

Exercises Data Science with R

Thomas Kirschstein

Chapter 1

There are quite a lot standard basic mathematical functions. Try to google for R cheat sheets if you are interested in a comprehensive overview.

```
3^2      #1
## [1] 9
sqrt(9) #2
## [1] 3
9^.5
## [1] 3
pi^2     #3
## [1] 9.869604
(abs(3^2-4^2))^.5 #4
## [1] 2.645751
log(exp(4), base = exp(1)) #5
## [1] 4
log(exp(4))
## [1] 4
log(100, base = 10) #6
## [1] 2
log10(100)
## [1] 2
factorial(8) #7
## [1] 40320
exp(factorial(3)) #8
## [1] 403.4288
```

Chapter 2

The chapter intends to focus on writing simple functions in R.

1. Formulate the EOQ formula in R

```
cost_eoq_fun <- function(q, d, co, cl) {
  # returns total cost per period
  # d...demand
  # q...lot size
  # co...ordering cost
  # cl...stock holding cost
  ((1/2)*cl*q)+((d/q)*co)
```

```

}

# test cost function
cost_eoq_fun(d=100, q=20, cl = .1, co = 50)
## [1] 251

eoq_fun <- function(co, d, cl) {
  # return optimal lot size
  # d...demand
  # co...ordering cost
  # cl...stock holding cost
  sqrt((2*co*d)/cl)
}

# optimal lot size
q.star <- eoq_fun(d = 100, cl = .1, co = 50)
# optimal cost
cost_eoq_fun(d=100, q=q.star, cl = .1, co = 50)
## [1] 31.62278

```

2. Derive a function for calculating weighted Euclidean distance between two points.

```

weuc_d2_func <- function(x,y,w) {
  # calculates weighted Euclidean distance between x and y
  # y,x...vectors
  # w.. weight vector
  sqrt(sum(w*(x-y)^2))
}

# test distacne function
weuc_d2_func(x = c(1,2,3), y= c(3,2,1), w=c(1,1,1) )
## [1] 2.828427
# result should be sqrt(8)

```

3. Alter your EOQ function by checking whether all arguments are supplied and stop execution while displaying a error message. -> postponed
4. Formulate a function for the Geometric Poisson density distribution.

```

geom_pois_dens_fun <- function(n, lambda, theta){
  # calculates density value of geometric Poisson distribution
  # n...integer, demand/successes, theta,lambda..parameters
  k.vec <- 1:n
  sum(exp(-lambda)*lambda^k.vec/factorial(k.vec)*(1-theta)^(n-k.vec)*choose(n-1, k.vec-1))
}

# test function
geom_pois_dens_fun(n=3, lambda=.5, theta = 2)
## [1] 0.1642687

```

Chapter 3

Basics on data types and data manipulation

1. Calculate the outer product of two vectors (without `outer()`)

```
x <- 1:5
y <- 10:6
as.matrix(x) %*% t(as.matrix(y))
##      [,1] [,2] [,3] [,4] [,5]
## [1,] 10  9  8  7  6
## [2,] 20 18 16 14 12
## [3,] 30 27 24 21 18
## [4,] 40 36 32 28 24
## [5,] 50 45 40 35 30
```

2. Define a function that calculates the trace of a matrix.

```
trace_func <- function(z){
  # calculates trace of z
  # z...matrix
  sum(diag(z))
}
A <- matrix(rnorm(9) , ncol=3)
trace_func(A)
## [1] 3.878349
```

3. Create a vector containing the first 100 Fibonacci numbers. Most commonly, the Fibonacci numbers are defined recursively by $F_n = F_{n-1} + F_{n-2}$ whereby $F_0 = 0$ and $F_1 = 1$. However, there is also an explicit formulation: $F_n = \sum_{k=0}^{\lfloor \frac{n-1}{2} \rfloor} \binom{n-k-1}{k}$ (check here)

```
fib_num_fun <- function(n){
  # calculate nth Fibonacci number
  # n...number
  k.vec <- 0:floor((n-1)/2)
  sum(choose(n-k.vec-1, k.vec))
}
# vectorize fib_num_fun such that it accepts input vectors
vfib_num_fun <- Vectorize(fib_num_fun)
# doesn't work
fib_num_fun(1:100)
## Warning in 0:floor((n - 1)/2): numerischer Ausdruck hat 100 Elemente: nur erstes
## wird genutzt
## [1] 100
# works
vfib_num_fun(1:100)
##      [1] 1.000000e+00 1.000000e+00 2.000000e+00 3.000000e+00 5.000000e+00
##      [6] 8.000000e+00 1.300000e+01 2.100000e+01 3.400000e+01 5.500000e+01
##     [11] 8.900000e+01 1.440000e+02 2.330000e+02 3.770000e+02 6.100000e+02
##     [16] 9.870000e+02 1.597000e+03 2.584000e+03 4.181000e+03 6.765000e+03
##     [21] 1.094600e+04 1.771100e+04 2.865700e+04 4.636800e+04 7.502500e+04
##     [26] 1.213930e+05 1.964180e+05 3.178110e+05 5.142290e+05 8.320400e+05
##     [31] 1.346269e+06 2.178309e+06 3.524578e+06 5.702887e+06 9.227465e+06
##     [36] 1.493035e+07 2.415782e+07 3.908817e+07 6.324599e+07 1.023342e+08
##     [41] 1.655801e+08 2.679143e+08 4.334944e+08 7.014087e+08 1.134903e+09
##     [46] 1.836312e+09 2.971215e+09 4.807527e+09 7.778742e+09 1.258627e+10
```

```
## [51] 2.036501e+10 3.295128e+10 5.331629e+10 8.626757e+10 1.395839e+11
## [56] 2.258514e+11 3.654353e+11 5.912867e+11 9.567220e+11 1.548009e+12
## [61] 2.504731e+12 4.052740e+12 6.557470e+12 1.061021e+13 1.716768e+13
## [66] 2.777789e+13 4.494557e+13 7.272346e+13 1.176690e+14 1.903925e+14
## [71] 3.080615e+14 4.984540e+14 8.065155e+14 1.304970e+15 2.111485e+15
## [76] 3.416455e+15 5.527940e+15 8.944394e+15 1.447233e+16 2.341673e+16
## [81] 3.788906e+16 6.130579e+16 9.919485e+16 1.605006e+17 2.596955e+17
## [86] 4.201961e+17 6.798916e+17 1.100088e+18 1.779979e+18 2.880067e+18
## [91] 4.660047e+18 7.540114e+18 1.220016e+19 1.974027e+19 3.194043e+19
## [96] 5.168071e+19 8.362114e+19 1.353019e+20 2.189230e+20 3.542248e+20
```

4. Create a matrix containing the all binominal coefficients up to $n = 50$

```
pas <- outer(1:50, 1:50, choose)
```

5. Preference matrices in the Analytical Hierarchy Process (AHP) show a form of (inverted) symmetry. How can you check this in R

The AHP method requires for a matrix $A = \{a_{ij} | i, j = 1, \dots, n\}$ that $a_{i,j} = \frac{1}{a_{j,i}}$. Thereby, diagonal elements are always 1 (i.e., $a_{ii} = 1$)

```
# create 3x3 matrix
A <- matrix(sample(1:10, 9), ncol=3)
# diagonals to 1
diag(A) <- 1
# checks matrix elementwise
A == t(1/A) # -> problem double counting of elements
##      [,1] [,2] [,3]
## [1,] TRUE FALSE FALSE
## [2,] FALSE TRUE FALSE
## [3,] FALSE FALSE TRUE
# Alternative: compare just elements of triangle sub-matrices
A[lower.tri(A)] == 1/A[upper.tri(A)]
## [1] FALSE FALSE FALSE
```

6. Calculate the synthesis of an AHP preference matrix

The synthesis is calculated in two steps: (a) normalizing A by dividing with column sums (b) calculate row means of normalized matrix

```
# convert A (chunk before) into a consistent matrix
A[lower.tri(A)] <- 1/A[upper.tri(A)]
# step (a) of synthesis
dev <- colSums(A) # calculate colSums
A.prime <- t(t(A) / dev) # Alternative a: using that R divides columnwise
A.prime <- A %*% diag(1 / dev) # Alternative b: use matrix calculation
# step (b) of synthesis
rowMeans(A.prime)
## [1] 0.64045946 0.28494659 0.07459395
```

Chapter 4

1. Construct a list with 100 entries. Afterwards, display only entries with odd indices.

```
x <- sample(x = 1:100, size = 100)
l1 <- as.list(x)
l1.odd <- l1[1:100 %% 2 == 1]
```

2. Construct a tibble or data frame consisting of 100 columns and 1 row. Extract only every 3rd column.

```
z <- matrix(sample(x = 1:1000, size = 100), nrow=1)
z <- as.data.frame(z)
z.sub <- z[, col(z) %% 3 == 0]
```

3. Construct a matrix of size 6×6 and fill it by sampling numbers between 1 and 100. Retain indices of all entries ≤ 50 .

```
A <- matrix(sample(x = 1:100, 36), ncol = 6)
which(A <= 50)
## [1] 1 2 3 5 6 9 12 13 14 18 19 20 24 25 30 34 36
```

4. Formulate a function for calculating the moving average over a vector.

```
ma_fun <- function(x, n){
  # calculates moving average
  # x...vector to be averaged, n...time window width
  cx <- c(0, cumsum(x))
  (cx[(n+1):length(cx)] - cx[1:(length(cx) - n)]) / n
}
# test function
ma_fun(x = 1:10, n=4)
## [1] 2.5 3.5 4.5 5.5 6.5 7.5 8.5
```