R-version of the code for "Linear Algebra: Theory, Intuition, Code" by *Mike X Cohen*

Alexander (Sasha) Pastukhov

2021-03-26

Contents

Introduction

This is an R-version of the code for “Linear Algebra: Theory, Intuition, Code” by Mike X Cohen. I have tried to keep the code as close to the original as possible even if it went against the spirit of R. E.g., some loops can be replaced with vectorized operations, tidyverse piping approach, or `apply()/replicate()/purrr::map()`. In most cases, I use `library::` disambiguation to make it easier to understand which function belongs to which package, in addition to importing libraries via `library()`.

Libraries the code relies upon

Matrix calculations

- pracma
- geigen for generalized eigenvalues
- matrixcalc for raising matrix to the power

Tidyverse packages for data wrangling, you can install all relevant packages (including ggplot2) via `install.packages("tidyverse")`.

- dplyr
- tidyr
- reshape2

Graphics

- ggplot2 for plotting
- plotly for 3D surface plot
- patchwork to create a composite figure out of multiple plots
- RColorBrewer for color schemes
- imager for working with images

Code

Chapter 2

Code block 2.1/2.2

```
aScalar <- 4
```

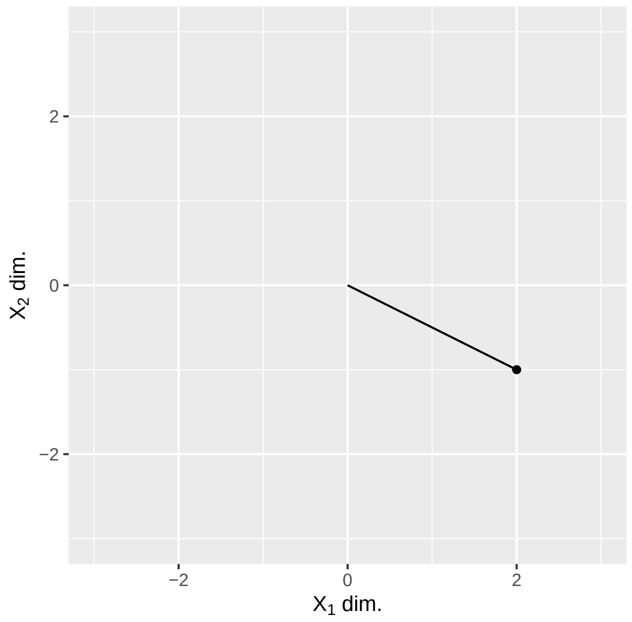
Code block 2.3/2.4

This is a ggplot2 rather than base R version. But *ggplot2* figures do look so much better.

```
library(ggplot2)
```

```
## Warning: package 'ggplot2' was built under R version 4.0.4
```

```
v <- c(2, -1)
ggplot(data=NULL, aes(x = c(0, v[1]), y=c(0, v[2])))  +
  geom_line() +
  geom_point(aes(x=v[1], y=v[2])) +
  scale_x_continuous(name=expression(paste(X[1], " dim.")), limits = c(-3, 3)) +
  scale_y_continuous(name=expression(paste(X[2], " dim.")), limits = c(-3, 3)) +
  coord_equal()
```



Code block 2.5/2.6

In R *atomic* vectors are created via `c()` function but, technically, they are neither column, nor row vectors. Matrix multiplication manual states that it "*multiplies two matrices, if they are conformable. If one argument is a vector, it will be promoted to either a row or column matrix to make the two arguments conformable. If both are vectors of the same length, it will return the inner product (as a matrix).*" At the same time, *transposing* an atomic vector `t(c(...))` transforms it into a single row matrix, thus a row vector, hinting that deep down an outcome of `c()` is a column vector.

To avoid ambiguity, I will use single row and single column matrices for, respectively, row and column vectors.

```r
v1 <- matrix(c(2, 5, 5, 7), nrow = 1) # row vector
v2 <- matrix(c(2, 5, 5, 7), ncol = 1) # column vector
v3 <- matrix(c(2, 5, 5, 7))           # also a column vector
```

Code block 2.7/2.8

```r
v1 <- matrix(c(2, 5, 5, 7), nrow = 1) # row vector
v1a <- t(c(2, 5, 5, 7))               # also a row vector
v2 <- t(v1)                           # column vector
```

Code block 2.9/2.10

For this example, you can also use atomic vectors directly without turning them into a column vector / single column matrix.

```
v1 <- matrix(c(2, 5, 4, 7), ncol=1)
v2 <- matrix(c(4, 1, 0, 2), ncol=1)
v3 <- 4 * v1  - 2 * v2
```

Chapter 3

Code block 3.1/3.2

There is no *explicit* dot product function in base R but there are multiple implementations in various libraries such as pracma used here.

```
v1 <- matrix(c(2, 5, 4, 7), ncol=1)
v2 <- matrix(c(4, 1, 0, 2), ncol=1)
dp <- pracma::dot(v1, v2)
```

However, a matrix multiplication of *atomic* vectors (we can convert a matrix back to an atomic vector via `c()` or `as.vector()`) gives us the dot product (well, the *inner* product, which is why the result is a 1-by-1 matrix).

```
dp <- c(v1) %*% as.vector(v2)
```

Code block 3.3/3.4

```
l1 <- 1
l2 <- 2
l3 <- -3
v1 <- matrix(c(4, 5, 1), ncol=1)
v2 <- matrix(c(-4, 0, -4), ncol=1)
v3 <- matrix(c(1, 3, 2), ncol=1)
l1 * v1 + l2 * v2 + l3 * v3
```

```
##      [,1]
## [1,]   -7
## [2,]   -4
## [3,]  -13
```

Code block 3.5/3.6

To compute the outer product, we must use atomic vectors, thus I've skipped the whole creating-column-vector-as-a-matrix thing.

```r
v1 <- c(2, 5, 4, 7)
v2 <- c(4, 1, 0, 2)
op <- outer(v1, v2)
op <- v1 %o% v2 # alternative call as operation
```

Alternatively, if we do start with vectors as single column matrices…

```r
v1 <- matrix(c(2, 5, 4, 7), ncol=1)
v2 <- matrix(c(4, 1, 0, 2), ncol=1)
op <- outer(c(v1), c(v2))
op <- c(v1) %o% c(v2) # alternative call as operation
```

Code block 3.7/3.8

```r
v1 <- matrix(c(2, 5, 4, 7), ncol=1)
v2 <- matrix(c(4, 1, 0, 2), ncol=1)
v3 <- v1 * v2
```

Code block 3.9/3.10

Please note that you need to explicitly specify the 2-norm via `type="2"`, as the *one norm* is used by default.

```r
v <- matrix(c(2, 5, 4, 7), ncol=1)
vMag <- norm(v, type="2")
v_unit <- v / vMag
```

Chapter 5

Code block 5.1/5.2

There is only one way to transpose a matrix in R: via function t().

```
A <- matrix(runif(n=2*5), nrow=2, ncol=5)
At1 <- t(A)
```

Code block 5.3/5.4

```
I <- diag(4)
O <- matrix(1, nrow=4, ncol=1) # or rep(1, times=4) to create an atomic vector
Z <- matrix(0, nrow=4, ncol=4)
```

Code block 5.5/5.6

```
D <- diag(c(1, 2, 3, 5))  # diagonal matrix
R <- matrix(runif(n = 3 * 4), nrow=3, ncol=4)
d <- diag(R) # diagonal elements
```

Code block 5.7/5.8

In r cbind() and rbind() concatenate, respectively, by column and row.

```
A <- matrix(runif(n = 3 * 5), nrow=3, ncol=5)
B <- matrix(runif(n = 3 * 4), nrow=3, ncol=4)
AB <- cbind(A, B)
```

Code block 5.9/5.10

There are two ways to compute lower and upper triangular parts of a matrix. First, to use `tril()` and `triu()` function from *pracma* library.

```
A <- matrix(runif(n = 5 * 5), nrow=5, ncol=5)
L <- pracma::tril(A)
U <- pracma::triu(A)
```

Alternatively, you can use base R functions lower.tri() and `upper.tri()` that give you a matrix of the same size with *logical* values indicating whether an element belongs to, respectively, lower or upper triangle. Note that by default, the diagonal *is not* included!

```
A <- matrix(runif(n = 5 * 5), nrow=5, ncol=5)

L <- A
# by setting the UPPER triangular part to 0, we get the LOWER triangular part and the
L[upper.tri(L)] <- 0

U <- A
# by setting the LOWER triangular part to 0, we get the UPPER triangular part and the
U[lower.tri(U)] <- 0
```

Code block 5.11/5.12

Note that there is a `toeplitz()` function in *stats* (base R) and `Topelitz()` (note the first capital letter) in *pracma* library. Here, I use base R version.

```
t <- c(1, 2, 3, 4)
T <- stats::toeplitz(t)
H <- pracma::hankel(t, b= c(t[-1], t[1]))
```

```
## Warning in pracma::hankel(t, b = c(t[-1], t[1])): a[n] not equal to b[1], b[1]
## set to a[n].
```

Code block 5.13/5.14

```
l <- 0.01
I <- diag(4)
A <- matrix(runif(4 * 4), nrow=4, ncol=4)
As <- A + l * I
```

Code block 5.15/5.16

Base R does not implement trace function, as it is simply a `sum(diag(M))`. However, you can use `Trace()` from *pracma* library.

```
A <- matrix(runif(4 * 4), nrow=4, ncol=4)
tr <- pracma::Trace(A)
```

Chapter 6

Code block 6.1/6.2

```
M1 <- matrix(runif(n=4*3), nrow=4, ncol=3)
M2 <- matrix(runif(n=3*5), nrow=3, ncol=5)
C <- M1 %*% M2
```

Code block 6.3/6.4

```
A <- matrix(runif(n=2*2), nrow=2, ncol=2)
B <- matrix(runif(n=2*2), nrow=2, ncol=2)
C1 <- A %*% B
C2 <- B %*% A
```

Code block 6.5/6.6

```
M1 <- matrix(runif(n=4*3), nrow=4, ncol=3)
M2 <- matrix(runif(n=4*3), nrow=4, ncol=3)
C <- M1 * M2
```

Code block 6.7/6.8

Note that by default, matrix is constructed by column. To match the code, we need to use `byrow=TRUE` option.

```
A <- matrix(c(1, 2, 3, 4, 5, 6), nrow=2, byrow = TRUE)
c(A)
```

```
## [1] 1 4 2 5 3 6
```

Code block 6.9/6.10

```
A <- matrix(runif(n=4*3), nrow=4, ncol=3)
B <- matrix(runif(n=4*3), nrow=4, ncol=3)
f <- pracma::Trace(t(A) %*% B)
```

Code block 6.11/6.12

```
A <- matrix(runif(n=4*3), nrow=4, ncol=3)
norm(A, type="F")
```

```
## [1] 2.236153
```

Chapter 7

Code block 7.1/7.2

You can use `Rank()` from the *pracma* library. Alternatively, you can use `rankMatrix()` function from the *Matrix* library, which in addition to the rank itself, returns information on the method used to estimate the rank via attributes.

```
A <- matrix(runif(n=3*6), nrow=3, ncol=6)
r1 <- pracma::Rank(A)
r2 <- Matrix::rankMatrix(A)
```

Code block 7.3/7.4

```
s <- runif(n=1)
A <- matrix(runif(n=3*5), nrow=3, ncol=5)
r1 <- pracma::Rank(A)
r2 <- pracma::Rank(s * A)
print(c(r1, r2))
```

```
## [1] 3 3
```

Code block 7.5/7.6

Source code for `Rank()` function from *pracma* library.

```
pracma::Rank
```

```
## function (M)
## {
```

```
##     if (length(M) == 0)
##         return(0)
##     if (!is.numeric(M))
##         stop("Argument 'M' must be a numeric matrix.")
##     if (is.vector(M))
##         M <- matrix(c(M), nrow = length(M), ncol = 1)
##     r1 <- qr(M)$rank
##     sigma <- svd(M)$d
##     tol <- max(dim(M)) * max(sigma) * .Machine$double.eps
##     r2 <- sum(sigma > tol)
##     if (r1 != r2)
##         warning("Rank calculation may be problematic.")
##     return(r2)
## }
## <bytecode: 0x0000000018e9f828>
## <environment: namespace:pracma>
```

Source code for `rankMatrix()` function from *Matrix* library.

```
Matrix::rankMatrix
```

```
## function (x, tol = NULL, method = c("tolNorm2", "qr.R", "qrLINPACK",
##     "qr", "useGrad", "maybeGrad"), sval = svd(x, 0, 0)$d, warn.t = TRUE)
## {
##     stopifnot(length(d <- dim(x)) == 2)
##     p <- min(d)
##     method <- match.arg(method)
##     if (useGrad <- (method %in% c("useGrad", "maybeGrad"))) {
##         stopifnot(length(sval) == p, diff(sval) <= 0)
##         if (sval[1] == 0) {
##             useGrad <- FALSE
##             method <- eval(formals()[["method"]])[[1]]
##         }
##         else {
##             ln.av <- log(abs(sval))
##             diff1 <- diff(ln.av)
##             if (method == "maybeGrad") {
##                 grad <- (min(ln.av) - max(ln.av))/p
##                 useGrad <- !is.na(grad) && min(diff1) <= min(-3,
##                   10 * grad)
##             }
##         }
##     }
##     if (!useGrad) {
##         x.dense <- is.numeric(x) || is(x, "denseMatrix")
##         if ((Meth <- method) == "qr")
##             method <- if (x.dense)
##                 "qrLINPACK"
##             else "qr.R"
##         else Meth <- substr(method, 1, 2)
##         if (Meth == "qr") {
##             if (is.null(tol))
##                 tol <- max(d) * .Machine$double.eps
##         }
##         else {
##             if (is.null(tol)) {
##                 if (!x.dense && missing(sval) && prod(d) >= 100000L)
##                   warning(gettextf("rankMatrix(<large sparse Matrix>, method = '%s'
##                     method), immediate. = TRUE, domain = NA)
##                 stopifnot(diff(sval) <= 0)
##                 tol <- max(d) * .Machine$double.eps
##             }
##             else stopifnot((tol <- as.numeric(tol)[[1]]) >= 0)
##         }
##     }
##     structure(if (useGrad)
##         which.min(diff1)
##     else if (Meth == "qr") {
##         if ((do.t <- (d[1L] < d[2L])) && warn.t)
##             warning(gettextf("rankMatrix(x, method='qr'): computing t(x) as nrow(x)
##         q.r <- qr(if (do.t)
##             t(x)
##         else x, tol = tol, LAPACK = method != "qrLINPACK")
##         if (x.dense && (method == "qrLINPACK"))
##             q.r$rank
##         else {
##             diagR <- if (x.dense)
##                 diag(q.r$qr)
##             else diag(q.r@R)
##             d.i <- abs(diagR)
##             if ((mdi <- max(d.i)) > 0)
##                 sum(d.i >= tol * mdi)
##             else 0L
##         }
##     }
##     else if (sval[1] > 0)
##         sum(sval >= tol * sval[1])
##     else 0L, method = method, useGrad = useGrad, tol = if (useGrad)
##         NA
##     else tol)
## }
## <bytecode: 0x0000000020c8a168>
## <environment: namespace:Matrix>
```

Chapter 8

Code block 8.1/8.2

```
A <- matrix(runif(n=3*4), nrow=3, ncol=4)
pracma::nullspace(A)
```

```
##            [,1]
## [1,] -0.5748233
## [2,] -0.4534361
## [3,]  0.3549781
## [4,]  0.5813470
```

Chapter 9

Code block 9.1/9.2

```
z <- complex(real=3, imaginary=4)
Z <- complex(length.out=2, real=0, imaginary=0)
Z[1] <- 3 + 4i
```

Code block 9.3/9.4

Base R does not have a function that generates random integers on the interval. I will use `sample()` with replacement from the range of integers to replicate this. Also note that I have renamed `i` to `im` to minimize the confusion.

```
r <- sample(-3:4, size=3, replace = TRUE)
im <- sample(-3:4, size=3, replace = TRUE)
Z <- r + im * 1i
print(Z)
```

```
## [1] -3+2i  2+2i  2+0i
```

```
print(Conj(Z))
```

```
## [1] -3-2i  2-2i  2+0i
```

Code block 9.5/9.6

`pracma::dot()` implements the Hermitian dot product.

```
v <- c(0, 1i)
pracma::dot(v, v)
```

```
## [1] 1+0i
```

Chapter 10

Code block 10.1/10.2

You can use `pracma::lu()` function but note that it works only on *square*, positive definite matrices.

```
# Using the square matrix from practice problem b
A <- matrix(c(2, 0, 1, 1, 1, 2, 3, 1, 3), nrow=3, byrow=TRUE)
pracma::lu(A)
```

```
## $L
##      [,1] [,2] [,3]
## [1,]  1.0    0    0
## [2,]  0.5    1    0
## [3,]  1.5    1    1
##
## $U
##      [,1] [,2] [,3]
## [1,]    2    0  1.0
## [2,]    0    1  1.5
## [3,]    0    0  0.0
```

Code block 10.3/10.4

```
A <- matrix(runif(n=2*4), nrow=2, ncol=4)
pracma::rref(A)
```

```
##      [,1] [,2]       [,3]      [,4]
## [1,]    1    0  1.0552476  2.070947
## [2,]    0    1 -0.8104639 -1.259689
```

Chapter 11

Code block 11.1/11.2

```
A <- matrix(runif(n=3*3), nrow=3, ncol=3)
det(A)
```

```
## [1] 0.02450625
```

Chapter 12

Code block 12.1/12.2

In R, you find the inverse via solve(). The latter solves an equation $Ax = b$, omitting the second argument defaults `b = I` and the equation is solved to find the inverse. I have added round() to make it easier to see that AA^-1 produces an identity matrix.

```
A <- matrix(rnorm(n=3*3), nrow=3, ncol=3)
Ai <- solve(A)
round(A %*% Ai, 4)
```

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

Code block 12.3/12.4

```
A <- matrix(rnorm(n=3*3), nrow=3, ncol=3)
Acat <- cbind(A, diag(1, nrow=3, ncol=3))
Ar <- pracma::rref(Acat) # RREF
Ar <- Ar[, 4:6] # keep inverse
Ai <- solve(A)
round(Ar - Ai, 4)
```

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

Code block 12.5/12.6

```
A <- matrix(rnorm(n=5*3), nrow=5, ncol=3)
Al <- solve(t(A) %*% A) %*% t(A)
round(Al %*% A, 4)
```

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

Code block 12.7/12.8

```
A <- matrix(rnorm(n=3*3), nrow=3, ncol=3)
A[2, ] <- A[1, ]
Api <- pracma::pinv(A)
Api %*% A
```

```
##            [,1]      [,2]       [,3]
## [1,]  0.8331182 0.1671025 -0.3333302
## [2,]  0.1671025 0.8326765  0.3337710
## [3,] -0.3333302 0.3337710  0.3342053
```

Chapter 13

Code block 13.1/13.2

```
A <- matrix(c(1, 2, 3, 1, 1, 1), nrow=3, byrow=TRUE)
b <- matrix(c(5.5, -3.5, 1.5), nrow=3)
lsfit(A, b)$coefficients
```

```
## Intercept        X1        X2
##       0.0      -2.5       4.0
```

Code block 13.3/13.4

In R, you first perform QT decomposition via qr() and get an object. Then, you can extract component matrices of the decomposition via qr.Q() and qr.R().

```
A <- matrix(rnorm(n=4*3), nrow=4, ncol=3)
QR <- qr(A)
Q <- qr.Q(QR)
R <- qr.R(QR)
```

Chapter 15

Code block 15.1/15.2

```
A <- matrix(c(2, 3, 3, 2), nrow=2, ncol=2, byrow=TRUE)
eigenA <- eigen(A)
V <- eigenA$vectors
L <- eigenA$values
```

Code block 15.3/15.4

```
n <- 3
A <- matrix(rnorm(n=n^2), nrow=n, ncol=n)
B <- matrix(rnorm(n=n^2), nrow=n, ncol=n)
eigenAB <- geigen::geigen(A, B)
evecs <- eigenAB$vectors
evals <- eigenAB$vals
```

Chapter 16

Code block 16.1/16.2

Note that by default number of left (matrix **U**) and right (matrix **V**) singular vectors is determined as, respectively, `nu = min(n, p)` and `nv = min(n, p)` for the $n \times p$ matrix. Therefore, I included `nv=ncol(A)` to replicate output by Python. Also note that R, like Matlab, return **V**. Also note that singular values attribute is `d` and it is a vector not a diagonal matrix.

```r
A <- matrix(c(1, 1, 0, 0, 1, 1), nrow=2, ncol=3, byrow=TRUE)
svdA <- svd(A, nv=ncol(A))
U <- svdA$u
s <- svdA$d
V <- svdA$v
```

Code block 16.3/16.4

```r
A <- matrix(rnorm(n=5^2), nrow=5, ncol=5)
s <- svd(A)$d
condnum <- max(s) / min(s)

#compare above with pracma::cond()
print(c(condnum, pracma::cond(A)))
```

```
## [1] 356.9647 356.9647
```

Chapter 17

Code block 17.1/17.2

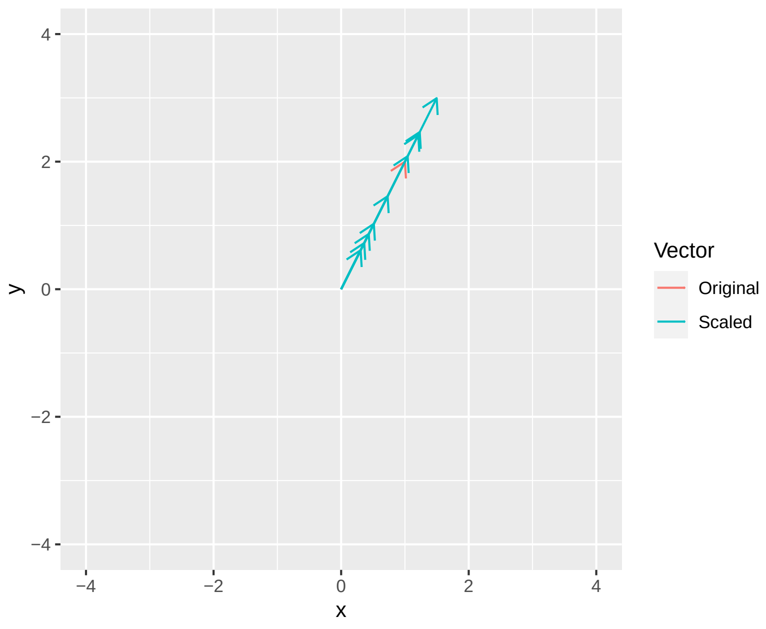
```r
m <- 4
A <- matrix(rnorm(n=m^2), nrow=m, ncol=m)
v <- matrix(rnorm(n=m), nrow=1, ncol=m)
v %*% A %*% t(v)
```

```
##            [,1]
## [1,] -0.6126505
```

Solutions for code challenges

Chapter 2

```r
library(ggplot2)
v <- c(1, 2)
s <- rnorm(10)

all_vectors <- data.frame(xend = c(v[1], v[1] * s),
                          yend = c(v[2], v[2] * s),
                          Vector = c("Original", rep("Scaled", length(s))),
                          x = 0,
                          y = 0)

ggplot(data=all_vectors, aes(x=x, y=y, xend=xend, yend=yend, color=Vector)) +
  geom_segment(arrow= arrow(length = unit(0.03, "npc"))) +
  xlim(-4, 4) +
  ylim(-4, 4) +
  coord_equal()
```



Chapter 3

Exercise 1

```
v1 <- c(1, 2, 3, 4, 5)
v2 <- c(2, 3, 4, 5, 6)
v3 <- c(3, 4, 5, 6, 7)
w <- c(-1, 3, 2)
result <- v1 * w[1] + v2 * w[2] + v3* w[3]
```

Exercise 2

```
v <- c(7, 4, -5, 8, 3)
o <- rep(1, length(v))
ave <- pracma::dot(v, o) / length(v)
```

Exercise 3

```
v <- c(7, 4, -5, 8, 3)
w <- runif(length(v))
wAve <- pracma::dot(v, w / sum(w))
```

Chapter 5

Exercise 1

```
A <- matrix(runif(n=4*2), nrow=4, ncol=2)
B <- matrix(runif(n=4*2), nrow=4, ncol=2)
C <- matrix(NA, nrow=2, ncol=2)
for(coli in 1:2){
  for(colj in 1:2){
    C[coli, colj] <- pracma::dot(A[, coli], B[, colj])
  }
}
```

Exercise 2

```
A <- matrix(runif(n=4*4), nrow=4, ncol=4)
Al <- pracma::tril(A)
S <- Al + t(Al)
```

Exercise 3

```
D <- matrix(0, nrow=4, ncol=8)
for(d in 1:min(dim(D))){
  D[d, d] <- d
}

# or
D <- diag(1:4, nrow=4, ncol=8)
```

Chapter 6

Exercise 1

```
A <- matrix(runif(n=2*4), nrow=2, ncol=4)
B <- matrix(runif(n=4*3), nrow=4, ncol=3)
C1 <- matrix(0, nrow=2, ncol=3)
for(i in 1:4){
  C1 <- C1 + outer(A[, i], B[i, ])
}
C1 - A %*% B
```

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

Exercise 2

```
D <- diag(1:4)
A <- matrix(runif(n=4*4), nrow=4, ncol=4)
C1 <- D * A
C2 <- D %*% A
print(diag(C1))
```

```
## [1] 0.0680205 1.9534080 0.2909992 2.7443596
```

```
print(diag(C2))
```

```
## [1] 0.0680205 1.9534080 0.2909992 2.7443596
```

Exercise 3

```
A <- diag(runif(3))
C1 <- (t(A) + A) / 2
C2 <- t(A) %*% A
C1 - sqrt(C2)
```

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

Exercise 4

Note that `norm()` requires a matrix as an input, therefore, we convert an atomic vector to a matrix for the norm computation.

```
m <- 5
A <- matrix(runif(n=m*m), nrow=m, ncol=m)
v <- runif(n=m)
LHS <- norm(A %*% v, type = "F")
RHS <- norm(A, type = "F") * norm(matrix(v), type="F")
RHS - LHS # should always be positive
```

```
## [1] 1.466629
```

Chapter 7

Exercise 1

```
A <- matrix(runif(n=9*2), nrow=9, ncol=2)
B <- matrix(runif(n=2*16), nrow=2, ncol=16)
C <- A %*% B
```

Exercise 2

```
Z <- matrix(0, nrow=5, ncol=5)
N <- matrix(runif(5 * 5), nrow=5, ncol=5)
ZN <-  Z + N * .Machine$double.eps * 1e-307
print(pracma::Rank(Z))
```

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

```
print(pracma::Rank(ZN))
```

```
## Warning in pracma::Rank(ZN): Rank calculation may be problematic.
```

```
## [1] 4
```

```
print(norm(ZN, type = "F"))
```

```
## [1] 5.928788e-323
```

Chapter 8

Exercise 1

```
A <- matrix(runif(n=4*3), nrow=4, ncol=3) %*% matrix(runif(n=3*4), nrow=3, ncol=4)
B <- matrix(runif(n=4*3), nrow=4, ncol=3) %*% matrix(runif(n=3*4), nrow=3, ncol=4)
n <- pracma::nullspace(A)
print(B %*% A %*% n) # zeros vector
```

```
##              [,1]
## [1,] 4.440892e-16
## [2,] 2.220446e-16
## [3,] 2.220446e-16
## [4,] 2.775558e-16
```

```
print(A %*% B %*% n) # not zeros vector
```

```
##           [,1]
## [1,] 1.0057757
## [2,] 0.9250043
## [3,] 1.2681747
## [4,] 1.6630496
```

Exercise 2

```
A <- matrix(runif(n=16*9), nrow=16, ncol=9) %*% matrix(runif(n=9*11), nrow=9, ncol=11)
rn <- pracma::nullspace(A)
ln <- pracma::nullspace(t(A))
r <- pracma::Rank(A)
print(ncol(rn) + r)
```

```
## [1] 11
```

```r
print(ncol(ln) + r)
```

```
## [1] 16
```

Chapter 9

Exercise 1

Base R does not implement Hermitian transpose directly and you are advised to compute it via `Conj(t(A))`, see *Notes* for [t()](https://stat.ethz.ch/R-manual/R-devel/library/base/html/t.html function.

```r
U <- 0.5 * matrix(c(1+1i, 1-1i, 1-1i, 1+1i), nrow=2, ncol=2)
print(U %*% Conj(t(U))) # Hermitian
```

```
##      [,1] [,2]
## [1,] 1+0i 0+0i
## [2,] 0+0i 1+0i
```

```r
print(U %*% t(U)) # not Hermitian
```

```
##      [,1] [,2]
## [1,] 0+0i 1+0i
## [2,] 1+0i 0+0i
```

Exercise 2

In contrast to Matlab, a complex matrix isSymmetric() only if it is Hermitian.

```r
r <- matrix(runif(n=3*3), nrow=3, ncol=3)
im <- matrix(runif(n=3*3), nrow=3, ncol=3)
A <- r + im * 1i
A1 <- A + Conj(t(A))
A2 <- A %*% Conj(t(A))
isSymmetric(A1)
```

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

```
isSymmetric(A2)
```

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

Chapter 10

Exercise 1

```
A <- matrix(c(2, 0, -3, 3, 1, 4, 1, 0, -1), nrow=3, byrow=TRUE)
x <- matrix(c(2, 3, 4), ncol=1)
b <- A %*% x
```

Exercise 2

```
A <- matrix(runif(n=3*6), nrow=3, ncol=6)
pracma::rref(A)
```

```
##      [,1] [,2] [,3]      [,4]      [,5]      [,6]
## [1,]    1    0    0  9.101523 -5.045534 -2.355027
## [2,]    0    1    0 -8.102776  4.831491  2.537638
## [3,]    0    0    1 -7.037939  5.222501  2.502307
```

Chapter 11

Exercise 1

```
A <- matrix(sample(0:10, size=4*4, replace=TRUE), nrow=4, ncol=4)
b <- sample(-10:-1, size=1)
print(det(b * A))
```

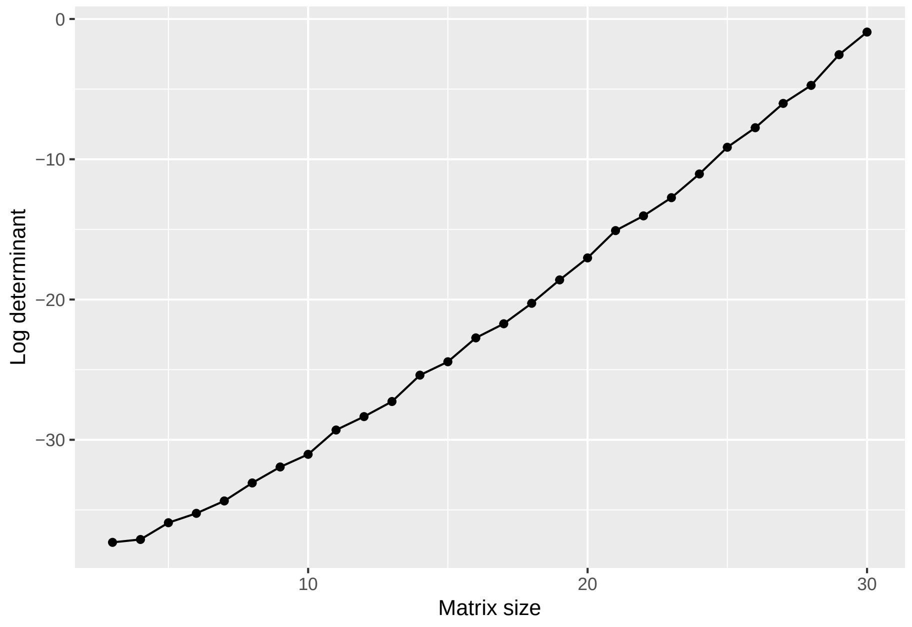
```
## [1] 225
```

```
print(b^nrow(A) * det(A))
```

```
## [1] 225
```

Exercise 2

```r
library(ggplot2)

ns <- 3:30
iters <- 100
dets <- matrix(0, nrow=length(ns), ncol = iters)

for(ni in 1:length(ns)){
  for(it in 1:iters){
    A <- matrix(rnorm(n=ns[ni]^2), nrow=ns[ni], ncol=ns[ni]) # step 1
    A[, 1] <- A[, 2]  # step 2
    dets[ni, it] <- abs(det(A)) # step 3
  }
}

dets_summary <-
  data.frame(MatrixSize = ns,
             LogDeterminant = log(apply(dets, MARGIN = 1, mean)))

ggplot(data=dets_summary, aes(x=MatrixSize, y=LogDeterminant)) +
  geom_line() +
  geom_point() +
  xlab("Matrix size") +
  ylab("Log determinant")
```



Chapter 12

Exercise 1

```
# create matrix
m <- 4
A <- matrix(rnorm(m^2), nrow=m, ncol=m)
M <- matrix(0, nrow=m, ncol=m)
G <- matrix(0, nrow=m, ncol=m)

# compute minors matrix
for(i in 1:m){
  for(j in 1:m){

    ## select rows and cols
    # implementation matching the original
    rows <- rep(TRUE, m)
    rows[i] <- FALSE
    cols <- rep(TRUE, m)
    cols[j] <- FALSE
    M[i, j] <- det(A[rows, cols])
    # a simpler R-version using negative (excluding) indexing
    M[i, j] <- det(A[-i, -j])

    # compute G
    G[i, j] <- (-1)^(i + j)
  }
}

# compute C
C <- M * G

# compute A
Ainv <- t(C) / det(A)
AinvI <- solve(A)
round(AinvI - Ainv, 4)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    0    0    0    0
## [2,]    0    0    0    0
## [3,]    0    0    0    0
## [4,]    0    0    0    0
```

Exercise 2

I renamed `T` into `TM`, as `T` is a logical `TRUE` in R.

```
# square matrix
A <- matrix(rnorm(5^2), nrow=5, ncol=5)
Ai <- solve(A)
Api <- pracma::pinv(A)
print(round(Ai - Api))
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    0    0    0    0    0
## [2,]    0    0    0    0    0
## [3,]    0    0    0    0    0
## [4,]    0    0    0    0    0
## [5,]    0    0    0    0    0
```

```
# tall matrix
TM <- matrix(rnorm(5*3), nrow=5, ncol=3)
TMl <- solve(t(TM) %*% TM) %*% t(TM)
TMpi <- pracma::pinv(TM)
print(round(TMl - TMpi))
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    0    0    0    0    0
## [2,]    0    0    0    0    0
## [3,]    0    0    0    0    0
```

Chapter 13

Exercise 2

```
m <- 4
n <- 4
A <- matrix(rnorm(n=m*n), nrow=m, ncol=n)
Q <- matrix(0, nrow=m, ncol=n)

for(i in 1:n){
  Q[, i] <- A[, i]

  # orthogonalize
  a <- A[, i] # convenience
  if (i > 1){
    for(j in 1:(i-1)){
      q <- Q[, j] # convenience
      Q[, i] <- Q[, i] - pracma::dot(a, q) / pracma::dot(q, q) * q
    }
  }

  # normalize
  Q[, i] <- Q[, i] / norm(matrix(Q[, i]), type="F")
}

QR <- qr(A)
Q2 <- qr.Q(QR)
```

Chapter 14

Exercise 3

```
# load the data into a table that we convert to a matrix
df <- read.csv("http://sincxpress.com/widget_data.txt", header=FALSE)
data <- as.matrix(df)
```

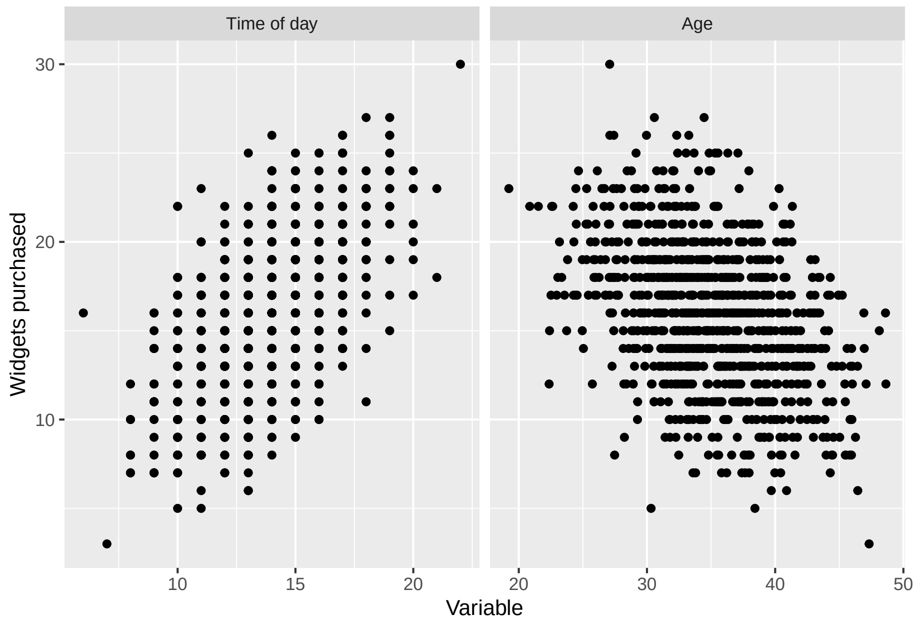
```
# design matrix
X <- cbind(rep(1, nrow(data)), data[, 1:2])
colnames(X) <- c("x1", "x2", "x3")

# outcome variable
y <- data[, 3]

# beta coefficients[]
beta <- lsfit(X, y)$coefficients[1:3]
```

```
## Warning in lsfit(X, y): 'X' matrix was collinear
```

```
# scaled coefficients (intercept not scaled)
betaScaled <- beta / apply(X, MARGIN=2, FUN=sd)
```

Exercise 4

```
library(dplyr)
library(ggplot2)
library(tidyr)

df_long <-
  df %>%
  dplyr::rename("Time of day" = 1, "Age" = 2, "Widgets purchased"=3) %>%
  tidyr::pivot_longer(cols = c("Time of day", "Age"), names_to = "Variable name", values_to = "Va
  dplyr::mutate(`Variable name` = factor(`Variable name`, levels=c("Time of day", "Age")))

ggplot(df_long, aes(x = Variable, y=`Widgets purchased`)) +
  geom_point() +
  facet_grid(.~`Variable name`, scales="free_x")
```



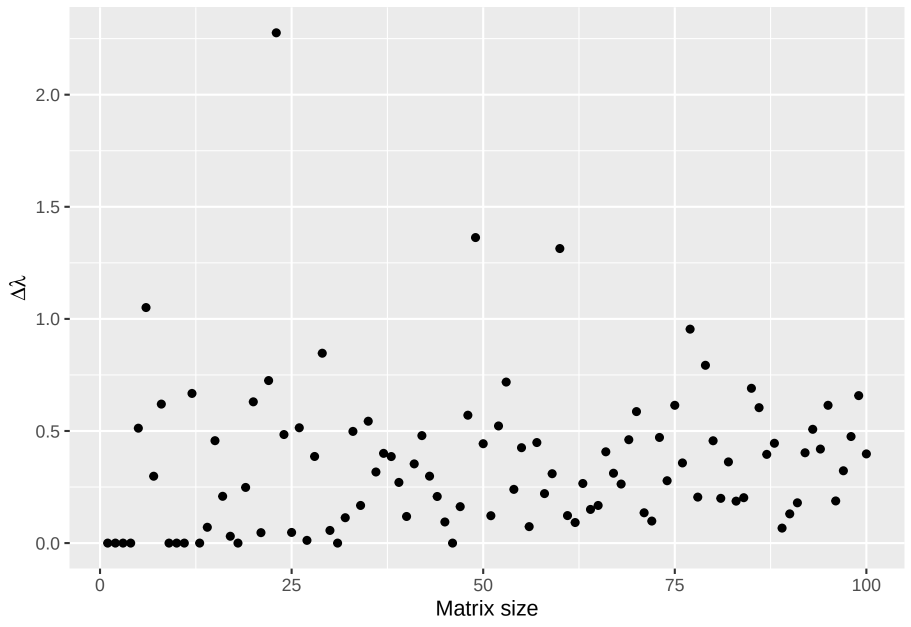
Exercise 5

```r
yHat <- X %*% beta
r2 <- 1 - sum((yHat - y)^2) / sum((y - mean(y))^2)
```

Chapter 15

Exercise 1

```r
avediffs <- rep(0, times=100)
for(n in 1:100){
  A <- matrix(rnorm(n=n^2), nrow=n, ncol=n)
  B <- matrix(rnorm(n=n^2), nrow=n, ncol=n)
  l1 <- geigen::geigen(A, B, symmetric=FALSE, only.values=TRUE)$values
  l2 <- eigen(solve(B) %*% A)$values

  # important to sort eigvals
  l1 <- sort(l1)
  l2 <- sort(l2)
  avediffs[n] <- mean(abs(l1-l2))
}

ggplot(data=NULL, aes(x=1:100, y=avediffs)) +
  geom_point() +
  xlab("Matrix size") +
  ylab(expression(paste(Delta, lambda)))
```



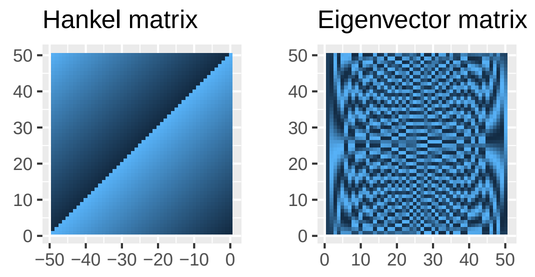
Exercise 2

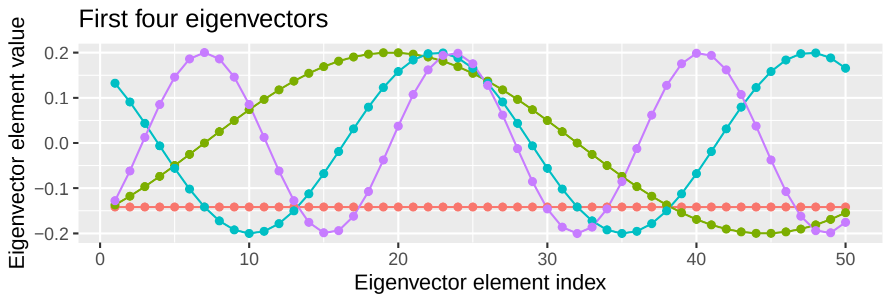
```
A <- diag(1:6)
eigenA <- eigen(A)
L <- eigenA$values
V <- eigenA$vectors
```

Exercise 3

```
library(dplyr)
library(ggplot2)
```

```r
library(patchwork)
library(reshape2)

v <- 1:50
lstrow <- c(v[length(v)], v[-length(v)])
H <- pracma::hankel(v, lstrow)
eigH <- eigen(H)
V <- eigH$vectors[, order(eigH$values, decreasing=TRUE)]

# the matrix
plotH <-
  ggplot(data=reshape2::melt(H), aes(x=1-Var1, y=Var2)) +
  geom_raster(aes(fill=value), show.legend=FALSE) +
  labs(title="Hankel matrix") +
  coord_equal() + xlab("") + ylab("")


# eigenvector matrix
plotV <-
  ggplot(data=reshape2::melt(V), aes(x=Var2, y=Var1)) +
  geom_raster(aes(fill=value), show.legend=FALSE) +
  labs(title="Eigenvector matrix") +
  coord_equal() + xlab("") + ylab("")

# a few eigenvectors
dfV <-
  data.frame(t(V)) %>%
  dplyr::slice_head(n=4) %>%
  dplyr::mutate(VectorIndex = 1:n()) %>%
  tidyr::pivot_longer(cols = c(X1:X50), names_to="Element", values_to="Value") %>%
  dplyr::group_by(VectorIndex) %>%
  dplyr::mutate(ElementIndex = 1:n()) %>%
  dplyr::select(-Element)

plot4 <-
  ggplot(data=dfV, aes(x = ElementIndex, y=Value, color=as.factor(VectorIndex))) +
  geom_line(show.legend=FALSE) +
  geom_point(show.legend=FALSE) +
  xlab("Eigenvector element index") +
  ylab("Eigenvector element value") +
  labs(title = "First four eigenvectors")

(plotH | plotV) / plot4
```





Chapter 16

Exercise 1

In R svd() defaults to "economy" mode. If you want the full matrix, you must specify dimensions for **U** and **V** explicitly.

```
m <- 6
n <- 3
A <- matrix(rnorm(n=m*n), nrow=m, ncol=n)

fullSVD <- svd(A, nu=m, nv=n)
economySVD <- svd(A)

cat(sprintf("Full SVD: (%d, %d), %d, (%d, %d)\n", nrow(fullSVD$u), ncol(fullSVD$u), length(fullSV
```

```
## Full SVD: (6, 6), 3, (3, 3)
```

```
cat(sprintf("Economy : (%d, %d), %d, (%d, %d)\n", nrow(economySVD$u), ncol(economySVD$u), length(
```

```
## Economy : (6, 3), 3, (3, 3)
```

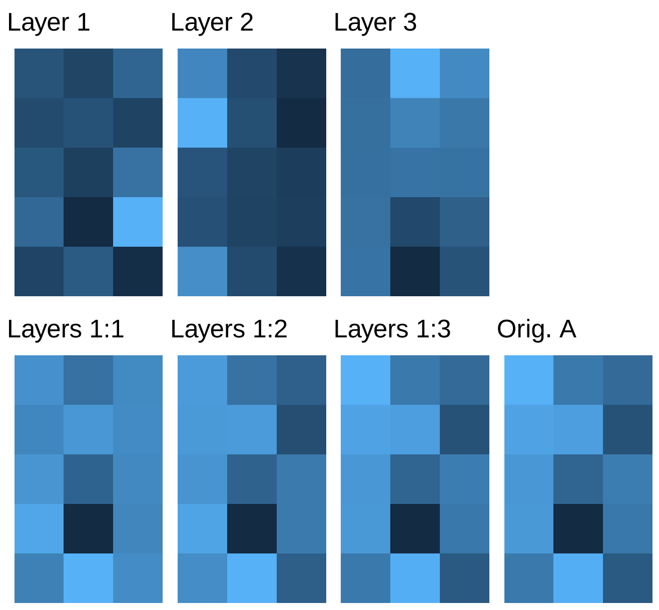
Exercise 2

Note that eigen() sorts eigenvalues and eigenvectors, so sorting is redundant and can be skipped (but I kept it to match the original code).

```r
A <- matrix(rnorm(n=4*5), nrow=4, ncol=5) # matrix
A <- matrix(a, nrow=4, ncol=5, byrow=TRUE)
eigAV <- eigen(t(A) %*% A)
V <-  eigAV$vectors[, order(eigAV$values, decreasing=TRUE)] # sort descent V
eigAU <- eigen(A %*% t(A))
U <-  eigAU$vectors[, order(eigAU$values, decreasing=TRUE)] # sort descent U

# create Sigma
sorted_values <- sort(eigAU$values, decreasing=TRUE)
S <- matrix(0, nrow=nrow(A), ncol=ncol(A))
for(i in 1:length(sorted_values)){
  S[i, i] <- sqrt(sorted_values[i])
}

svdA <- svd(A) # svd
```

Exercise 3

```r
library(ggplot2)
library(patchwork)
library(RColorBrewer)
library(reshape2)


A <- matrix(rnorm(n=5*3), nrow=5, ncol=3)
svdA <- svd(A)
S <- diag(svdA$d) # need Sigma matrix

fill_palette <- colorRampPalette(rev(brewer.pal(11, "Spectral")))
sc <- scale_colour_gradientn(colours = fill_palette(100), limits=c(min(A), max(A)))


one_layer_plots <- list()
lowrank_plots <- list()
for(i in 1:3) {
  onelayer <- outer(svdA$u[, i], svdA$v[i, ]) * svdA$d[i]
  one_layer_plots[[i]] <-
    ggplot(data=reshape2::melt(t(onelayer)), aes(x=Var1, y=Var2, fill=value)) +
    geom_tile(show.legend=FALSE) + theme_void() +
    labs(title=sprintf("Layer %d", i)) +
    coord_equal() + sc

  lowrank <- matrix(svdA$u[, 1:i], ncol=i) %*% S[1:i,1:i] %*% t(svdA$v)[1:i,]
  lowrank_plots[[i]] <-
    ggplot(data=reshape2::melt(t(lowrank)), aes(x=Var1, y=Var2, fill=value)) +
    geom_tile(show.legend=FALSE) + theme_void() +
    labs(title=sprintf("Layers 1:%d", i)) +
    coord_equal() + sc
}

plotA <-
  ggplot(data=reshape2::melt(t(A)), aes(x=Var1, y=Var2, fill=value)) +
  geom_tile(show.legend=FALSE) + theme_void() +
  labs(title="Orig. A") +
  coord_equal() + sc

layout <- "
ABC#
DEFG
"
one_layer_plots[[1]] + one_layer_plots[[2]] + one_layer_plots[[3]] +
  lowrank_plots[[1]] + lowrank_plots[[2]] + lowrank_plots[[3]] + plotA +
  plot_layout(design = layout)
```



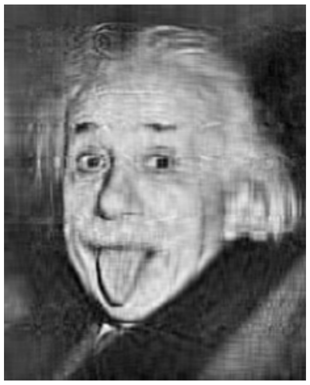
Exercise 4

```r
m <- 6
n <- 16
condnum <- 42

# create U and V from random numbers
U <- qr(matrix(rnorm(m*m), nrow=m, ncol=m))
V <- qr(matrix(rnorm(n*n), nrow=n, ncol=n))

# create singular values vector
s <- seq(condnum, 1, length.out = min(c(m,n)))
S <- diag(s, nrow=m, ncol=n)
# ↓ original code ↓
# S <- matrix(0, nrow=m, ncol=n)
# for(i in 1:min(c(m, n))){
#   S[i, i] <- s[i]
# }

A <- qr.Q(U) %*% S %*% t(qr.Q(V)) # construct matrix
pracma::cond(A)
```

```
## [1] 42
```

Exercise 5

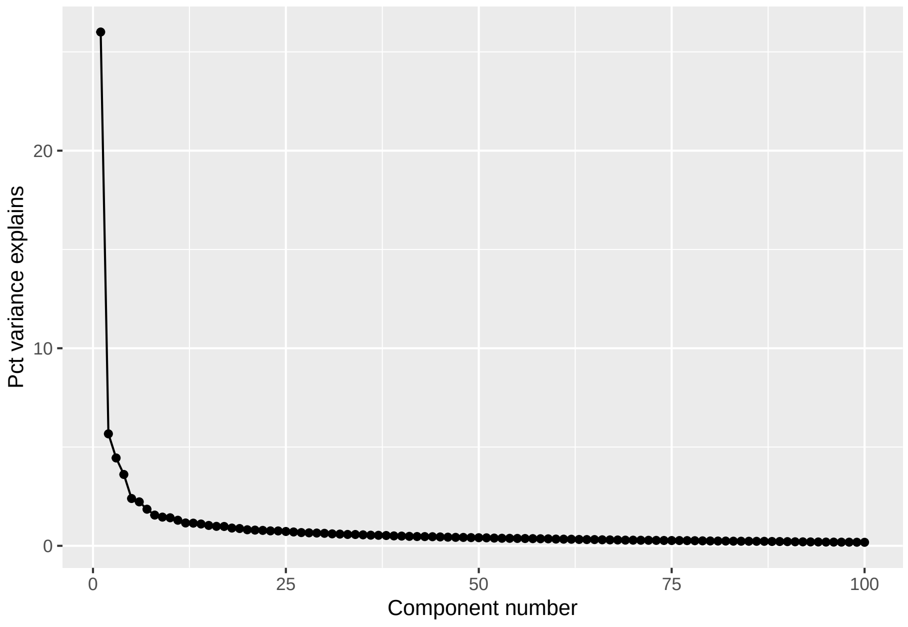
```
library(imager)

rankN <- 20

picture <- imager::load.image('https://upload.wikimedia.org/wikipedia/en/8/86/Einstein_tongue.jpg

pic <- as.matrix(picture)
picSVD <- svd(pic, nu=nrow(pic), nv=ncol(pic))
S <- diag(picSVD$d, nrow=nrow(pic), ncol=ncol(pic))

lowrank <- picSVD$u[, 1:rankN] %*% S[1:rankN, 1:rankN] %*% t(picSVD$v)[1:rankN,]

plot(imager::as.cimg(lowrank), axes=FALSE)
```



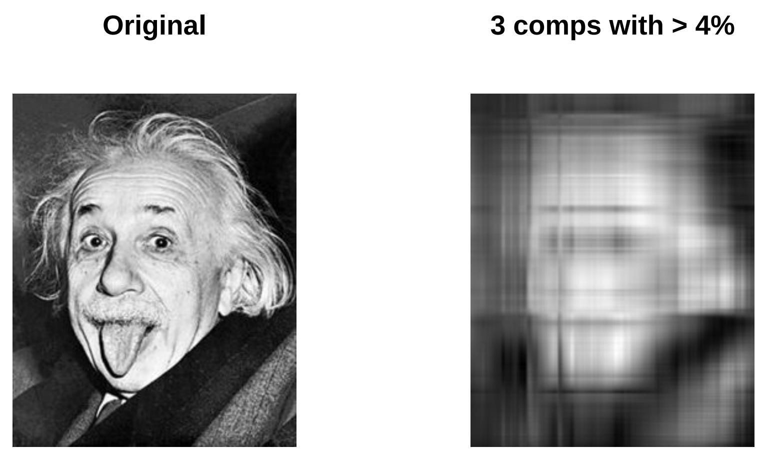
Exercise 6

```
library(imager)

# convert to percent explained
```

```
s <- 100 * picSVD$d / sum(picSVD$d)
ggplot(data=NULL, aes(x=1:100, y=s[1:100])) +
  geom_line() +
  geom_point() +
  xlab("Component number") +
  ylab("Pct variance explains")
```



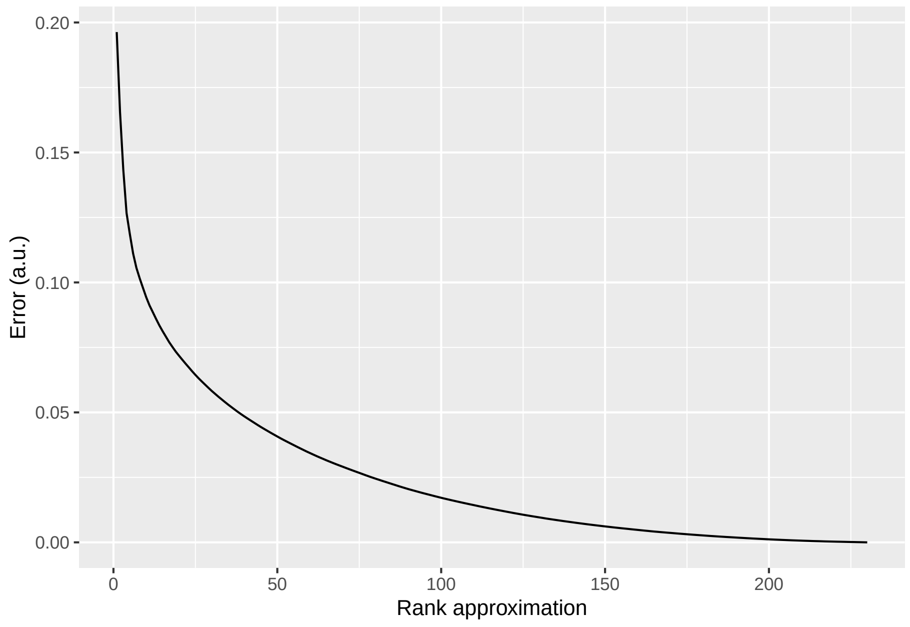
```
thresh <- 4 # threshold in percent
comps <- s > thresh # comps greater than X%
lowrank <- picSVD$u[, comps] %*% S[comps, comps] %*% t(picSVD$v)[comps,]

layout(t(c(1,2)))
plot(picture, axes=FALSE, main="Original")
plot(as.cimg(lowrank), axes=FALSE, main=sprintf("%s comps with > %g%%", sum(comps), th
```



Exercise 7

Note `matrix(picSVD$u[, 1:si], ncol=si)`. Without `matrix(, ncol=si)`, `picSVD$u[, 1:si]` becomes an atomic vector and matrix multiplication breaks down.

```r
library(ggplot2)

RMS <- rep(0, length(s))
for(si in 1:length(s)){
  lowrank <- matrix(picSVD$u[, 1:si], ncol=si) %*% S[1:si, 1:si] %*% t(picSVD$v)[1:si,]
  diffimg <- lowrank - pic
  RMS[si] <- sqrt(mean(diffimg^2))
}

ggplot(data=NULL, aes(x=1:length(RMS), y=RMS)) +
  geom_line() +
  xlab("Rank approximation") +
  ylab("Error (a.u.)")
```



Exercise 8

Reminder, you need to specify `nu` and `nv` to ensure full matrices U and V.

```
library(matrixcalc)

X <- matrix(sample(1:6, size = 4*2, replace = TRUE), nrow=4, ncol=2)
svdX <- svd(X, nu=nrow(X), nv=ncol(X)) # eq. 29
U <- svdX$u
S <- diag(svdX$d, nrow=nrow(X), ncol=ncol(X))
V <- svdX$v

longV1 <- solve(t(U%*%S%*%t(V))%*%U%*%S%*%t(V)) %*% t(U%*%S%*%t(V)) # eq. 30
longV2 <- solve(V%*%t(S)%*%t(U)%*%U%*%S%*%t(V)) %*% t(U%*%S%*%t(V)) # eq. 31
longV3 <- solve(V%*%t(S)%*%S%*%t(V)) %*% t(U%*%S%*%t(V)) # eq. 32
longV4 <- V %*% matrixcalc::matrix.power(t(S) %*% S, -1) %*% t(V)%*%V%*%t(S)%*%t(U) #

MPpinv <- pracma::pinv(X) # eq. 34
```

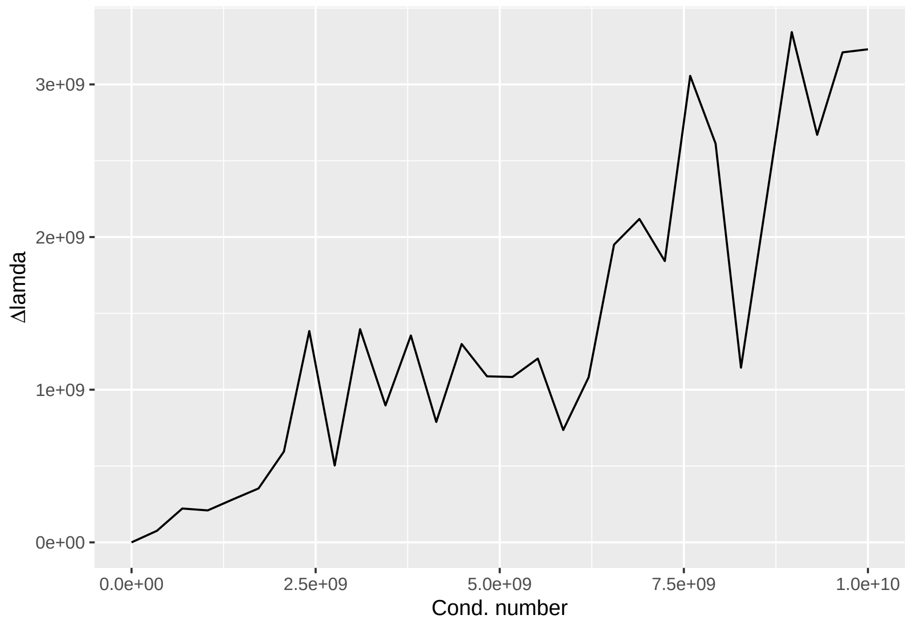
Exercise 9

```
k <- 5
n <- 13
a <- pracma::pinv(matrix(1, nrow=n, ncol=1) * k)
a - 1 / (k * n) # check for zeros
```

```
##                [,1]         [,2]         [,3]         [,4]         [,5]
## [1,] 8.673617e-18 6.938894e-18 6.938894e-18 6.938894e-18 6.938894e-18
##                [,6]         [,7]         [,8]         [,9]        [,10]
## [1,] 6.938894e-18 6.938894e-18 6.938894e-18 6.938894e-18 6.938894e-18
##               [,11]        [,12]        [,13]
## [1,] 6.938894e-18 6.938894e-18 6.938894e-18
```

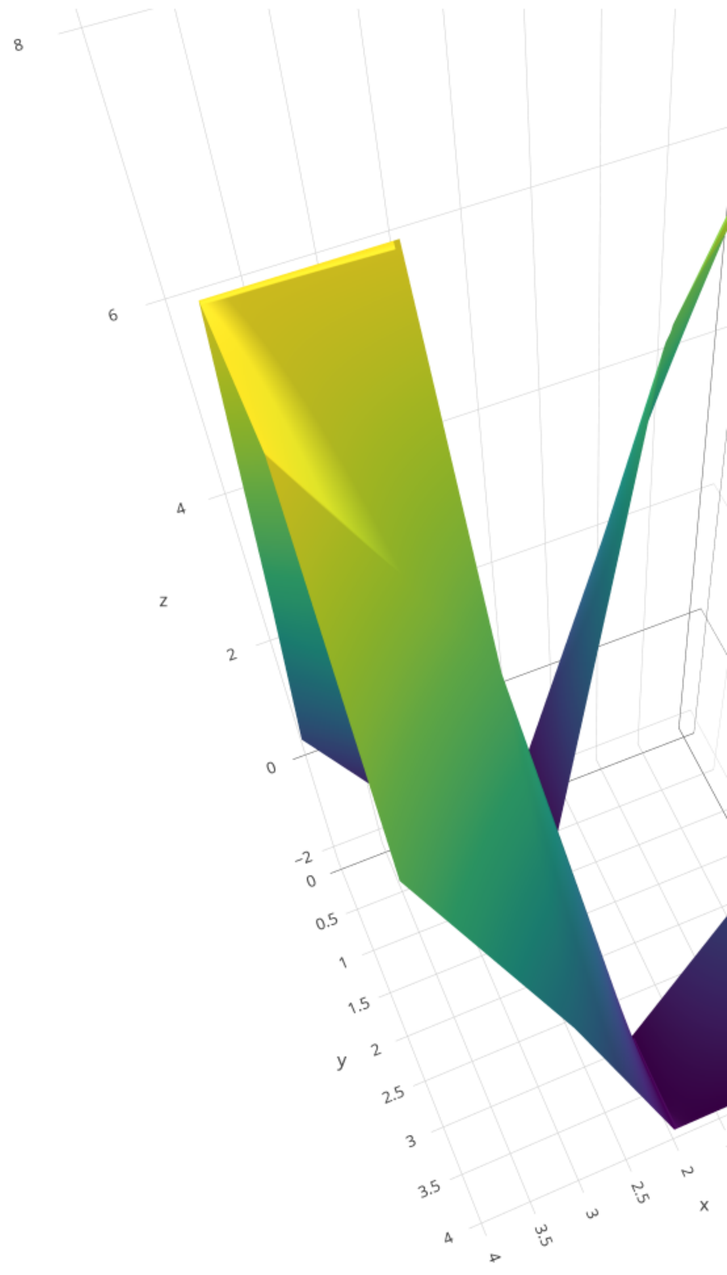
Exercise 10

```
M <- 10
cns <- seq(10, 1e10, length.out=30)
avediffs <- rep(0, length(cns))

# loop over condition numbers
for(condi in 1:length(cns)){
  # create A
  U <- qr.Q(qr(matrix(rnorm(M^2), nrow=M, ncol=M)))
  V <- qr.Q(qr(matrix(rnorm(M^2), nrow=M, ncol=M)))
  S <- diag(cns[condi], nrow=M, ncol=M)
  A <- U %*% S %*% t(V) # construct matrix

  # create B
  U <- qr.Q(qr(matrix(rnorm(M^2), nrow=M, ncol=M)))
  V <- qr.Q(qr(matrix(rnorm(M^2), nrow=M, ncol=M)))
  S <- diag(cns[condi], nrow=M, ncol=M)
  B <- U %*% S %*% t(V) # construct matrix

  # GEDs and sort
  l1 <- sort(eigen(A)$values)
  l2 <- sort(eigen(B)$values)

  avediffs[condi] <- mean(abs(l1-l2))
}

ggplot(data=NULL, aes(x=cns, y=avediffs)) +
  geom_line() +
  xlab("Cond. number") +
  ylab(expression(paste(Delta, lamda)))
```



## Chapter 17 ### Exercise 1 {-}

```
library(plotly)

A <- matrix(c(-2, 3, 2, 8), nrow=2, ncol=2, byrow=TRUE)
vi <- seq(-2, 2, step=0.1)
quadform <- matrix(0, nrow=length(vi), ncol=length(vi))

for(i in 1:length(vi)){
  for(j in 1:length(vi)){
    v <- matrix(c(vi[i], vi[j]), ncol=1)
    quadform[i, j] <- t(v) %*% A %*% v / (t(v) %*% v)
  }
}

plot_ly(z = quadform, type = "surface")
```


8
6
4
z
2
0
−2
0
0.5
1
1.5
y
2
2.5
3
3.5
4
4
3.5
3
2.5
2
x

Exercise 2

```r
n <- 4
nIterations <- 500
defcat <- rep(0, nIterations)

for(iteri in 1:nIterations){
  # create matrix
  A <- matrix(sample(-10:10, size=n^2, replace=TRUE), nrow=n, ncol=n)
  e <- eigen(A)$values
  while (is.complex(e)){
    A <- matrix(sample(-10:10, size=n^2, replace=TRUE), nrow=n, ncol=n)
    e <- eigen(A)$values
  }

  # "zero" threshold (from rank)
  t <- n * pracma::eps(max(svd(A)$d))

  # test definiteness
  if (all(sign(e) == 1)) {
    defcat[iteri] <- 1 # pos. def
  }
  else if (all(sign(e)>-1 & (sum(abs(e)<t)>0))){
    defcat[iteri] <- 2 # pos. semidef
  }
  else if (all(sign(e)<1 & (sum(abs(e)<t)>0))){
    defcat[iteri] <- 4 # neg. semidef
  }
  else if (all(sign(e) == -1)) {
    defcat[iteri] <- 5 # neg. def
  }
  else {
    defcat[iteri] <- 3 # indefinite
  }
}

# print out summary
for(i in 1:5)
{
  print(sprintf("cat %d: %d", i, sum(defcat == i)))
}
```

```
## [1] "cat 1: 1"
## [1] "cat 2: 0"
## [1] "cat 3: 499"
## [1] "cat 4: 0"
## [1] "cat 5: 0"
```

Chapter 18

Exercise 1

```
n <- 200
X <- matrix(rnorm(n*4), nrow=n, ncol=4) # data
X <- apply(X, MARGIN=2, FUN=scale, scale=FALSE) # mean-center
covM <- t(X) %*% X / (n-1) # covariance
stdM <- solve(diag(apply(X, MARGIN=2, FUN=sd))) # stdevs
corM <- stdM %*% t(X) %*% X %*% stdM / (n - 1) # R

# compare ours against R's
print(round(covM - cov(X), 3))
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    0    0    0    0
## [2,]    0    0    0    0
## [3,]    0    0    0    0
## [4,]    0    0    0    0
```

```
print(round(corM - cor(X), 3))
```

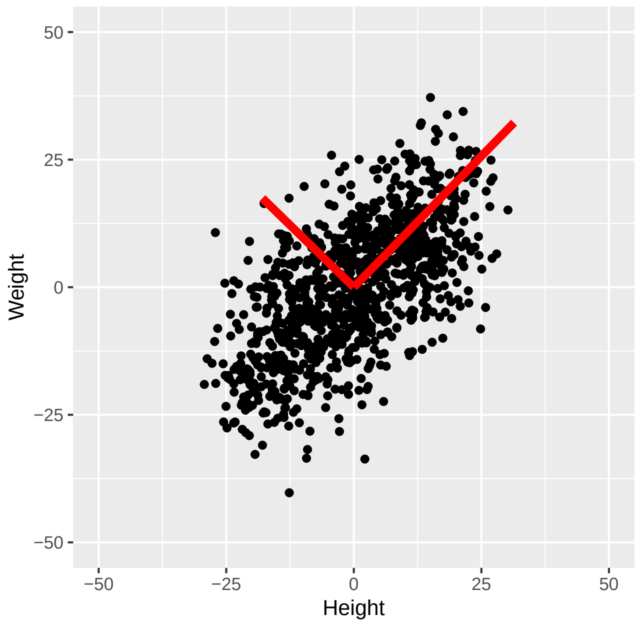
```
##      [,1] [,2] [,3] [,4]
## [1,]    0    0    0    0
## [2,]    0    0    0    0
## [3,]    0    0    0    0
## [4,]    0    0    0    0
```

Chapter 19

Exercise 1

```
library(ggplot2)
```

```
# create data
N <- 1000
h <- rnorm(n=N, mean = seq(150, 190, length.out = N), sd=5)
w <- h * .7 - 50 + rnorm(N, mean=0, sd=10)

# covariance
X <- cbind(h, w)
X <- apply(X, MARGIN=2, scale, scale=FALSE)
C <- t(X) %*% X / (length(h) - 1)

# PCA
eigC <- eigen(C)

# sorting below is redundant in R, as values and vectors are presorted
i <- order(eigC$values, decreasing=TRUE)
V <- eigC$vectors[, i]
eigvals <- eigC$values[i]
eigvals <- 100 * eigvals / sum(eigvals)
scores <- X %*% V # not used, but useful code

# plot data with PCs
ggplot(data=NULL, aes(x = X[, 1], y = X[, 2])) +
  geom_point() +
  geom_segment(aes(x=0, y=0, xend=V[1, 1] * 45, yend=V[2, 1] * 45), color="red", size=
  geom_segment(aes(x=0, y=0, xend=V[1, 2] * 25, yend=V[2, 2] * 25), color="red", size=
  scale_x_continuous(name="Height", limits=c(-50, 50)) +
  scale_y_continuous(name="Weight", limits=c(-50, 50)) +
  coord_equal()
```



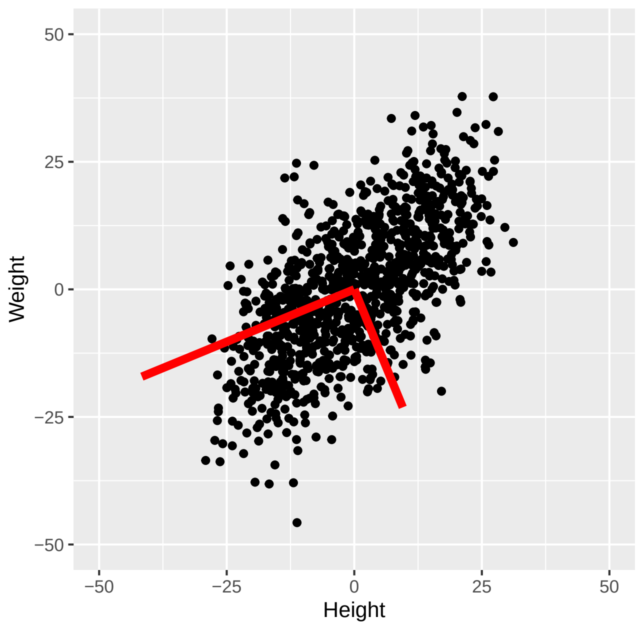
Exercise 2

```r
X <- apply(X, MARGIN=2, scale, scale=FALSE)
svdX <- svd(X, nu=nrow(X), nv=ncol(X))
scores <- X %*% svdX$v
s <- diag(svdX$d)^2 / (nrow(X) - 1)
s <- 100 * s / sum(s) # s == eigvals
```

Exercise 3

```r
library(ggplot2)

# create data
N <- 1000
h <- rnorm(n=N, mean = seq(150, 190, length.out = N), sd=5)
w <- h * .7 - 50 + rnorm(N, mean=0, sd=10)

# covariance
X <- cbind(h, w)
C <- t(X) %*% X / (length(h) - 1)
```

```
# PCA
eigC <- eigen(C)

# sorting below is redundant in R, as values and vectors are presorted
i <- order(eigC$values, decreasing=TRUE)
V <- eigC$vectors[, i]
eigvals <- eigC$values[i]
eigvals <- 100 * eigvals / sum(eigvals)
scores <- X %*% V # not used, but useful code

# now we center the data
X <- apply(X, MARGIN=2, scale, scale=FALSE)

# plot data with PCs
ggplot(data=NULL, aes(x = X[, 1], y = X[, 2])) +
  geom_point() +
  geom_segment(aes(x=0, y=0, xend=V[1, 1] * 45, yend=V[2, 1] * 45), color="red", size=
  geom_segment(aes(x=0, y=0, xend=V[1, 2] * 25, yend=V[2, 2] * 25), color="red", size=
  scale_x_continuous(name="Height", limits=c(-50, 50)) +
  scale_y_continuous(name="Weight", limits=c(-50, 50)) +
  coord_equal()
```



Function references

C

- Complex numbers in R: complex()
- Complex conjugate: Conj()
- Concatenating matrices by columns cbind() or rows rbind()
- Condition of matrix: pracma::cond()
- Cross product: crossprod()

D

- Determinant: det()
- Diagonal: diag()
- Dot product: pracma::dot()

E

- Eigenvalues and eigenvectors: eigen()

G

- Generalized Eigenvalues: giegen::geigen()

H

- Hankel matrix: pracma::hankel()

O

- Outer product: outer()

P

- Power function for matrix: matrixcalc::matrix.power()
- Pseudoinverse: pracma::pinv()

Q

- QR decomposition: qr(). To reconstruct Q, R, or X matrices see qr auxiliaries.

R

- Random numbers from a normal distribution: rnorm()
- Random numbers from a uniform distribution: runif()
- Rank: pracma::Rank()
- Reduced Row Echelon Form: pracma::rref()

S

- Sampling from list of numbers (used in the code to sample integers from a range): sample()
- Singular value decomposition: svd()

T

- Toeplitz matrix: toeplitz()
- Trace: pracma::Trace()
- Transpose: t()
- Triangular parts of matrix. Lower pracma::tril() and upper pracma::triu(). Logical indexes (excluding diagonal by default): lower.tri() and upper.tri().

U

- Upper triangular part of a matrix: pracma::triu() and upper.tri() for logical indexes (excluding diagonal by default).

V

- Vector, creating or converting to an atomic vector: c()
- Vector, converting to an atomic vector: as.vector()