vectors$num_vec2 <- runif(10, min = 0, max = 1)
```

Data Frames

Data frames are lists of variables of equal length, producing a rectangular structure. Unlike matrix columns, data frame columns can be of different storage types and classes.

```
groceries <- data.frame(
  item = c("flour", "sugar", "potatoes", "tomatoes", "bananas"),
  price = c(.99, 1.49, 1.99, 2.79, 1.89)
)


groceries
```

```
##       item price
## 1    flour  0.99
## 2    sugar  1.49
## 3 potatoes  1.99
## 4 tomatoes  2.79
## 5  bananas  1.89
```

Object indexing

Numerical indexing

We can slice vectors, matrices, or arrays using *numerical* indexing.

```
mat1
```

```
##       [,1]  [,2] [,3] [,4]
## [1,] -0.35 -0.15 0.05 0.25
## [2,] -0.25 -0.05 0.15 0.35
```

```
mat1[1, 3:4]
```

```
## [1] 0.05 0.25
```

Indexing by names

When objects contain attributes such as `names()` (`list`, `data.frame`), `rownames()` and `colnames()` (`matrix`), or `dimnames()` (`array`), we can also index by character names:

Naming and slicing a vector

```
names(price) <- item
str(price)
```

```
##  Named num [1:5] 0.99 1.49 1.99 2.79 1.89
##  - attr(*, "names")= chr [1:5] "flour" "sugar" "potatoes" "tomatoes" ...
```

```
price[c("potatoes", "bananas")]
```

```
## potatoes  bananas
##     1.99     1.89
```

Logical indexing

Lastly, we can use logical evaluations to select subsets of elements. Suppose we only want those objects from groceries that are countable (i.e., end with an "s") and cost at least EUR 2.00.

```
# Countable items
groceries$item
```

```
## [1] "flour"    "sugar"    "potatoes" "tomatoes" "bananas"
```

```
is_countable <-
  substr(groceries$item, nchar(groceries$item), nchar(groceries$item)) == "s"

# Pricey items
groceries$price
```

```
## [1] 0.99 1.49 1.99 2.79 1.89
```

```
is_very_pricey <- groceries$price > 2.00

# Find subset
groceries[is_very_pricey & is_countable, ]
```

```
##       item price
## 4 tomatoes  2.79
```

R objects vs mathematical objects

Time for some confusion!

Scalars, vectors, and matrices are frequently used objects in linear algebra.

Unfortunately, they do not translate straightforwardly into numeric R objects

Scalars

A scalar is just a single numeric element (e.g., an integer or a real number).

R does **not** have an object type for scalars. Scalars are stored as numeric vectors of length 1.

```
x <- 1.5
class(x)
```

```
## [1] "numeric"
```

```
typeof(x)
```

```
## [1] "double"
```

```
is.vector(x)
```

```
## [1] TRUE
```

```
length(x)
```

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

Of course, it is not that easy. In math, the inner product of two vectors returns a scalar:

$$s = \mathbf{a}'\mathbf{b} = \begin{bmatrix} 0.5 & 1 & 1.5 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = 0.5 \times 1 + 1 \times 2 + 1.5 \times 3 = 7$$

In R, however,...

```r
a <- c(0.5, 1, 1.5)
b <- c(1, 2, 3)

s <- t(a) %*% b
s
```

```
##      [,1]
## [1,]    7
```

```r
class(s)
```

```
## [1] "matrix" "array"
```

```r
typeof(s)
```

```
## [1] "double"
```

```r
is.vector(s)
```

```
## [1] FALSE
```

```r
is.matrix(s)
```

```
## [1] TRUE
```

```r
dim(s)
```

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

. . . `s` will be stored as a matrix of dimensions 1×1. This can be annoying, because it prevents you from using `s` for simple tasks like scalar multiplication:

$$7 \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} = \begin{bmatrix} 7 & 21 \\ 14 & 28 \end{bmatrix}$$

```r
m <- matrix(1:4, nrow = 2, ncol = 2)
s * m
```

```
## Error in s * m: non-conformable arrays
```

So, to make this work, we need to store `s` as object type `vector` to ensure it behaves like a scalar:

```r
m <- matrix(1:4, nrow = 2, ncol = 2)
s <- as.vector(s)
s * m
```

```
##      [,1] [,2]
## [1,]    7   21
## [2,]   14   28
```

Vectors

Whereas the mathematical concept of a scalar is stored as a `vector` object in R, the mathematical concept of a vector *sometimes* requires the object type `matrix`.

We usually distinguish row and column vectors in linear algebra. Row vectors are matrices of dimensions $1 \times K$ whereas column vectors are matrices of dimensions $K \times 1$.

In R, vectors are just serial listings, without fixed dimensions. R will *mostly* treat numeric `vector` objects like column vectors. Yet sometimes, when (pre-)multiplying other vectors or matrices, R will automatically transpose the vector on the left to ensure conformability:

```r
t(a) %*% b
```

```
##      [,1]
## [1,]    7
```

```r
a %*% b
```

```
##      [,1]
## [1,]    7
```

So `a %*% b`, same as the more literal `t(a) %*% b`, yields the inner product $\mathbf{a}'\mathbf{b}$.

To get the outer product $\mathbf{ab}'$, which results in a matrix of dimensions 3×3, you can do the following:

```
a %*% t(b)
```

```
##      [,1] [,2] [,3]
## [1,]  0.5    1  1.5
## [2,]  1.0    2  3.0
## [3,]  1.5    3  4.5
```

You can ensure unambiguous dimensions by defining vectors as matrices:

```
##      [,1]
## [1,]  0.5
## [2,]  1.0
## [3,]  1.5
```

```
##      [,1] [,2] [,3]
## [1,]  0.5    1  1.5
```

Summary

- Scalars are vectors, and sometimes matrices.
- Vectors are column vectors, but sometimes act like row vectors, and can be defined as matrices.
- Matrices are, in fact, matrices.

Probability distributions

Commands

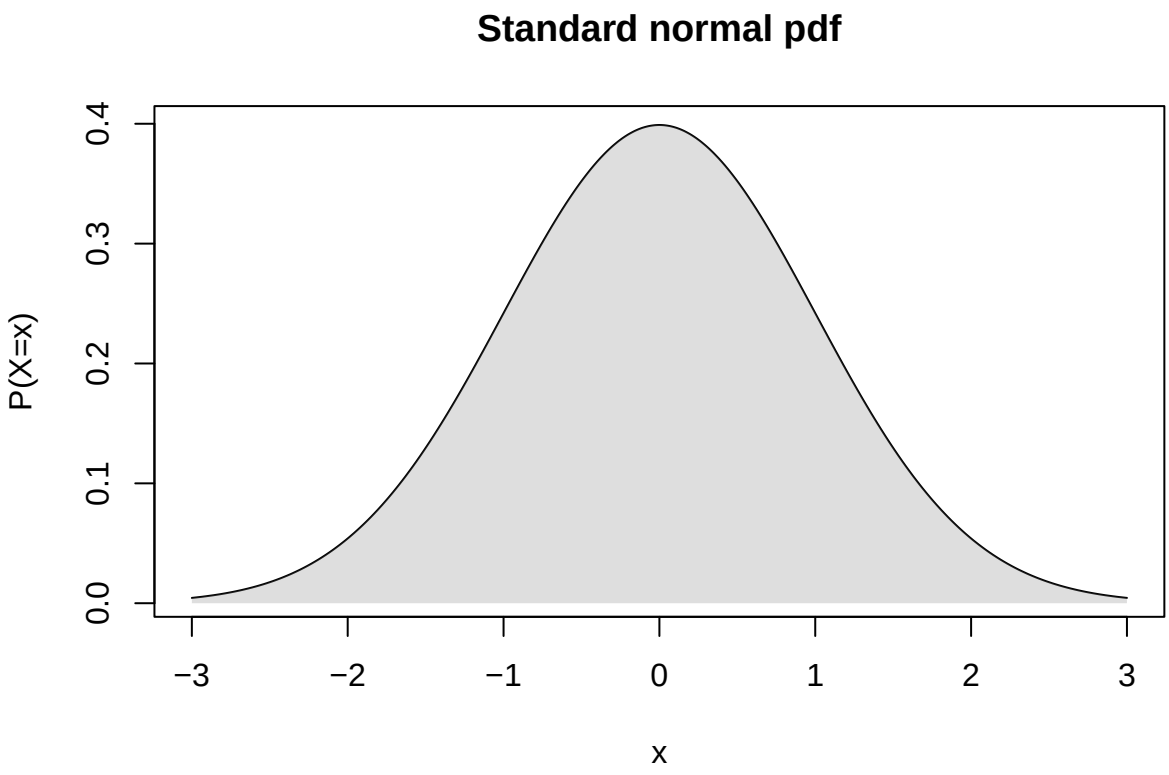
For common probability distributions, R features the following four commands (where `dist` is a placeholder):

1. `ddist(x)`: Probability density/mass function (pdf/pmf). Takes a value x and returns the probability density/mass $P(X = x)$.

2. `pdist(x)`: Cumulative distribution function (CDF). Takes a value x and returns the cumulative probability $P(X \leq x)$.
3. `qdist(q)`: Quantile function or inverse CDF. Takes a cumulative probability $P(X \leq x)$ and returns the corresponding value x.
4. `rdist(n)`: Produces n random draws from the distribution.

The standard normal distribution

Let's illustrate these using everyone's favorite, the standard normal, $\mathrm{N} \sim (0, 1)$:

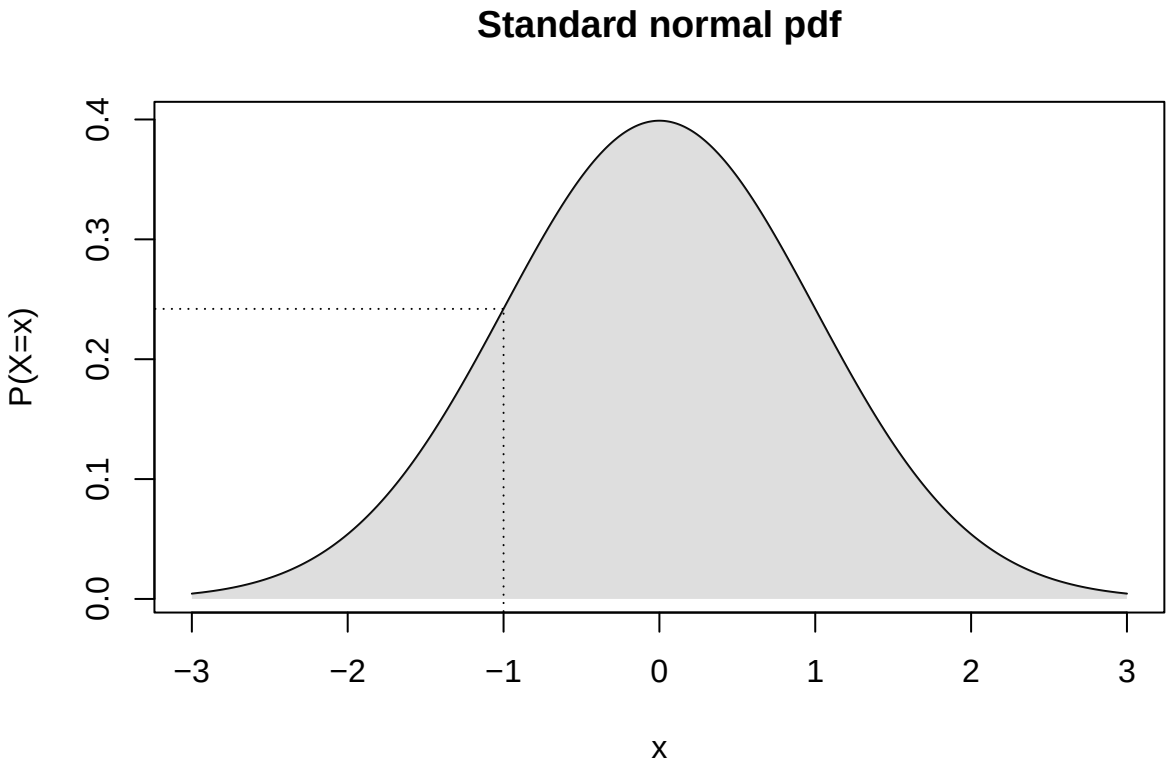


pdf

To get the probability density at any value x, e.g., $x = -1$, run:

```r
dnorm(-1, mean = 0, sd = 1)
```

```
## [1] 0.2419707
```

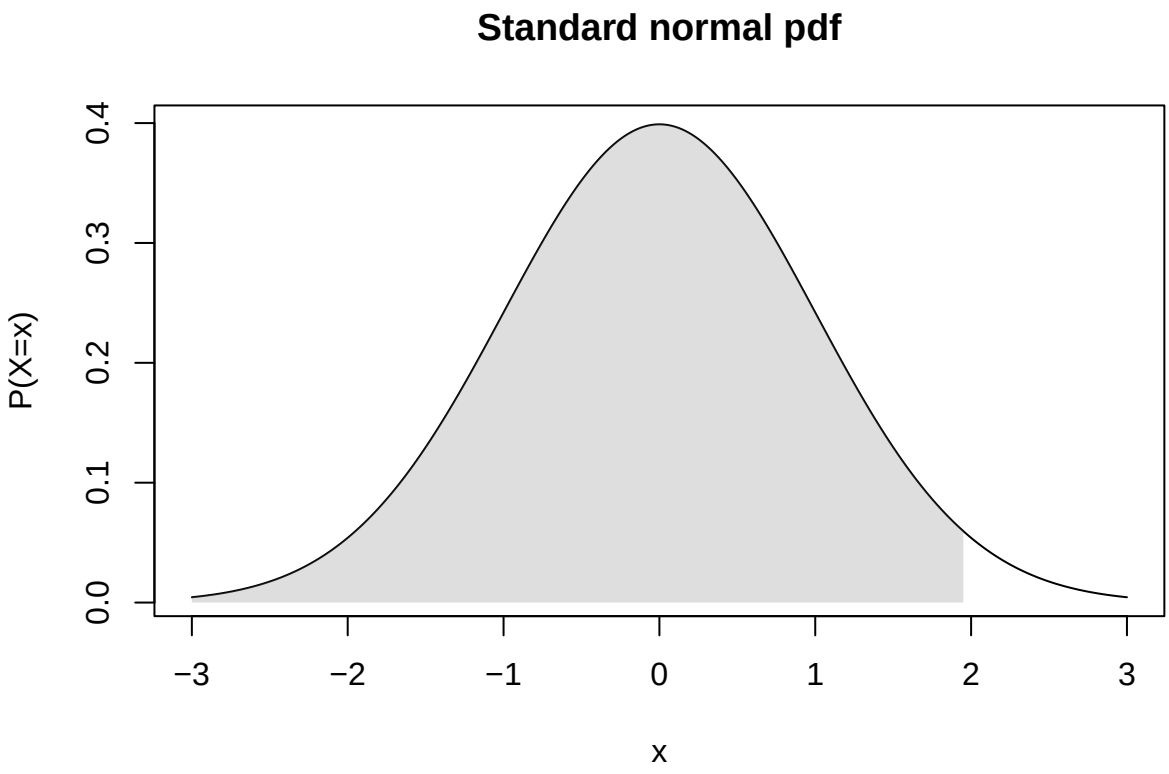


CDF and quantile function

To get the cumulative probability up to any value x, e.g., $x = 1.9599$, run:

```r
pnorm(1.9599, mean = 0, sd = 1)
```

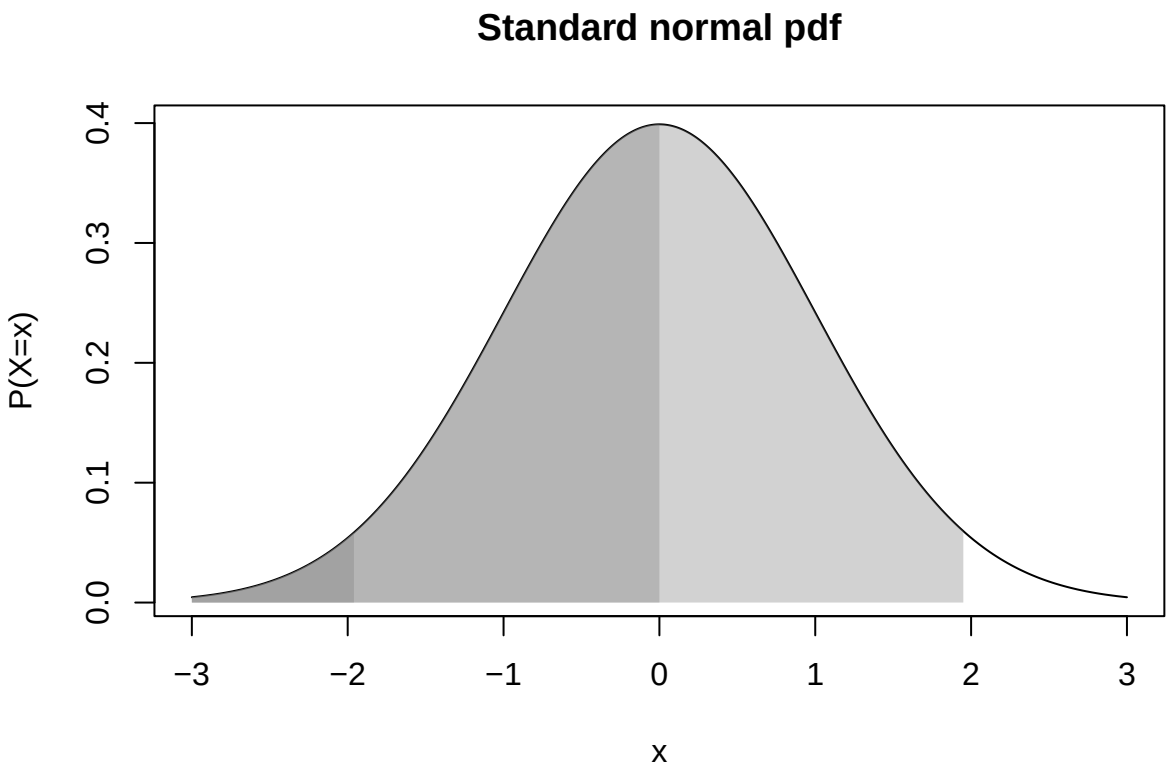
```
## [1] 0.9749963
```



Conversely, use `qnorm` to get the x values for any desired cumulative probability:

```
qnorm(c(.025, .5, .975))
```

```
## [1] -1.959964  0.000000  1.959964
```



Random number generation

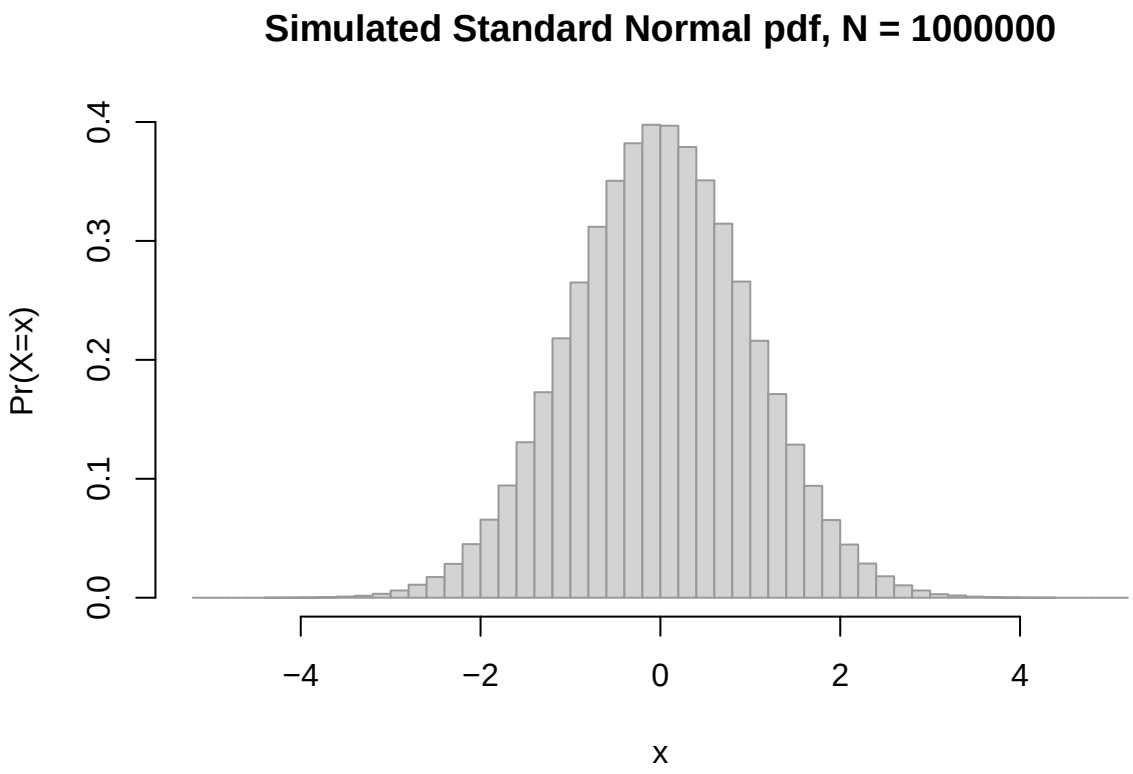
Lastly, to generate random draws from a distribution with specific parameters, use `rnorm()`. The higher the number of draws, the better the chances that the frequency distribution of your random draws will approximate the underlying pdf:

```
# Define number of draws
n_sim <- 1000000L

# Take random draws
x_sim <- rnorm(n_sim, mean = 0, sd = 1)

# Visualize
hist(
  x_sim,
```

```
  main = paste0("Simulated Standard Normal pdf, N = ", n_sim),
  xlab = "x",
  ylab = "Pr(X=x)",
  freq = FALSE,
  breaks = log(n_sim) * 3,
  border = "gray60"
)
```



Control structures and apply functions

Control structures in R

1. if, else, ifelse
2. for
3. while
4. repeat

5. break
6. next
7. return

. . . we will only focus on a few here.

Auxiliary tools for indexing

`seq_len()` and `seq_along()` are very helpful in defining integer sequences:

```
# seq_len()
nrow(groceries)
```

```
## [1] 5
```

```
1:nrow(groceries)
```

```
## [1] 1 2 3 4 5
```

```
seq_len(nrow(groceries))
```

```
## [1] 1 2 3 4 5
```

```
# seq_along
length(item)
```

```
## [1] 5
```

```
seq_along(item)
```

```
## [1] 1 2 3 4 5
```

for-loop

```r
for (i in seq_len(nrow(groceries))) {
  print(paste0("Item ", i, ": ", groceries$item[i]))
}
```

```
## [1] "Item 1: flour"
## [1] "Item 2: sugar"
## [1] "Item 3: potatoes"
## [1] "Item 4: tomatoes"
## [1] "Item 5: bananas"
```

if, else

Let's re-establish which foods are pricey ($\geq$ 1.50EUR) ...

```r
for (i in seq_len(nrow(groceries))) {
  if (groceries$price[i] >= 1.5) {
    print(paste0("Item ", i, ": Sooooo pricey!"))
  } else {
    print(paste0("Item ", i, ": Sooooo cheap!"))
  }
}
```

```
## [1] "Item 1: Sooooo cheap!"
## [1] "Item 2: Sooooo cheap!"
## [1] "Item 3: Sooooo pricey!"
## [1] "Item 4: Sooooo pricey!"
## [1] "Item 5: Sooooo pricey!"
```

We may want to store this information in an object (e.g., a vector or a variable).

As a general rule:

- Whenever possible, initialize an empty *container* of fixed dimensions and subsequently fill it with values.

- Avoid sequentially growing objects by appending elements/rows/columns via `c()`, `rbind()`, or `cbind()`.

```
## Initialize container (here, a vector)
pricey_or_not <- rep(NA, nrow(groceries))


## Fill container
for (i in seq_len(nrow(groceries))) {
  if (groceries$price[i] >= 1.5) {
    pricey_or_not[i] <- "pricey"
  } else {
    pricey_or_not[i] <- "cheap"
  }
}
```

ifelse

For efficiency gains, we may vectorize the operation:

```
ifelse(groceries$price >= 1.5, "pricey", "cheap")
```

```
## [1] "cheap"  "cheap"  "pricey" "pricey" "pricey"
```

lapply, sapply, vapply

Use `lapply()`, `sapply()`, or `vapply()` to apply functions element-wise (e.g., to elements of vector, variables of a data frame, or objects in a list).

- `lapply()` returns a list by default.
- `sapply()` will simplify this list to a vector/matrix/array if the list consists of elements/vectors/matrices of equal lengths/dimensions.
- `vapply()` allows you to specify which type of eligible output you want.

```
lapply(groceries, is.numeric)
```

```
## $item
## [1] FALSE
##
## $price
## [1] TRUE
```

```
sapply(groceries, is.numeric)
```

```
##  item price
## FALSE  TRUE
```

```
vapply(groceries, is.numeric, logical(1))
```

```
##  item price
## FALSE  TRUE
```

```
vapply(groceries, is.numeric, numeric(1))
```

```
##  item price
##     0     1
```

apply

Use `apply()` on matrices or arrays to apply a function across selected dimension(s) of a matrix or array.

Row- and column-means

```
mat1
```

```
##       [,1]  [,2] [,3] [,4]
## [1,] -0.35 -0.15 0.05 0.25
## [2,] -0.25 -0.05 0.15 0.35
```

```
apply(mat1, 1, mean)
```

```
## [1] -0.05  0.05
```

```
apply(mat1, 2, mean)
```

```
## [1] -0.3 -0.1  0.1  0.3
```

Custom functions

You can apply custom functions to all apply functions:

```
count_negative <- function (x) {
  sum(x < 0)
}
apply(mat1, 1, count_negative)
```

```
## [1] 2 2
```

```
apply(mat1, 2, count_negative)
```

```
## [1] 2 2 0 0
```

Programming

Good coding practices

Style Guide

- Adhere to a style guide. It not only ensures consistency, but also allows your readers to focus on *what* you code instead of *how* you code it.
- The most prolific style guide is presented in Advanced R by Hadley Wickham.
- In RStudio, select code and use the shortcut `Ctrl+Shift+A` to automatically reformat your code according to this style guide.

KIS: Keep it simple

- Be clear about inputs and outputs from the start.
- Break down the problem top-down:
  - Split a task into sub-tasks (and, maybe, split these into sub-sub-tasks).
  - Each task should be solved in a self-contained step.
- Then, aggregate solutions bottom-up:
  - Output from completed sub-sub-tasks become inputs to sub-tasks.
  - Output from completed sub-tasks become final output.

DRY: Don't repeat yourself

- Do not perform the same calculation twice; define an object.
- Do not perform the same operation twice; define a function.
- This can not only speed up your code and make it more legible; it also makes debugging *much* easier.

Writing simple functions

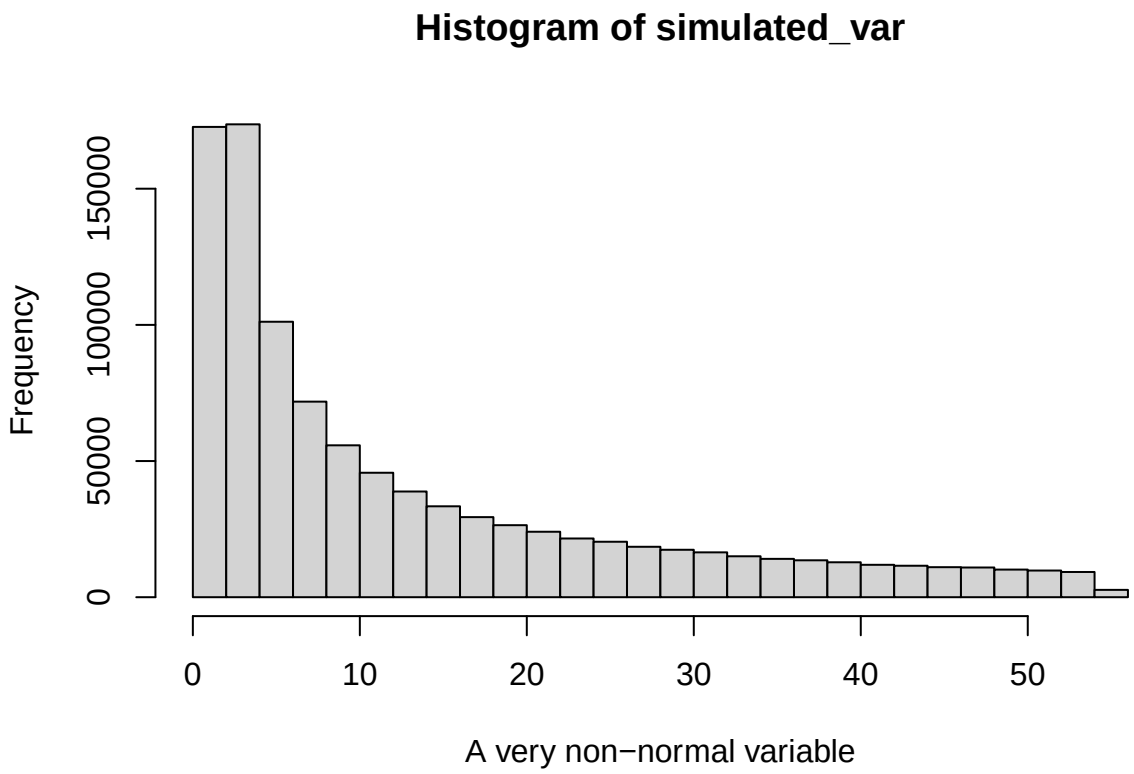
Here, we will program a simple function that allows us to simulate the *Central Limit Theorem* (what does it say again?).

Simulate data

We will use simulated data input:

```r
# Simulate
simulated_var <- exp(runif(1000000, 0,4))


# Plot
hist(
  simulated_var,
  xlab = "A very non-normal variable"
)
```



Writing a function: Good

- adheres to style guide
- uses comments
- meaningful names for function, arguments, and objects
- avoids redundancy and repetition

- gives user control and information over output

```r
simulate_clt <- function(
  data,
  n_means = 1000L,
  n_obs = 1000L,
  value = c("all", "sims", "analytical", "quantiles"),
  seed = 20210325L
) {
  # Set seed
  set.seed(seed)

  # Initialize container
  samples <- matrix(NA, nrow = n_obs, ncol = n_means)

  ## Fill container
  for (i in seq_len(n_means)) {
    samples[, i] <- sample(data, size = n_obs)
  }

  # Obtain sample means
  sample_means <- apply(samples, 2, mean)

  # Calculate normal-approximation 95% CI
  if (value %in% c("all", "analytical")) {
    mean_of_means <- mean(sample_means)
    sd_of_means <- sd(sample_means)
    analytical_ci <- c(
      mean_of_means,
      mean_of_means + qnorm(.025) * sd_of_means,
      mean_of_means + qnorm(.975) * sd_of_means
    )
```

```
  }

  # Calculate quantile-based 95% CI
  if (value %in% c("all", "quantiles")) {
    quantile_ci <- quantile(sample_means, c(.5, .025, .975))
  }

  # Return conditionally
  if (value == "sims") {
    return(sample_means)
  } else if (value == "analytical") {
    return(analytical_ci)
  } else if (value == "quantiles") {
    return(quantile_ci)
  } else {
    return(
      list(sims = sample_means,
           analytical = analytical_ci,
           quantiles = quantile_ci)
      )
  }
}
```

Note: This function does likely not use the most computationally efficient way of solving the problem, but it incorporates many of the concepts discussed so far.

Writing a function: Bad

- does not adhere to style guide
- does not use comments
- no meaningful names for function, arguments, and objects
- lots of redundancy and repetition

- no control and information over output
- no seed → output not reproducible

```
clt.function <- function(data,n1,n2) {c(mean(apply(replicate(n1,sample(data,n2)),2,mean)),
mean(apply(replicate(n1,sample(data,n2)),2,mean))-1.96*sd(apply(replicate(n1,sample(data,n2)),
mean(apply(replicate(n1,sample(data,n2)),2,mean))+1.96*sd(apply(replicate(n1,sample(data,n2)),
```

```
simulate_clt(
  data = simulated_var,
  n_means = 1000L,
  n_obs = 200L,
  value = "quantiles",
  seed = 20210329L
)
```

```
##      50%     2.5%    97.5%
## 13.34851 11.53176 15.30893
```

```
clt.function(
  data = simulated_var,
  n1 = 1000L,
  n2 = 200L
)
```

```
## [1] 13.40816 11.42934 15.30087
```