MTH3045: Statistical Computing: Exercises

Dr. Ben Youngman
b.youngman@exeter.ac.uk
Laver 817; ext. 2314

13/01/2025

1 Chapter 1 exercises

1. Generate a sample of $n = 20$ $N(1, 3^2)$ random variates, and without using `mean()`, `var()` or `sd()` write `R` functions to calculate the sample mean, $\bar{x}$, sample variance, s^2, and sample standard deviation, s, where

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \text{ and } s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2.$$

   Note than `sum()` may be used.

Solution

```
n <- 20
y <- rnorm(n, 1, 3)

# mean
mean2 <- function(x) {
  # function to calculate mean of a vector
  # x is a vector
  # returns a scalar
  sum(x) / length(x)
}

mean2(y)
```

```
## [1] 0.75986
```

```
# check it works
all.equal(mean(y), mean2(y))
```

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

```
# variance
var2 <- function(x) {
  # function to calculate variance of a vector
  # x is a vector
  # returns a scalar
  xbar <- mean2(x)
  sum((x - xbar)^2) / (length(x) - 1)
```

```
}

var2(y)
```

```
## [1] 8.09811
```

```
# check it works
all.equal(var(y), var2(y))
```

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

```
# standard deviation
sd2 <- function(x) {
  # function to calculate standard deviation of a vector
  # x is a vector
  # returns a scalar
  sqrt(var2(x))
}

sd2(y)
```

```
## [1] 2.845718
```

```
# check it works
all.equal(sd(y), sd2(y))
```

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

Note that above for `mean2()` we've not used a `for` loop. A `for` loop can be used, but usually if on cna be avoided, then it should. Then for `var2()` we've re-used `mean2()` and for `sd2()` we've re-used `var2()`. It's often a good idea to re-use functions. In this case, we break the function `sd2()` into various smaller functions, which can often be good practice, as whether the final function is correct can be assessed by whether the simpler functions it comprises are correct.

---

2. Consider computing $\Pr(Z \geq z) = 1 - \Phi(z)$ where $Z \sim \text{Normal}(0, 1)$, or, for short, $Z \sim N(0, 1)$. For $z = 0, 0.5, 1, 1.5, 2, \ldots$ compute this in `R` in three different ways using the following three commands

```
pnorm(z, lower.tail = FALSE)
1 - pnorm(z)
pnorm(-z)
```

and find the lowest value of z for which the three don't give the same answer.

Solution

```
z <- seq(0, 7, by = .5)
cbind(
  z,
  pnorm(z, lower.tail = FALSE),
  1 - pnorm(z),
  pnorm(-z)
)
```

```
##         z
##  [1,] 0.0 5.000000e-01 5.000000e-01 5.000000e-01
##  [2,] 0.5 3.085375e-01 3.085375e-01 3.085375e-01
##  [3,] 1.0 1.586553e-01 1.586553e-01 1.586553e-01
##  [4,] 1.5 6.680720e-02 6.680720e-02 6.680720e-02
##  [5,] 2.0 2.275013e-02 2.275013e-02 2.275013e-02
##  [6,] 2.5 6.209665e-03 6.209665e-03 6.209665e-03
##  [7,] 3.0 1.349898e-03 1.349898e-03 1.349898e-03
##  [8,] 3.5 2.326291e-04 2.326291e-04 2.326291e-04
##  [9,] 4.0 3.167124e-05 3.167124e-05 3.167124e-05
## [10,] 4.5 3.397673e-06 3.397673e-06 3.397673e-06
## [11,] 5.0 2.866516e-07 2.866516e-07 2.866516e-07
## [12,] 5.5 1.898956e-08 1.898956e-08 1.898956e-08
## [13,] 6.0 9.865876e-10 9.865877e-10 9.865876e-10
## [14,] 6.5 4.016001e-11 4.015999e-11 4.016001e-11
## [15,] 7.0 1.279813e-12 1.279865e-12 1.279813e-12
```

Above we see that for $z = 6.0$, the second approximation to the standard normal tail probability doesn't give the same answer as the other two approximations (although the discrepancy is tiny).

---

3. The formula $\text{Var}(Y) = \text{E}(Y^2) - [\text{E}(Y)]^2$ is sometimes called the 'short-cut' variance formula, i.e. a short-cut for $\text{Var}(Y) = \text{E}[Y - \text{E}(Y)]^2$. Compare computing the *biased* version of $\text{Var}(Y)$ using the two formulae above for the samples `y1` and `y2` below.

```
y1 <- 1:10
y2 <- y1 + 1e9
```

Solution

We'll start with a function to calculate the short-cut formula

```
bvar1 <- function(x) {
  # function to calculate short-cut variance
  # x is a vector
  # returns a scalar
  mean(x^2) - mean(x)^2
}
```

and the we'll write a function to calculate the other formula.

```
bvar2 <- function(x) {
  # function to calculate variance
  # x is a vector
  # returns a scalar
  mean((x - mean(x))^2)
}
```

For `y1` we have

```
bvar1(y1)
```

```
## [1] 8.25
```

```
bvar2(y1)
```

```
## [1] 8.25
```

and so both formulae give the same answer, but for `y2` we have

```
bvar1(y2)
```

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

```
bvar2(y2)
```

```
## [1] 8.25
```

and see that they don't. The short-cut formula is clearly wrong, bcause the variance of a sample doesn't change if we add a constant, which is the only difference between `y1` and `y2`. (We'll learn about the cause of this in Chapter 2.)

---

2 Chapter 2 exercises

1. (a) Compute $\pi + \mathrm{e}$, where $\pi = 3.1415927 \times 10^0$ and $\mathrm{e} = 2.7182818 \times 10^0$, using floating point addition in base 10, working to five decimal places.

Solution

As π and e have a common exponent, we sum their mantissas, i.e.

$$\begin{aligned} \pi + \mathrm{e} &= (3.14159 \times 10^0) + (2.71828 \times 10^0) \\ &= 5.85987 \times 10^0 \end{aligned}$$

(b) Now compute $10^6\pi + \mathrm{e}$, using floating point addition in base 10, now working to seven decimal places.

Solution

We first need to put the numbers on a common exponent, which is that of $10^6\pi$.

$$\begin{aligned} 10^6\pi &= 3.1415927 \times 10^6 \\ \mathrm{e} &= 2.7182818 \times 10^0 = 0.0000027 \times 10^6. \end{aligned}$$

Then summing their mantissas gives

$$\begin{aligned} 10^6\pi + \mathrm{e} &= (3.1415927 \times 10^6) + (2.7182818 \times 10^0) \\ &= (3.1415927 \times 10^6) + (0.0000027 \times 10^6) \\ &= (3.1415927 + 0.0000027) \times 10^6 \\ &= 3.1415954 \times 10^6. \end{aligned}$$

(c) What would happen if we computed $10^6\pi + e$ using a base 10 floating point representation, but only worked with five decimal places?

Solution

If we put 2.7182818×10^0 onto the exponent 10^6 we get 0.00000×10^6 to five decimal places, and so e becomes negligible alongside $10^6\pi$ if we only work to five decimal places.

---

2. Write 2π in binary form using single- and double precision arithmetic. [Hint: I recommend you consider the binary forms for π given in the lecture notes, and you might also use `bit2decimal()` from the lecture notes to check your answer.]

Solution

The key here is to note that we want to calculate $\pi \times 2$. As we have a number in the form $S \times (1 + F) \times 2^{E-e}$, then we just need to raise E by one. For both single- and double-precision, this corresponds to changing the last zero in the 0s and 1s for the E term to a one.

```
bit2decimal <- function(x, e, dp = 20) {
# function to convert bits to decimal form
# x: the bits as a character string, with appropriate spaces
# e: the excess
# dp: the decimal places to report the answer to
bl <- strsplit(x, ' ')[[1]] # split x into S, E and F components by spaces
# and then into a list of three character vectors, each element one bit
bl <- lapply(bl, function(z) as.integer(strsplit(z, '')[[1]]))
names(bl) <- c('S', 'E', 'F') # give names, to simplify next few lines
S <- (-1)^bl$S # calculate sign, S
E <- sum(bl$E * 2^c((length(bl$E) - 1):0)) # ditto for exponent, E
F <- sum(bl$F * 2^(-c(1:length(bl$F)))) # and ditto to fraction, F
z <- S * 2^(E - e) * (1 + F) # calculate z
out <- format(z, nsmall = dp) # use format() for specific dp
# add (S, E, F) as attributes, for reference
attr(out, '(S,E,F)') <- c(S = S, E = E, F = F)
out
}
bit2decimal('0 10000001 10010010000111111011011', 127)
```

```
## [1] "6.28318548202514648438"
## attr(,"(S,E,F)")
##           S           E           F
##   1.0000000 129.0000000   0.5707964
```

```
bit2decimal('0 10000000001 1001001000011111101101010100010001000010110100011000', 1023)
```

```
## [1] "6.28318530717958623200"
## attr(,"(S,E,F)")
##           S           E           F
```

```
##     1.0000000 1025.0000000     0.5707963
```

---

3. Find the calculation error in `R` of $b - a$ where $a = 10^{16}$ and $b = 10^{16} + \exp(0.5)$.

Solution

```
a <- 1e16
b <- 1e16 + exp(.5)
b - a
```

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

---

4. Create the following: The row vector

$$\mathbf{a} = (2, 4, 6),$$

the 2×3 matrix

$$\mathbf{B} = \begin{pmatrix} 6 & 5 & 4 \\ 3 & 2 & 1 \end{pmatrix}$$

and a list with first element $\mathbf{a}$ and second element $\mathbf{B}$, and an arbitrary (i.e. with whatever values you like) $5 \times 3 \times 2$ array with a `'name'` attribute that is `'array1'`.
For each of the above, consider whether your code could be simpler.

Solution

```
a <- t(c(2, 4, 6))
B <- matrix(6:1, 2, byrow = TRUE)
l <- list(a, B)
arr <- array(rnorm(30), c(5, 3, 2))
attr(arr, 'name') <- 'array1'
arr
```

```
## , , 1
##
##             [,1]        [,2]       [,3]
## [1,] -0.1665812  0.05562442  0.6567512
## [2,] -0.3405456 -0.55061824 -1.2692691
## [3,]  0.2649471 -0.28015988  1.2724331
## [4,] -0.9606964  0.32101714  0.8776822
## [5,] -0.4040439  0.51386765 -0.6041180
##
## , , 2
##
##             [,1]       [,2]       [,3]
## [1,] -1.4899078 -1.6131559 0.07536778
## [2,]  0.1407264  0.5466237 0.22770119
## [3,]  0.5799854 -1.1506912 0.60413031
## [4,] -1.1158511 -1.7293453 0.59833274
```

```
## [5,] -0.6673627 -0.5074252 0.59765211
##
## attr(,"name")
## [1] "array1"
```

---

5. Produce a $3 \times 4 \times 4 \times 2$ `array` containing Uniform(0, 1) random variates and compute the mean over its second and third margins using `apply(..., ..., means)` and `rowMeans()` or `colMeans()`.

Solution

```
arr <- array(NA, c(3, 4, 4, 2))
arr[] <- runif(prod(dim(arr))) # saves names to get the number right
apply(arr, 2:3, mean)
```

```
##           [,1]      [,2]      [,3]      [,4]
## [1,] 0.6598318 0.3636343 0.5869241 0.4264603
## [2,] 0.4855723 0.5240322 0.4078280 0.5484461
## [3,] 0.2890008 0.5000593 0.3627857 0.4479420
## [4,] 0.4775275 0.3867275 0.4655408 0.3439320
```

```
rowMeans(aperm(arr, c(2, 3, 1, 4)), dims = 2)
```

```
##           [,1]      [,2]      [,3]      [,4]
## [1,] 0.6598318 0.3636343 0.5869241 0.4264603
## [2,] 0.4855723 0.5240322 0.4078280 0.5484461
## [3,] 0.2890008 0.5000593 0.3627857 0.4479420
## [4,] 0.4775275 0.3867275 0.4655408 0.3439320
```

```
colMeans(aperm(arr, c(1, 4, 2, 3)), dims = 2)
```

```
##           [,1]      [,2]      [,3]      [,4]
## [1,] 0.6598318 0.3636343 0.5869241 0.4264603
## [2,] 0.4855723 0.5240322 0.4078280 0.5484461
## [3,] 0.2890008 0.5000593 0.3627857 0.4479420
## [4,] 0.4775275 0.3867275 0.4655408 0.3439320
```

---

6. (a) Create a 3-element `list` of vectors comprising Uniform(0, 1) variates of length 3, 5 and 7, respectively.

Solution

```
lst <- list(runif(3), runif(5), runif(7))
```

(b) Create another list in which the vectors above are sorted into descending order.

Solution

```
lst2 <- lapply(lst, sort, decreasing = TRUE)
```

(c) Then create a vector comprising the minimum of each vector in the list, and another stating which element is the minimum. [Hint: for the latter you may want to consult the 'See Also' part of the `min()` function's help file.]

Solution

```
sapply(lst, min)
```

```
## [1] 0.22358422 0.40101038 0.05418506
```

```
sapply(lst, which.min)
```

```
## [1] 3 5 5
```

---

7. Use a `for()` loop to produce the following. [Hint: the function `paste()` might be useful.]

```
## [1] "iteration 1"
## [1] "iteration 2"
## [1] "iteration 3"
## [1] "iteration 4"
## [1] "iteration 5"
## [1] "iteration 6"
## [1] "iteration 7"
## [1] "iteration 8"
## [1] "iteration 9"
## [1] "iteration 10"
```

Solution

```
for (i in 1:10) print(paste('iteration', i))
```

```
## [1] "iteration 1"
## [1] "iteration 2"
## [1] "iteration 3"
## [1] "iteration 4"
## [1] "iteration 5"
## [1] "iteration 6"
## [1] "iteration 7"
## [1] "iteration 8"
## [1] "iteration 9"
## [1] "iteration 10"
```

---

8. Consider the following two infinite series that represent π.

$$\pi = 4\left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \frac{1}{13} - \cdots\right]$$
$$\pi = 3 + \frac{4}{2 \times 3 \times 4} - \frac{4}{4 \times 5 \times 6} + \frac{4}{6 \times 7 \times 8} - \frac{4}{8 \times 9 \times 10} + \cdots$$

Use a `while()` loop to find which converges to `pi` in `R` to within $\epsilon_m^{1/3}$ using the fewest terms, where ϵ_m is `R`'s machine tolerance.

Solution

Here's the first approximation...

```
quarter_pi <- 1
terms1 <- 1
denom <- 3
multiplier <- -1
my_pi <- 4 * quarter_pi
while(abs(my_pi - pi) > .Machine$double.eps^(1/3)) {
  terms1 <- terms1 + 1
  quarter_pi <- quarter_pi + multiplier / denom
  my_pi <- 4 * quarter_pi
  denom <- denom + 2
  multiplier <- -1 * multiplier
}
terms1
```

```
## [1] 165141
```

...and here's the second approximation...

```
my_pi <- 3
terms2 <- 2
denoms <- c(2, 3, 4)
multiplier <- 1
while(abs(my_pi - pi) > .Machine$double.eps^(1/3)) {
  my_pi <- my_pi + multiplier * 4 / prod(denoms)
  denoms <- denoms + 2
  multiplier <- -1 * multiplier
  terms2 <- terms2 + 1
}
terms2
```

```
## [1] 36
```

---

9. (a) Write a function based on a `for()` loop to calculate the cumulative sum of a vector, `y`, i.e. so that its ith value, y_i say, is

$$y_i = \sum_{j=1}^{i} x_j.$$

Solution

Either of the following two functions are options (although others exist).

```
my_cumsum <- function(x) {
# function 1 to calculate cumulative sum of a vector
# x is a vector
# returns a vector of length length(x)
out <- numeric(length(x))
for (i in 1:length(x)) {
  out[i] <- sum(x[1:i])
}
out
}

my_cumsum2 <- function(x) {
# function 2 to calculate cumulative sum of a vector
# x is a vector
# returns a vector of length length(x)
out <- x
for (i in 2:length(x)) {
  out[i] <- out[i] + out[i - 1]
}
out
}
```

We see that both perform the same calculation.

```
x <- runif(10)
cumsum(x)
```

```
##  [1] 0.8291765 1.6531608 1.7352366 1.8885832 2.3897789 3.0083146 3.4798496 3.528
##  [9] 4.3646237 5.1984644
```

```
my_cumsum(x)
```

```
##  [1] 0.8291765 1.6531608 1.7352366 1.8885832 2.3897789 3.0083146 3.4798496 3.528
##  [9] 4.3646237 5.1984644
```

```
my_cumsum2(x)
```

```
##  [1] 0.8291765 1.6531608 1.7352366 1.8885832 2.3897789 3.0083146 3.4798496 3.528
##  [9] 4.3646237 5.1984644
```

(b) Then benchmark its execution time against `R`'s vectorised function `cumsum()` for summing 1000 iid $Uniform[0, 1]$ random variables.

Solution

```
x <- runif(1e3)
microbenchmark::microbenchmark(
  cumsum(x),
  my_cumsum(x),
```

```
  my_cumsum2(x)
)
```

```
## Unit: nanoseconds
```

---

10. (a) To start a board game, a player must throw three sixes using a conventional die. Write a function to simulate this, which returns the total number of throws that the player has taken.

Solution

```
sixes1 <- function() {
# function to simulate number of throws needed
# to reach three sixes
# n is an integer
# returns total number of throws taken
out <- sample(1:6, 3, replace = TRUE)
while(sum(out == 6) < 3) {
  out <- c(out, sample(1:6, 1))
}
return(length(out))
}
```

(b) Then use 1000 simulations to empirically estimate the distribution of the number of throws needed.

Solution

```
samp1 <- replicate(1e3, sixes1())
table(samp1)
```

```
## samp1
##  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
##  2  9 17 32 33 36 37 57 44 47 42 46 53 41 47 37 47 30 43 34 21 29 21 26 20 11 13
## 34 35 36 37 38 39 40 41 42 43 44 45 47 49 50 52 56 58 61
## 10  7  8  6  6  6  3  7  5  1  2  3  2  4  1  1  1  1  1
```

```
hist(samp1, xlab = 'Number of throws', main = 'Histogram of number of throws')
```

(c) I think you'll agree that this would be a rather dull board game! It is therefore proposed that a player should instead throw two consecutive sixes. Write a function to simulate this new criterion, and estimate its distribution empirically.

Solution

```
sixes2 <- function() {
# function to simulate number of throws needed
```

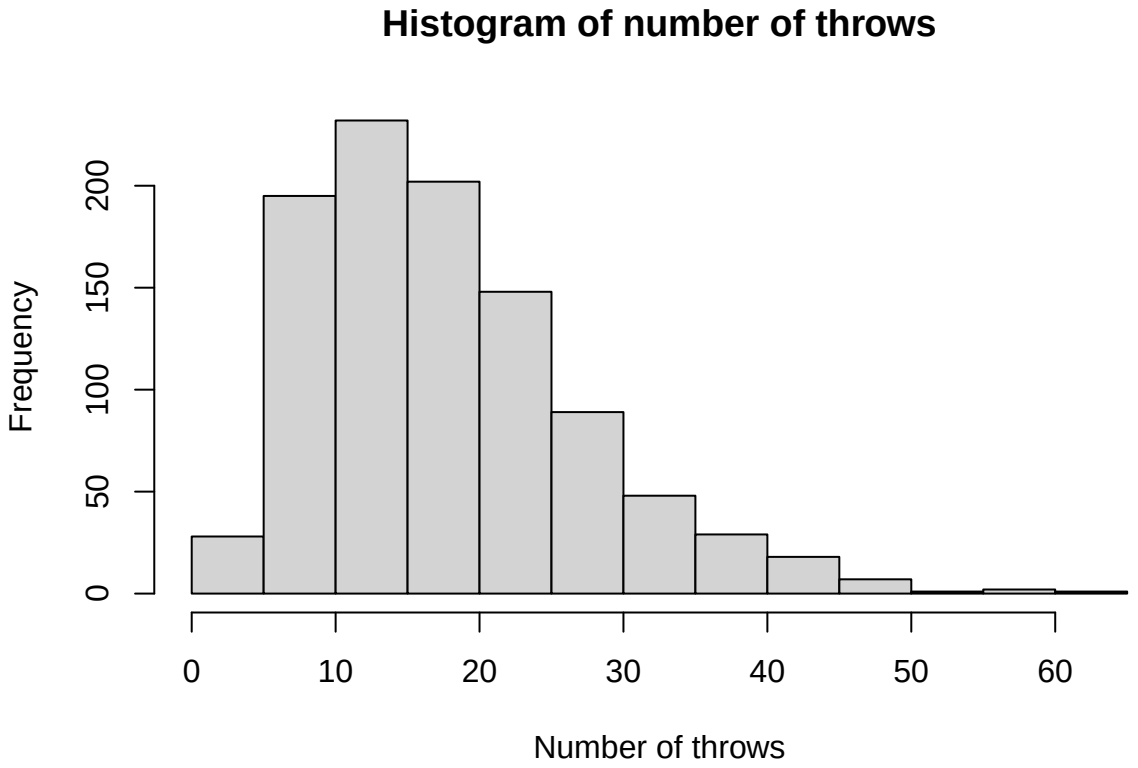


Figure 1: Histogram of empirical distribution of number of throws for starting method 1.

```
# for two consecutive sixes
# n is an integer
# returns total number of throws taken
out <- sample(1:6, 2, replace = TRUE)
cond <- TRUE
while(cond) {
  if (sum(out[1:(length(out) - 1)] + out[2:length(out)] == 12) > 0) {
    cond <- FALSE
  } else {
    out <- c(out, sample(1:6, 1))
  }
}
return(length(out))
}
sixes2()
```

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

(d) Then use 1000 simulations to empirically estimate the distribution of the number of throws needed.

Solution

```
samp2 <- replicate(1e3, sixes2())
table(samp2)
```

```
## samp2
##   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20  21
##  30  17  26  21  18  20  42  17  20  25  20  14  16  12  16  15  12  16   8  20
##  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44
##   8  10  17  14  12  11  12  16  10  16   9  14  10   8  12   6   7   6  14  11
##  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67
##   5  11   9   3   8   7   5   9   6   2   7   4   5   4   9   7   5   9   3   8
##  71  72  73  74  75  76  78  79  80  81  82  83  84  85  86  87  88  89  90  91
##   6   8   6   5   2   8   4   5   1   7   2   5   4   2   2   8   1   4   6   1
##  96  97  98  99 100 101 103 104 105 106 107 108 109 110 111 112 113 115 117 118
##   1   5   3   3   1   1   4   2   2   1   2   3   4   5   1   3   1   4   1   1
## 122 124 125 129 131 133 134 135 139 142 143 144 146 152 153 160 165 168 172 175
##   3   2   1   2   1   1   3   1   3   2   1   2   3   1   1   1   1   1   1   1
## 191 201 202 204 208 213 235 239 327 372
##   1   1   1   1   1   1   1   1   2   1
```

```
hist(samp2, xlab = 'Number of throws', main = 'Histogram of number of throws')
```

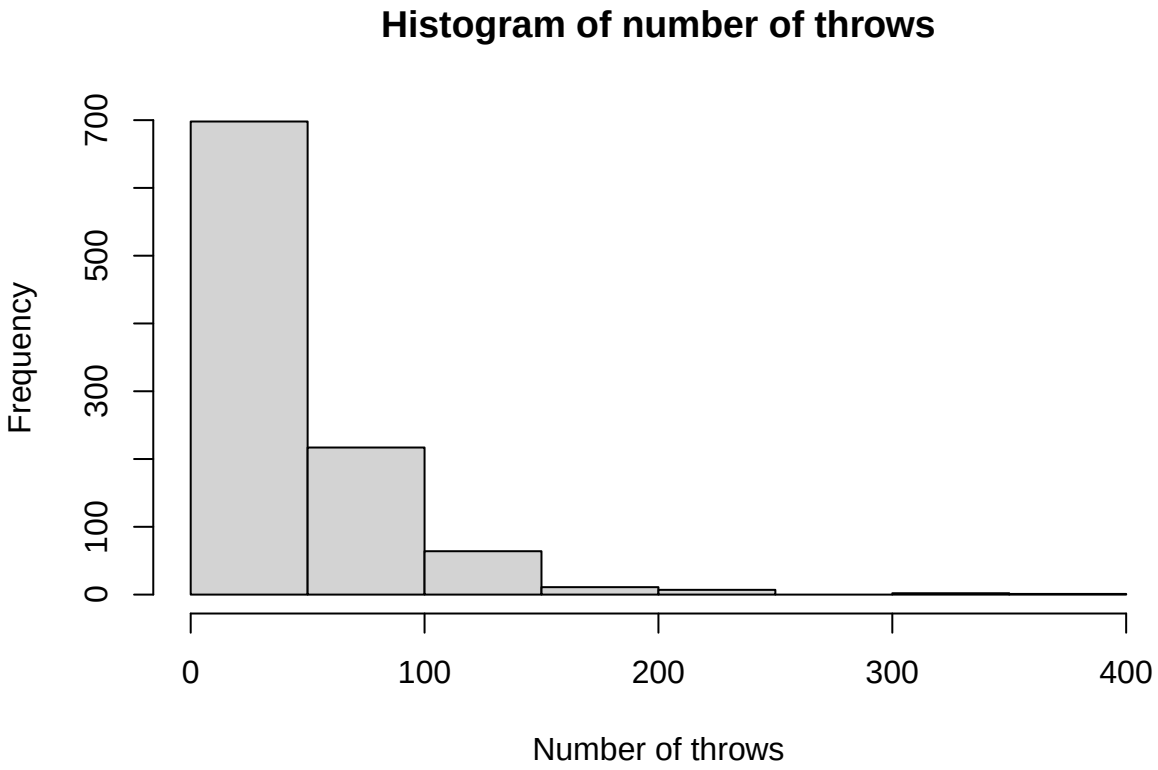


Figure 2: Histogram of empirical distribution of number of throws for starting method 2.

(e) By comparing sample mean starting numbers of throws, which starting criterion should get a player into the game quickest?

Solution

```
mean(samp1) - mean(samp2)
```

```
## [1] -24.2
```

So the second approach, by comparing means, takes more throws before the game can begin.

---

11. Consider the following two functions for calculating

$$d_i = x_{i+1} - x_i, \qquad i = 1, \ldots, n-1$$

where $\mathbf{x}' = (x_1, \ldots, x_n)$.

```
diff1 <- function(x) {
# function to calculate differences of a vector
# based on a for loop
# x is a vector
# returns a vector of length (length(x) - 1)
out <- numeric(length(x) - 1)
for (i in 1:(length(x) - 1)) {
  out[i] <- x[i + 1] - x[i]
}
out
}
```

```
diff2 <- function(x) {
# function to calculate differences of a vector
# based on vectorisation
# x is a vector
# returns a vector of length (length(x) - 1)
id <- 1:(length(x) - 1)
x[id + 1] - x[id]
}
```

The first, `diff1()` uses a straightforward `for()` loop to calculate d_i, for $i = 1, \ldots, n-1$, whereas `diff2()` could be seen to be a vectorised alternative. Benchmark the two for a vector of $n = 1000$ iid $N(0, 1)$ random variates by comparing the median difference in execution time.

Solution

```
x <- rnorm(1000)
microbenchmark::microbenchmark(
  diff1(x),
  diff2(x)
  )
```

```
## Unit: microseconds
```

---

12. The following function assesses whether all elements is a logical vector are `TRUE`.

```
all2 <- function(x) {
# function to calculate whether all elements are TRUE
# returns a scalar
# x is a logical vector
```

```
sum(x) == length(x)
}
```

(a) Calculate the following and benchmark `all2()` against `R`'s built-in function `all()`, which does the same.

```
n <- 1e4
x1 <- !logical(n)
```

Solution

```
all2(x1)
```

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

```
all(x1)
```

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

```
microbenchmark::microbenchmark(
  all2(x1),
  all(x1)
)
```

```
## Unit: microseconds
```

We see that both take a similar amount of time.

(b) Now swap the first element of `x1` so that it's `FALSE` and repeat the benchmarking.

Solution

```
x1[1] <- FALSE
all2(x1)
```

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

```
all(x1)
```

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

```
microbenchmark::microbenchmark(
  all2(x1),
  all(x1)
)
```

```
## Unit: nanoseconds
```

Now `all2()` is much slower. This is because it's performed a calculation on the entire `x1` vector, whereas `all()` has stopped as soon as it's found a `FALSE`.

(c) Evaluate the function `any2()` below against `R`'s built-in function `any()` similarly.

```
any2 <- function(x) {
# function to calculate whether any elements are TRUE
# returns a scalar
# x is a logical vector
sum(x) > 0
}
```

Solution

```
x2 <- logical(n)
any2(x2)
```

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

```
any(x2)
```

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

```
microbenchmark::microbenchmark(
  any2(x2),
  any(x2)
)
```

```
## Unit: microseconds
```

We again see that both take a similar amount of time.

(d) Now swap the first element of `x2` so that it's `FALSE` and repeat the benchmarking.

Solution

```
x2[1] <- TRUE
microbenchmark::microbenchmark(
  any2(x2),
  any(x2)
)
```

```
## Unit: nanoseconds
```

This time `any2()` is much slower, for similar reasoning to `all2()` being much slower than `all()`, except that `any()` is stopped when it reaches a `TRUE`.

---

3 Chapter 3 exercises

1. Calculate $\mathbf{A}^{\mathrm{T}}\mathbf{B}$ in `R` where $\mathbf{A}$ is a $n \times p$ matrix comprising N(0, 1) random variates and $\mathbf{B}$ is a $n \times n$ matrix comprising Uniform([0, 1]) random variates for $n = 1000$ and $p = 500$, using `t(A) %*% B` and `crossprod(A, B)`.

   (a) Confirm that both produce the same result.

Solution

```
A <- matrix(rnorm(n * p), n)
B <- matrix(runif(n * n), n)
all.equal(t(A) %*% B, crossprod(A, B))
```

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

(b) Then benchmark the time that two commands take to complete.

Solution

```
microbenchmark::microbenchmark(
  t(A) %*% B,
  crossprod(A, B)
)
```

```
## Unit: milliseconds
```

---

2. Consider the calculation $\mathbf{AD}$ where

$$\mathbf{A} = \begin{pmatrix} -0.72 & -2.21 & 0.56 & -1.26 \\ -1.17 & -1.20 & -0.31 & 2.80 \\ -0.45 & 0.13 & -0.16 & 2.14 \\ -0.40 & -1.84 & 0.71 & 0.79 \\ -1.18 & -0.51 & -1.50 & 0.06 \\ 1.85 & 0.47 & 0.54 & -1.93 \end{pmatrix} \text{ and } \mathbf{D} = \begin{pmatrix} 0.08 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.75 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.10 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.52 \end{pmatrix}.$$

Write a function in `R` that takes a matrix `A` and a vector `d` as its arguments and computes $\mathbf{AD}$, where $\mathbf{A}$ = `A` and diag($\mathbf{D}$) = `d`, and where diag($\mathbf{D}$) denotes the vector comprising the diagonal elements of $\mathbf{D}$. Consider whether your function is performing redundant calculations and, if it is, try and avoid them.

Solution

We might start with

```
AD <- function(A, d) {
# function to compute A * D
# A is a matrix
# D is a matrix with diagonal elements d
# returns a matrix
D <- diag(d)
A %*% D
}
```

but if we do this then we're multiplying a lot of zeros unnecessarily. Instead, the following avoids this

```
AdiagD <- function(A, d) {
# function to compute A * D slightly more efficiently
# A is a matrix
```

```
# D is a matrix with diagonal elements d
# returns a matrix
t(t(A) * d)
}
```

and the following confirms that both give the same result

```
AD(A, diag(D))
```

```
##         [,1]    [,2]   [,3]    [,4]
## [1,] -0.0576 -1.6575  0.056 -0.6552
## [2,] -0.0936 -0.9000 -0.031  1.4560
## [3,] -0.0360  0.0975 -0.016  1.1128
## [4,] -0.0320 -1.3800  0.071  0.4108
## [5,] -0.0944 -0.3825 -0.150  0.0312
## [6,]  0.1480  0.3525  0.054 -1.0036
```

```
AdiagD(A, diag(D))
```

```
##         [,1]    [,2]   [,3]    [,4]
## [1,] -0.0576 -1.6575  0.056 -0.6552
## [2,] -0.0936 -0.9000 -0.031  1.4560
## [3,] -0.0360  0.0975 -0.016  1.1128
## [4,] -0.0320 -1.3800  0.071  0.4108
## [5,] -0.0944 -0.3825 -0.150  0.0312
## [6,]  0.1480  0.3525  0.054 -1.0036
```

---

3. The following function generates an arbitrary $n \times n$ positive definite matrix.

```
pdmatrix <- function(n) {
# function to generate an arbitrary n x n positive definite matrix
# n is an integer
# returns a matrix
L <- matrix(0, n, n)
L[!lower.tri(L)] <- abs(rnorm(n * (n + 1) / 2))
tcrossprod(L)
}
```

By generating random n-vectors of independent N(0, 1) random variates, $\mathbf{x}_1, \ldots, \mathbf{x}_m,$, say, and random $n \times n$ positive definite matrices, $\mathbf{A}_1, \ldots, \mathbf{A}_m$, say, confirm that $\mathbf{x}_i^{\mathrm{T}} \mathbf{A}_i \mathbf{x}_i > 0$ for $i = 1, \ldots, m$ with $m = 100$ and $n = 10$.

[Note that this can be considered a simulation-based example of trying to prove a result by considering a large number of simulations. Such an approach can be very valuable when an analytical approach is not possible.]

Solution

There are a variety of ways we can tackle this. One of the tidier seems to be to use `all()` and `replicate()`.

```
check_pd <- function(A, x) {
# function to check whether a matrix is positive definite
```

```
# A is a matrix
# returns a logical
sum(x * (A %*% x)) > 0
}
m <- 1e2
n <- 10
all(replicate(1e2,
  check_pd(pdmatrix(n), rnorm(n))
))
```

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

---

4. For the cement factory data of Example 3.25 compute $\hat{\boldsymbol{\beta}}$ by inverting $\mathbf{X}^{\mathrm{T}}\mathbf{X}$ and multiplying by $\mathbf{X}^{\mathrm{T}}\mathbf{y}$, i.e. $\hat{\boldsymbol{\beta}} = (\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{y}$, and by solving $\mathbf{X}^{\mathrm{T}}\mathbf{X}\hat{\boldsymbol{\beta}} = \mathbf{X}^{\mathrm{T}}\mathbf{y}$.

Solution

```
X <- cbind(1, prod$days, prod$temp)
XtX <- crossprod(X)
y <- prod$output
Xty <- crossprod(X, y)
(betahat1 <- solve(XtX) %*% Xty)
```

```
##              [,1]
## [1,]  9.12688541
## [2,]  0.20281539
## [3,] -0.07239294
```

```
(betahat2 <- solve(XtX, Xty))
```

```
##              [,1]
## [1,]  9.12688541
## [2,]  0.20281539
## [3,] -0.07239294
```

---

5. Show in R that $\mathbf{L}$ is a Cholesky decomposition of $\mathbf{A}$ for

$$\mathbf{A} = \begin{pmatrix} 547.56 & 348.66 & 306.54 \\ 348.66 & 278.26 & 199.69 \\ 306.54 & 199.69 & 660.38 \end{pmatrix} \text{ and } \mathbf{L} = \begin{pmatrix} 23.4 & 0.0 & 0.0 \\ 14.9 & 7.5 & 0.0 \\ 13.1 & 0.6 & 22.1 \end{pmatrix}.$$

Solution

We'll load $\mathbf{A}$ and $\mathbf{L}$ as `A` and `L`, respectively.

```
A <- matrix(c(547.56, 348.66, 306.54, 348.66, 278.26, 199.69, 306.54,
              199.69, 660.38), 3, 3)
L <- matrix(c(23.4, 14.9, 13.1, 0, 7.5, 0.6, 0, 0, 22.1), 3, 3)
```

Then we need to show that $\mathbf{L}$ is lower-triangular

```
all(L[upper.tri(L)] == 0)
```

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

which it is, that all its diagonal elements are positive

```
all(diag(L) > 0)
```

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

which they are, and that $\mathbf{A} = \mathbf{L}\mathbf{L}^{\mathrm{T}}$

```
all.equal(A, tcrossprod(L))
```

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

which it does.

---

6. For the matrix $\mathbf{A}$ below, find its Cholesky decomposition, $\mathbf{L}$, where $\mathbf{A} = \mathbf{L}\mathbf{L}^{\mathrm{T}}$ and $\mathbf{L}$ is a lower triangular matrix, and confirm that $\mathbf{L}$ is a Cholesky decomposition of $\mathbf{A}$:

$$\mathbf{A} = \begin{pmatrix} 0.797 & 0.839 & 0.547 \\ 0.839 & 3.004 & 0.855 \\ 0.547 & 0.855 & 3.934 \end{pmatrix}.$$

Solution

We'll load $\mathbf{A}$

```
A <- cbind(
  c(0.797, 0.839, 0.547),
  c(0.839, 3.004, 0.855),
  c(0.547, 0.855, 3.934)
  )
```

and then we'll find $\mathbf{L}$

```
L <- t(chol(A))
L
```

```
##           [,1]      [,2]     [,3]
## [1,] 0.8927486 0.0000000 0.000000
## [2,] 0.9397943 1.4562921 0.000000
## [3,] 0.6127145 0.1917022 1.876654
```

Then we'll check that it's lower-triangular

```
all(L[upper.tri(L)] == 0)
```

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

which it is, then we'll check that its diagonal elements are positive

```
all(diag(L) > 0)
```

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

which they are, and finally we'll check that $\mathbf{L}\mathbf{L}^{\mathrm{T}} = \mathbf{A}$

```
all.equal(A, tcrossprod(L))
```

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

which it does. So we conclude that $\mathbf{L}$ is a lower-triangular Cholesky decomposition of $\mathbf{A}$.

---

7. Form the matrices

$$\mathbf{A}_{11} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \ \mathbf{A}_{12} = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}, \ \mathbf{A}_{21} = \begin{pmatrix} 9 & 10 \\ 11 & 12 \end{pmatrix}, \ \mathbf{A}_{22} = \begin{pmatrix} 13 & 14 \\ 15 & 16 \end{pmatrix}$$

in R and then use these to form

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{pmatrix}.$$

Solution

```
(A11 <- matrix(1:4, 2, 2, byrow = TRUE))
```

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

```
(A12 <- matrix(5:8, 2, 2, byrow = TRUE))
```

```
##      [,1] [,2]
## [1,]    5    6
## [2,]    7    8
```

```
(A21 <- matrix(9:12, 2, 2, byrow = TRUE))
```

```
##      [,1] [,2]
## [1,]    9   10
## [2,]   11   12
```

```
(A22 <- matrix(13:16, 2, 2, byrow = TRUE))
```

```
##      [,1] [,2]
## [1,]   13   14
## [2,]   15   16
```

```
(A <- rbind(cbind(A11, A12), cbind(A21, A22)))
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    5    6
## [2,]    3    4    7    8
## [3,]    9   10   13   14
## [4,]   11   12   15   16
```

---

8. Using $\mathbf{A}_{11}$ and $\mathbf{A}_{12}$ from Question 7, compute $\mathbf{A}_{11} \otimes \mathbf{A}_{12}$ in R.

Solution

```
A11 <- matrix(1:4, 2, 2)
A12 <- matrix(5:8, 2, 2)
A11 %x% A12
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    5    7   15   21
## [2,]    6    8   18   24
## [3,]   10   14   20   28
## [4,]   12   16   24   32
```

---

9. Repeat the formation of $\mathbf{A}$ in Question 7 by considering a Kronecker sum.

Solution

One option is

```
A11 <- matrix(1:4, 2, 2)
kronecker(matrix(c(0, 4, 8, 12), 2, 2, byrow = TRUE), A11, FUN = '+')
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    5    7
## [2,]    2    4    6    8
## [3,]    9   11   13   15
## [4,]   10   12   14   16
```

but there are others.

---

10. Write a function in `R` called `solve_chol()` to solve a system of linear equations $\mathbf{Ax} = \mathbf{b}$ based on the Cholesky decomposition $\mathbf{A} = \mathbf{LL}^{\mathrm{T}}$.

Solution

The following is one option for `solve_chol()`.

```
solve_chol <- function(L, b) {
# Function to solve LL^Tx = b for x
# L is a lower-triangular matrix
# b is a vector of length nrow(L)
# return vector of same length as b
y <- forwardsolve(L, b)
backsolve(t(L), y)
}
```

We'll quickly check that we get the same result as `solve()` using the data from Example 3.2.

```
y <- c(.7, 1.3, 2.6)
mu <- 1:3
Sigma <- matrix(c(4, 2, 1, 2, 3, 2, 1, 2, 2), 3, 3)
res1 <- solve(Sigma, y - mu)
```

```
L <- t(chol(Sigma))
all.equal(res1, solve_chol(L, y - mu))
```

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

Note above that we could use `backsolve(L, y, upper.tri = FALSE, transpose = TRUE)` instead of `backsolve(t(L), y)`, which avoids transposing `L`. Both give the same result, though.

---

11. Show that solving $\mathbf{Ax} = \mathbf{b}$ for $\mathbf{x}$ is equivalent to solving $\mathbf{Ly} = \mathbf{b}$ for $\mathbf{y}$ and then $\mathbf{L}^{\mathrm{T}}\mathbf{x} = \mathbf{y}$ for $\mathbf{x}$ if $\mathbf{A}$ has Cholesky decomposition $\mathbf{A} = \mathbf{LL}^{\mathrm{T}}$. Confirm this based on the cement factory data of Example 3.25 by taking $\mathbf{A} = \mathbf{X}^{\mathrm{T}}\mathbf{X}$, where $\mathbf{X}$ is the linear model's design matrix.

Solution

Let $\mathbf{L}^{\mathrm{T}}\mathbf{x} = \mathbf{y}$. To solve $\mathbf{LL}^{\mathrm{T}}\mathbf{x} = \mathbf{b}$ for $\mathbf{x}$, we want to first solve $\mathbf{Ly} = \mathbf{b}$ for $\mathbf{y}$, and then $\mathbf{L}^{\mathrm{T}}\mathbf{x} = \mathbf{y}$ for $\mathbf{x}$. We can confirm this numerically in `R`.

We already have `X` from Question 4, so we'll re-use that `X`. We'll set $\mathbf{A} = \mathbf{X}^{\mathrm{T}}\mathbf{X}$, and call this `A`. We'll then use `chol()` to calculate its Cholesky decomposition in upper-triangular form, `U`, and lower-triangular form, `L`.

```
A <- crossprod(X)
U <- chol(A)
L <- t(U)
```

The following two commands then solve $\mathbf{Ly} = \mathbf{b}$ for $\mathbf{y}$, and then $\mathbf{L}^{\mathrm{T}}\mathbf{x} = \mathbf{y}$ for $\mathbf{x}$

```
cbind(
  solve(t(L), solve(L, Xty)),
  backsolve(t(L), forwardsolve(L, Xty))
)
```

```
##              [,1]        [,2]
## [1,]  9.12688541  9.12688541
## [2,]  0.20281539  0.20281539
## [3,] -0.07239294 -0.07239294
```

although double use of `solve()` is inefficient compared to using `forwardsolve()` and then `backsolve()`.

Alternatively, if we have $\mathbf{A} = \mathbf{U}^{\mathrm{T}}\mathbf{U}$, for upper-triangular $\mathbf{U}$, then we have the following two options

```
cbind(
  backsolve(U, forwardsolve(t(U), Xty)),
  backsolve(U, forwardsolve(U, Xty, upper.tri = TRUE, transpose = TRUE))
)
```

```
##              [,1]        [,2]
## [1,]  9.12688541  9.12688541
## [2,]  0.20281539  0.20281539
## [3,] -0.07239294 -0.07239294
```

the latter of which is ever so slightly more efficient for its avoidance of `t(U)`.

---

12. Show that $\mathbf{U}$ and $\mathbf{\Lambda}$ form an eigen-decomposition of $\mathbf{A}$ for

$$\mathbf{A} = \begin{pmatrix} 3.40 & 0.00 & 0.00 \\ 0.00 & 0.15 & -2.06 \\ 0.00 & -2.06 & -1.05 \end{pmatrix}, \quad \mathbf{U} = \begin{pmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & 0.6 & -0.8 \\ 0.0 & 0.8 & 0.6 \end{pmatrix} \text{ and } \mathbf{\Lambda} = \begin{pmatrix} 3.4 & 0.0 & 0.0 \\ 0.0 & -2.6 & 0.0 \\ 0.0 & 0.0 & 1.7 \end{pmatrix}.$$

Solution

We'll load $\mathbf{A}$, $\mathbf{\Lambda}$ and $\mathbf{U}$ as `A`, `Lambda` and `U`, respectively.

```
A <- cbind(
  c(3.4, 0, 0),
  c(0, .152, -2.064),
  c(0, -2.064, -1.052)
)
Lambda <- diag(c(3.4, -2.6,  1.7))
U <- matrix(c(1, 0, 0, 0, .6, .8, 0, -.8, .6), 3, 3)
```

Then we need to show that $\mathbf{U}$ is orthogonal,

```
all.equal(crossprod(U), diag(nrow(U)))
```

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

which it is, that $\mathbf{\Lambda}$ is diagonal,

```
all(Lambda - diag(diag(Lambda)) == 0)
```

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

and that $\mathbf{A} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^{\mathrm{T}}$

```
all.equal(A, U %*% tcrossprod(Lambda, U))
```

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

which it does.

---

13. For the matrix $\mathbf{A}$ below, find $\mathbf{U}$ and $\mathbf{\Lambda}$ in its eigen-decomposition of the form $\mathbf{A} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^{\mathrm{T}}$m where $\mathbf{U}$ is orthogonal and $\mathbf{\Lambda}$ is diagonal:

$$\mathbf{A} = \begin{pmatrix} 0.797 & 0.839 & 0.547 \\ 0.839 & 3.004 & 0.855 \\ 0.547 & 0.855 & 3.934 \end{pmatrix}.$$

Solution

We'll load $\mathbf{A}$

```
A <- cbind(
  c(0.797, 0.839, 0.547),
  c(0.839, 3.004, 0.855),
  c(0.547, 0.855, 3.934)
)
```

and then find $\mathbf{U}$ and $\mathbf{\Lambda}$

```
eA <- eigen(A, symmetric = TRUE)
U <- eA$vectors
U
```

```
##            [,1]       [,2]        [,3]
## [1,] -0.2317139 -0.1941590  0.95321085
## [2,] -0.5372074 -0.7913728 -0.29178287
## [3,] -0.8109975  0.5796821 -0.07906848
```

```
Lambda <- diag(eA$values)
Lambda
```

```
##          [,1]     [,2]      [,3]
## [1,] 4.656641 0.000000 0.0000000
## [2,] 0.000000 2.583555 0.0000000
## [3,] 0.000000 0.000000 0.4948042
```

Then we need to show that $\mathbf{U}$ is orthogonal,

```
all.equal(crossprod(U), diag(nrow(U)))
```

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

which it is, that $\mathbf{\Lambda}$ is diagonal,

```
all(Lambda - diag(diag(Lambda)) == 0)
```

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

which it is, and that $\mathbf{A} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^{\mathrm{T}}$

```
all.equal(A, U %*% tcrossprod(Lambda, U))
```

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

which it does.

---

14. Show that $\mathbf{U}$, $\mathbf{D}$ and $\mathbf{V}$ form a singular value decomposition of $\mathbf{A}$ for

$$\mathbf{A} = \begin{pmatrix} 0.185 & 8.700 & -0.553 \\ 0.555 & -2.900 & -1.659 \\ 3.615 & 8.700 & -2.697 \\ 1.205 & -26.100 & -0.899 \end{pmatrix}, \quad \mathbf{U} = \begin{pmatrix} -0.2 & -0.2 & 1.0 \\ -0.5 & -0.8 & -0.3 \\ -0.8 & 0.6 & -0.1 \end{pmatrix},$$

$$\mathbf{D} = \begin{pmatrix} 29 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 1 \end{pmatrix} \text{ and } \mathbf{V} = \begin{pmatrix} 0.00 & 0.76 & 0.65 \\ -1.00 & 0.00 & 0.00 \\ 0.00 & -0.65 & 0.76 \end{pmatrix}.$$

Solution

We'll load $\mathbf{A}$, $\mathbf{U}$, $\mathbf{D}$ and $\mathbf{V}$ as `A`, `U`, `D` and `V`, respectively.

```
A <- matrix(c(0.185, 0.555, 3.615, 1.205, 8.7, -2.9, 8.7, -26.1,
              -0.553, -1.659, -2.697, -0.899), 4, 3)
U <- matrix(c(-0.3, 0.1, -0.3, 0.9, 0.1, 0.3, 0.9, 0.3, -0.3, -0.9,
```

```
          0.3, 0.1), 4, 3)
D <- diag(c(29, 5, 1))
V <- matrix(c(0, -1, 0, 0.76, 0, -0.65, 0.65, 0, 0.76), 3, 3)
```

We want to check that $\mathbf{U}$ and $\mathbf{V}$ are orthogonal

```
all.equal(crossprod(U), diag(ncol(U)))
```

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

```
all.equal(crossprod(V), diag(nrow(V)))
```

```
## [1] "Mean relative difference: 9.999e-05"
```

which they both are, that $\mathbf{D}$ is diagonal

```
all(D - diag(diag(D)) == 0)
```

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

```
all.equal(crossprod(V), diag(nrow(V)))
```

```
## [1] "Mean relative difference: 9.999e-05"
```

which it is, and finally that $\mathbf{A} = \mathbf{U}\mathbf{D}\mathbf{V}^{\mathrm{T}}$

```
all.equal(A, U %*% tcrossprod(D, V))
```

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

which is true.

---

15. By considering $\sqrt{\mathbf{A}}$ as $\mathbf{A}^{1/2}$, i.e. as a matrix power, show how an eigen-decomposition can be used to general multivariate Normal random vectors and then write a function to implement this in `R`.

Solution

From Example 3.14, to generate a multivariate Normal random vector, $\mathbf{Y}$, say, from the $MVN_p(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ distribution we need to compute $\mathbf{Y} = \boldsymbol{\mu} + \mathbf{L}\mathbf{Z}$, where $\boldsymbol{\Sigma} = \mathbf{L}\mathbf{L}^{\mathrm{T}}$ and $\mathbf{Z} = (Z_1, \ldots, Z_p)^{\mathrm{T}}$, where Z_i, $i = 1, \ldots, p$, are independent $N(0, 1)$ random variables. Given an eigen-decomposition of $\boldsymbol{\Sigma} = \mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^{\mathrm{T}}$, we can write this as $\boldsymbol{\Sigma} = \mathbf{U}\boldsymbol{\Lambda}^{1/2}\boldsymbol{\Lambda}^{1/2}\mathbf{U}^{\mathrm{T}}$. As $\boldsymbol{\Lambda}$ is diagonal, $\boldsymbol{\Lambda} = \boldsymbol{\Lambda}^{\mathrm{T}}$ and hence $\boldsymbol{\Lambda}^{1/2}\mathbf{U}^{\mathrm{T}} = (\mathbf{U}\boldsymbol{\Lambda}^{1/2})^{\mathrm{T}}$ and so $\boldsymbol{\Sigma} = \mathbf{L}\mathbf{L}^{\mathrm{T}}$ if $\mathbf{L} = \mathbf{U}\boldsymbol{\Lambda}^{1/2}$. Therefore we can generate $MVN_p(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ random variables with $\mathbf{Y} = \boldsymbol{\mu} + \mathbf{U}\boldsymbol{\Lambda}^{1/2}\mathbf{Z}$.

The following `R` function generates `n` multivariate Normal random vectors with mean `mu` and variance-covariance matrix `Sigma`.

```
rmvn_eigen <- function(n, mu, Sigma) {
# Function to generate MVN random vectors with
# n is a integer, giving the number of independent vectors to simulate
# mu is a p-vector of the MVN mean
# Sigma is a p x p matrix of the MVN variance-covariance matrix
# returns a p times n matrix
eS <- eigen(Sigma, symmetric = TRUE)
p <- nrow(Sigma)
```

```
Z <- matrix(rnorm(p * n), p)
mu + eS$vectors %*% diag(sqrt(eS$values)) %*% Z
}
rmvn_eigen(5, mu, Sigma)
```

```
##          [,1]     [,2]     [,3]       [,4]       [,5]
## [1,] 3.812779 1.328476 4.819841 -1.4787812 -0.6061167
## [2,] 5.302633 3.159313 3.723752 -0.1035905 -0.5120484
## [3,] 5.310861 4.700222 3.865721  2.8249234  0.7614778
```

---

16. Show how an eigen-decomposition can be used to solve a system of linear equations $\mathbf{Ax} = \mathbf{b}$ for $\mathbf{x}$ by matrix multiplications and vector divisions, only. Confirm this in `R` by solving $\mathbf{\Sigma z} = \mathbf{y} - \boldsymbol{\mu}$ for $\mathbf{z}$, with $\mathbf{y}$, $\boldsymbol{\mu}$ and $\mathbf{\Sigma}$ as in Example 3.2.

Solution

To solve $\mathbf{Ax} = \mathbf{b}$ for $\mathbf{x}$, if $\mathbf{A} = \mathbf{U\Lambda U}^{\mathrm{T}}$, then we want to solve $\mathbf{U\Lambda U}^{\mathrm{T}}\mathbf{x} = \mathbf{b}$ for $\mathbf{x}$. The following manipulations can be used

$$\mathbf{\Lambda U}^{\mathrm{T}}\mathbf{x} = \mathbf{U}^{-1}\mathbf{b} \tag{1}$$
$$\mathbf{\Lambda U}^{\mathrm{T}}\mathbf{x} = \mathbf{U}^{\mathrm{T}}\mathbf{b} \tag{2}$$
$$\mathbf{U}^{\mathrm{T}}\mathbf{x} = (\mathbf{U}^{\mathrm{T}}\mathbf{b})/\mathrm{diag}(\mathbf{\Lambda}) \tag{3}$$
$$\mathbf{x} = \mathbf{U}^{-\mathrm{T}}(\mathbf{U}^{\mathrm{T}}\mathbf{b})/\mathrm{diag}(\mathbf{\Lambda}) \tag{4}$$
$$\mathbf{x} = \mathbf{U}[(\mathbf{U}^{\mathrm{T}}\mathbf{b})/\mathrm{diag}(\mathbf{\Lambda})] \tag{5}$$

in which (1) results from premultiplying by $\mathbf{U}^{-1}$, (2) from orthogonality of $\mathbf{U}$, i.e. $\mathbf{U}^{\mathrm{T}} = \mathbf{U}^{-1}$, (3) from elementwise division, given diagonal $\mathbf{\Lambda}$, (4) from premultiplying by $\mathbf{U}^{-\mathrm{T}}$ and (5) from orthogonality of $\mathbf{U}$, again.

Next we'll load the data from Example 3.2

```
y <- c(.7, 1.3, 2.6)
mu <- 1:3
Sigma <- matrix(c(4, 2, 1, 2, 3, 2, 1, 2, 2), 3, 3)
res1 <- solve(Sigma, y - mu)
```

Then we'll go through the calculations given above

```
eS <- eigen(Sigma, symmetric = TRUE)
lambda <- eS$values
U <- eS$vectors
res2 <- U %*% (crossprod(U, y - mu) / lambda)
```

which we see gives the same result as `solve()`, once we use `as.vector()` to convert `res2` from a one-column matrix to a vector.

```
all.equal(res1, as.vector(res2))
```

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

---

17. Show in R that if $\mathbf{H}_4 = \mathbf{UDV}^{\mathrm{T}}$ is the SVD of $\mathbf{H}_4$, the 4×4 Hilbert matrix, then solving $\mathbf{H}_4\mathbf{x} = (1, 1, 1, 1)^{\mathrm{T}}$ for $\mathbf{x}$ reduces to solving

$$\mathbf{V}^{\mathrm{T}}\mathbf{x} \simeq \begin{pmatrix} -1.21257 \\ -4.80104 \\ -26.08668 \\ 234.33089 \end{pmatrix}$$

and then use this to solve $\mathbf{H}_4\mathbf{x} = (1, 1, 1, 1)^{\mathrm{T}}$ for $\mathbf{x}$.

Solution

We ultimately need to solve $\mathbf{UDV}^{\mathrm{T}}\mathbf{x} = \mathbf{b}$, where $\mathbf{b} = (1, 1, 1, 1)^{\mathrm{T}}$. Pre-multiplying by $\mathbf{U}^{-1} = \mathbf{U}^{\mathrm{T}}$, we then need to solve $\mathbf{DV}^{\mathrm{T}}\mathbf{x} = \mathbf{U}^{\mathrm{T}}\mathbf{b}$, which is equivalent to solving $\mathbf{V}^{\mathrm{T}}\mathbf{x} = (\mathbf{U}^{\mathrm{T}}\mathbf{b})/\mathrm{diag}(\mathbf{D})$. The following calculates the SVD of $\mathbf{H}_4$ and then computes $(\mathbf{U}^{\mathrm{T}}\mathbf{b})/\mathrm{diag}(\mathbf{D})$.

```
hilbert <- function(n) {
# Function to evaluate n by n Hilbert matrix.
# n is an integer
# Returns n by n matrix.
ind <- 1:n
1 / (outer(ind, ind, FUN = '+') - 1)
}
H4 <- hilbert(4)
b <- rep(1, 4)
svdH <- svd(H4)
V <- svdH$v
U <- svdH$u
b2 <- crossprod(U, b)
z <- b2 / svdH$d
z
```

```
##             [,1]
## [1,]  -1.212566
## [2,]  -4.801038
## [3,] -26.086677
## [4,] 234.330888
```

which is as given in the question, subject to rounding. Finally we want to solve $\mathbf{V}^{\mathrm{T}}\mathbf{x} = (\mathbf{U}^{\mathrm{T}}\mathbf{b})/\mathrm{diag}(\mathbf{D})$ for $\mathbf{x}$, which we can do with either of the following

```
cbind(
  solve(t(V), z),
  V %*% z
)
```

```
##      [,1] [,2]
## [1,]   -4   -4
## [2,]   60   60
## [3,] -180 -180
## [4,]  140  140
```

and we see that the latter gives the same result as `solve()`

```
all.equal(solve(H4, b), as.vector(V %*% z))
```

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

once we ensure that both are vectors. Note that solving systems of linear equations via the SVD is only a sensible option *if* we already have the SVD. Otherwise, solving via other decompositions is more efficient.

---

18. Show that $\mathbf{Q}$ and $\mathbf{R}$ form a QR decomposition of $\mathbf{A}$ for

$$\mathbf{A} = \begin{pmatrix} 0.337 & 0.890 & -1.035 \\ 0.889 & 6.070 & -1.547 \\ -1.028 & -1.545 & 4.723 \end{pmatrix}, \quad \mathbf{Q} = \begin{pmatrix} -0.241 & 0.101 & 0.965 \\ -0.635 & -0.769 & -0.078 \\ 0.734 & -0.631 & 0.249 \end{pmatrix}$$

and

$$\mathbf{R} = \begin{pmatrix} -1.4 & -5.2 & 4.7 \\ 0.0 & -3.6 & -1.9 \\ 0.0 & 0.0 & 0.3 \end{pmatrix}.$$

Solution

We'll load $\mathbf{A}$, $\mathbf{Q}$ and $\mathbf{R}$ as `A`, `Q` and `R`, respectively.

```
A <- matrix(c(0.3374, 0.889, -1.0276, 0.8896, 6.0704, -1.5452,
          -1.0351, -1.5468, 4.7234), 3, 3)
Q <- matrix(c(-0.241, -0.635, 0.734, 0.101, -0.769, -0.631, 0.965,
          -0.078, 0.249), 3, 3)
R <- matrix(c(-1.4, 0, 0, -5.2, -3.6, 0, 4.7, -1.9, 0.3), 3, 3)
```

We want to show that $\mathbf{Q}$ is orthogonal

```
all.equal(crossprod(Q), diag(nrow(Q)))
```

```
## [1] "Mean relative difference: 0.001286839"
```

which it is (after allowing for a bit of error), that $\mathbf{R}$ is upper-triangular

```
all(R[lower.tri(R)] == 0)
```

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

which it is, and that $\mathbf{A} = \mathbf{QR}$

```
all.equal(A, Q %*% R)
```

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

which it does.

---

19. The determinant of the $n \times n$ Hilbert matrix is given by

$$|\mathbf{H}_n| = \frac{c_n^4}{c_{2n}}$$

where

$$c_n = \prod_{i=1}^{n-1} i!$$

is the Cauchy determinant.

(a) Write a function in R, `det_hilbert(n, log = FALSE)` that evaluates $|\mathbf{H}_n|$ and $\log(|\mathbf{H}_n|)$ if `log = FALSE` or `log = TRUE`, respectively. Your function should compute $\log(|\mathbf{H}_n|)$, and then return $|\mathbf{H}_n|$ if `log = FALSE`, as with `dmvn1()` in Example 3.2.

Solution

It will perhaps be tidier to write a function to calculate the Cauchy determinant

```
det_cauchy <- function(n, log = FALSE) {
# function to calculate Cauchy determinant
# n in an integer
# log is a logical; defaults to FALSE
# returns a scalar
out <- sum(lfactorial(seq_len(n - 1)))
if (!log)
  out <- exp(out)
out
}
```

and then to use that to calculate the determinant of $\mathbf{H}_n$.

```
det_hilbert <- function(n, log = FALSE) {
# function to calculate determinant of Hilbert matrix
# n in an integer
# log is a logical; defaults to FALSE
# returns a scalar
out <- 4 * det_cauchy(n, TRUE) - det_cauchy(2 * n, TRUE)
if (!log)
  out <- exp(out)
out
}
```

(b) Calculate $|\mathbf{H}_n|$ and $\log(|\mathbf{H}_n|)$ through the QR decomposition of $\mathbf{H}_n$ for $n = 5$ and confirm that both give the same result as `det_hilbert()` above.

Solution

We'll start with a function `det_QR()`, which calculates the determinant of a matrix via its QR decomposition

```
det_QR <- function(A) {
# function to calculate determinant of a matrix via QR decomposition
# A is a matrix
# returns a scalar
qrA <- qr(A)
R <- qr.R(qrA)
prod(abs(diag(R)))
}
```

and we see that this gives the same answer as `det_hilbert()` for $n = 5$.

```
det_QR(hilbert(5))
```

```
## [1] 3.749295e-12
```

```
det_hilbert(5)
```

```
## [1] 3.749295e-12
```

Then we'll write a function to calculate the logarithm of the determinant of a matrix through its QR decomposition.

```
logdet_QR <- function(A) {
# function to calculate log determinant of a matrix via QR decomposition
# A is a matrix
# returns a scalar
qrA <- qr(A)
R <- qr.R(qrA)
sum(log(abs(diag(R))))
}
```

which we also see gives the same answer as `det_hilbert()` with `log = TRUE`.

```
logdet_QR(hilbert(5))
```

```
## [1] -26.30945
```

```
det_hilbert(5, log = TRUE)
```

```
## [1] -26.30945
```

---

20. Compute $\mathbf{H}_4^{-1}$ via its QR decomposition, and confirm your result with `solve()` and `qr.solve()`.

Solution

If $\mathbf{H}_4 = \mathbf{QR}$ then $\mathbf{H}_4^{-1} = (\mathbf{QR})^{-1} = \mathbf{R}^{-1}\mathbf{Q}^{-1} = \mathbf{R}^{-1}\mathbf{Q}^{\mathrm{T}}$, since $\mathbf{Q}$ is orthogonal. Therefore we want to solve $\mathbf{RX} = \mathbf{Q}^{\mathrm{T}}$ for $\mathbf{X}$. The following calculates the QR decomposition of $\mathbf{H}_4$ with `qr()` and then extracts $\mathbf{Q}$ and $\mathbf{R}$ as `Q` and `R`, respectively.

```
H4 <- hilbert(4)
qrH <- qr(H4)
Q <- qr.Q(qrH)
R <- qr.R(qrH)
```

Then we want to solve $\mathbf{RX} = \mathbf{Q}^{\mathrm{T}}$ for $\mathbf{X}$, which we do with `backsolve()`, since $\mathbf{R}$, stored as `R`, is upper-triangular.

```
X1 <- backsolve(R, t(Q))
```

The following use `solve()` and `qr.solve()`. Note that if we already have the QR decomposition from `qr()`, then `qr.solve()` uses far fewer calculations to obtain the inverse.

```
X2 <- solve(H4)
X3 <- qr.solve(qrH)
```

We see that both give the same answer

```
all.equal(X2, X3)
```

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

and also the same answer as with `backsolve()` above

```
all.equal(X1, X2)
```

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

---

21. Benchmark Cholesky, eigen (with `symmetric = TRUE` and `symmetric = FALSE`), singular value and QR decompositions of $\mathbf{A} = \mathbf{I}_{100} + \mathbf{H}_{100}$, where $\mathbf{H}_{100}$ is the 100×100 Hilbert matrix. (If you're feeling impatient, consider reducing the value of argument `times` for function `microbenchmark::microbenchmark()`.)

Solution

```
hilbert <- function(n) {
# Function to evaluate n by n Hilbert matrix.
# n is an integer
# Returns n by n matrix.
ind <- 1:n
1 / (outer(ind, ind, FUN = '+') - 1)
}
```

```
H100 <- hilbert(1e2) + diag(1, 1e2)
microbenchmark::microbenchmark(
  chol(H100),
  eigen(H100, symmetric = TRUE),
  eigen(H100),
  svd(H100),
  qr(H100),
  times = 1e2
)
```

```
## Unit: microseconds
```

If we consider median computation times, we see that the Cholesky decomposition is quickest, at nearly six times quicker than the QR decomposition, which is next quickest. The QR decomposition is then about just under three times quicker than the symmetric eigen-decomposition, which takes about the same amount of time as the singular value decomposition. The asymmetric eigen-decomposition is slowest, and demonstrates that *if* we know the matrix we want an eigen-decomposition of is symmetric, then we should pass this information to `R`.

Remark. *The following shows us that if we only want the eigenvalues of a symmetric matrix, then we can further save times by specifying* ***`only.values = TRUE`****.*

```
microbenchmark::microbenchmark(
  eigen(H100, symmetric = TRUE),
  eigen(H100, symmetric = TRUE, only.values = TRUE),
```

```
  times = 1e2
)
```

```
## Unit: microseconds
```

---

22. Given that

$$\mathbf{A} = \begin{pmatrix} 1.23 & 0.30 & 2.58 \\ 0.30 & 0.43 & 1.92 \\ 2.58 & 1.92 & 10.33 \end{pmatrix}, \ \mathbf{A}^{-1} = \begin{pmatrix} 6.9012 & 16.9412 & -4.8724 \\ 16.9412 & 55.2602 & -14.5022 \\ -4.8724 & -14.5022 & 4.0092 \end{pmatrix},$$

$$\mathbf{B} = \begin{pmatrix} 1.49 & 0.40 & 2.76 \\ 0.40 & 0.53 & 2.16 \\ 2.76 & 2.16 & 11.20 \end{pmatrix},$$

and that $\mathbf{B} = \mathbf{A} + \mathbf{L}\mathbf{L}^{\mathrm{T}}$, where $\mathbf{L}$ is a lower-triangular matrix, find $\mathbf{B}^{-1}$ using Woodbury's formula, i.e. using

$$(\mathbf{A} + \mathbf{U}\mathbf{V}^{\mathrm{T}})^{-1} = \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{U}(\mathbf{I}_n + \mathbf{V}^{\mathrm{T}}\mathbf{A}^{-1}\mathbf{U})^{-1}\mathbf{V}^{\mathrm{T}}\mathbf{A}^{-1}.$$

Solution

We note that we can write $\mathbf{U} = \mathbf{V} = \mathbf{L}$ and that $\mathbf{L}\mathbf{L}^{\mathrm{T}} = \mathbf{B} - \mathbf{A}$, and so we can obtain $\mathbf{L}$ via the Cholesky decomposition of $\mathbf{B} - \mathbf{A}$. We'll load $\mathbf{A}$, $\mathbf{A}^{-1}$ and $\mathbf{B}$ below as `A`, `iA` and `B`, and then compute the lower-triangular Cholesky decomposition of $\mathbf{B} - \mathbf{A}$ and store this as `L`.

```
A <- matrix(
  c(1.23, 0.3, 2.58, 0.3, 0.43, 1.92, 2.58, 1.92, 10.33),
  3, 3)
iA <- matrix(
  c(6.9012, 16.9412, -4.8724, 16.9412, 55.2602, -14.5022,
    -4.8724, -14.5022, 4.0092),
  3, 3)
B <- matrix(
  c(1.49, 0.4, 2.76, 0.4, 0.53, 2.16, 2.76, 2.16, 11.2),
  3, 3)
L <- t(chol(B - A))
```

Then we'll write a function to implement Woodbury's formula,

```
woodbury <- function(iA, U, V) {
# function to implement Woodbury's formula
# iA, U and V are matrices
# returns a matrix
I_n <- diag(nrow(iA))
iAU <- iA %*% U
iA - iAU %*% solve(I_n + crossprod(V, iAU), crossprod(V, iA))
}
```

and then use this to find $\mathbf{B}^{-1}$, which we'll call `iB`,

```
iB <- woodbury(iA, L, L)
```

and see that, subject to rounding, this is equal to $\mathbf{B}^{-1}$

```
all.equal(iB, solve(B))
```

```
## [1] "Mean relative difference: 9.113679e-06"
```

---

23. Recall the cement factory data of Example 3.25. Now suppose that a further observation has been obtained based on the factory operating at 20 degrees for 14 days. For given σ^2, the sampling distribution of $\hat{\boldsymbol{\beta}}$ is $MVN_3(\hat{\boldsymbol{\beta}}, \sigma^{-2}(\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1})$. Use the Sherman-Morrison formula to give an expression for the estimated standard errors of $\hat{\boldsymbol{\beta}}$ in terms of σ given that

$$(\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1} = \begin{pmatrix} 2.78 \times 10^{0} & -1.12 \times 10^{-2} & -1.06 \times 10^{-1} \\ -1.12 \times 10^{-2} & 1.46 \times 10^{-4} & 1.75 \times 10^{-4} \\ -1.0 \times 10^{-1} & 1.75 \times 10^{-4} & 4.79 \times 10^{-3} \end{pmatrix}.$$

Solution

If we refer to the Sherman-Morrison formula, we take $\mathbf{A}^{-1} = (\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}$ and $\mathbf{u} = \mathbf{v} = (1, 14, 20)^{\mathrm{T}}$. Then we can calculate the updated variance-covariance matrix of the sampling distribution of $\hat{\boldsymbol{\beta}}$ as

$$\sigma^{-2}(\mathbf{A} + \mathbf{u}\mathbf{v}^{\mathrm{T}})^{-1} = \sigma^{-1}\left[\mathbf{A}^{-1} - \frac{\mathbf{A}^{-1}\mathbf{u}\mathbf{v}^{\mathrm{T}}\mathbf{A}^{-1}}{1 + \mathbf{v}^{\mathrm{T}}\mathbf{A}^{-1}\mathbf{u}}\right]$$

which can be calculated in R with

```
V <- matrix(
  c(2.78, -0.0112, -0.106,
    -0.0112, 0.000146, 0.000175,
    -0.106, 0.000175, 0.00479),
  3, 3)
u <- c(1, 14, 20)
V2 <- V - (V %*% tcrossprod(u) %*% V) / (1 + crossprod(u, V %*% u))[1, 1]
V2
```

```
##               [,1]          [,2]          [,3]
## [1,]  2.580467260 -0.0089572393 -0.1029269103
## [2,] -0.008957239  0.0001207912  0.0001404583
## [3,] -0.102926910  0.0001404583  0.0047426700
```

Taking the diagonal elements of V2 we get

$$\left(\text{e.s.e.}(\hat{\beta}_0), \text{e.s.e.}(\hat{\beta}_1), \text{e.s.e.}(\hat{\beta}_2)\right) = \sigma^{-1}\left(1.606\ , 0.011\ , 0.069\right).$$

---

24. Consider a polynomial regression model of the form

$$Y \mid x \sim N(m(x), \sigma^2)$$

for $\sigma > 0$ and with

$$m(x) = \beta_0 + \sum_{i=1}^{p} \beta_i x^i$$

for $(p+1)$-vector of regression coefficients $\boldsymbol{\beta} = (\beta_0, \beta_1, \ldots, \beta_p)^\mathsf{T}$.

Now let Y_i, where $i = 1, 2, \ldots, 145$, denote the global surface temperature anomaly for years $1880, 1881, \ldots, 2024$. Then let $x_i = i - 70$. We can load these data in `R` with

```
data <- read.csv('https://byoungman.github.io/MTH3045/hockey.csv')
```

For $p = 3$, estimate $E(Y_i \mid x_i)$ for $i = 1, \ldots, 145$ using maximum likelihood.

Solution

Note that we can write $E(Y \mid x) = \mathbf{x}^\mathsf{T}\boldsymbol{\beta}$, where $\mathbf{x}^\mathsf{T} = c(1, x, x^2, \ldots, x^p)$. Then our maximum likelihood estimate of $\boldsymbol{\beta}$ is $\hat{\boldsymbol{\beta}} = (\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y}$, where $\mathbf{X}$ has ith row $\mathbf{x}_i^\mathsf{T} = (1, x_i, x_i^2, x_i^3)$. We can form this with

```
x <- 1:length(data$year) - 70
X <- cbind(1, x, x^2, x^3)
```

and then read in the response data with

```
y <- data$temperature
```

Then we can obtain $\hat{\boldsymbol{\beta}}$ with

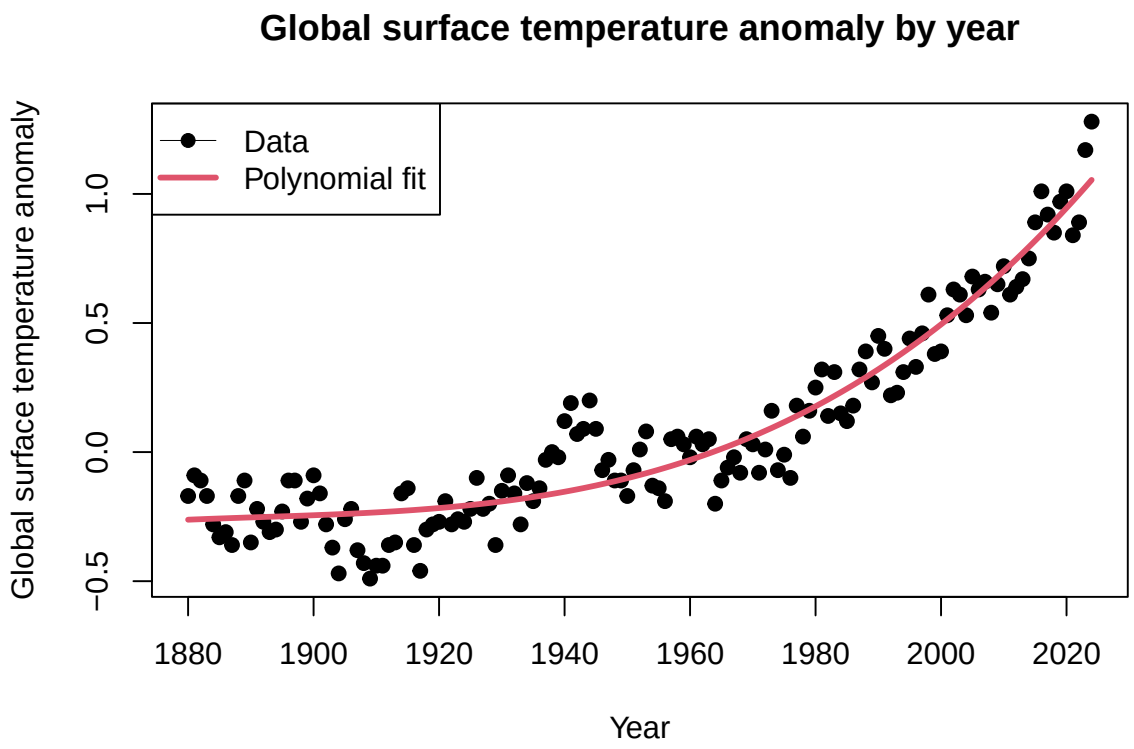
```
beta_hat <- solve(crossprod(X), crossprod(X, y))
```

Finally, our estimates of $E(Y_1 \mid x_1), \ldots, E(Y_{145} \mid x_{145})$ can be calculated as $\hat{\mathbf{y}} = \mathbf{X}\hat{\boldsymbol{\beta}}$, as below

```
y_hat <- X %*% beta_hat
```

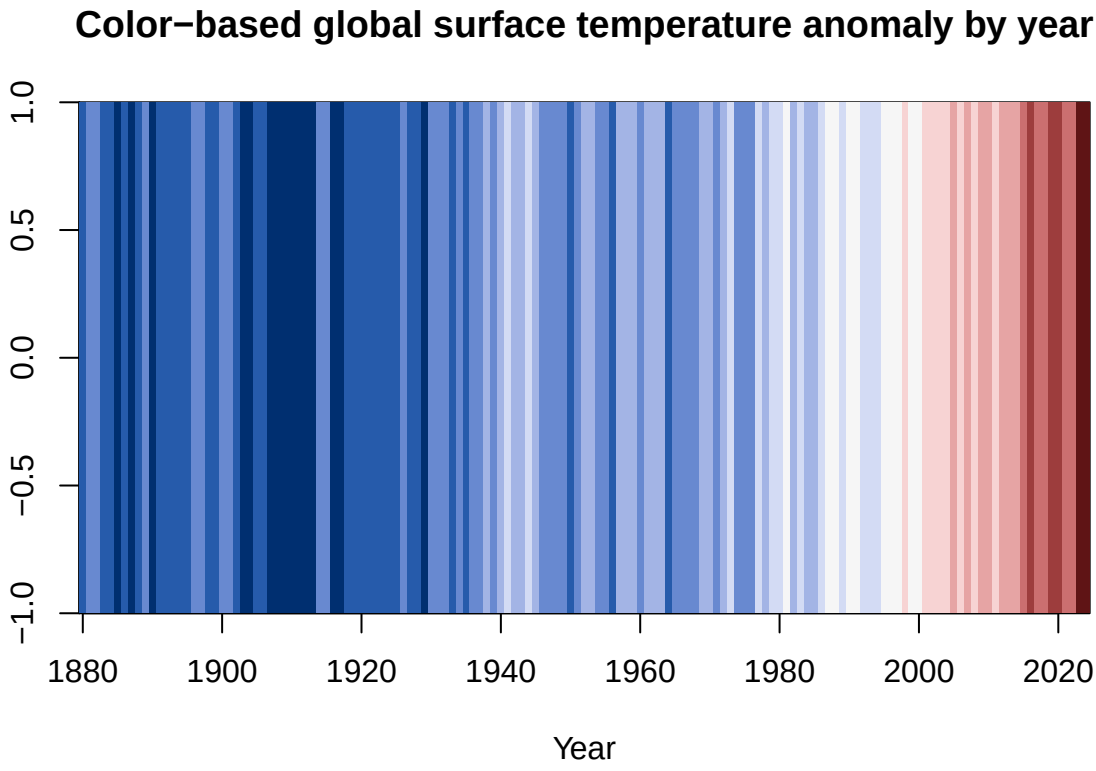
(Aside: these are the data seen in the so-called hockey stick graph.

```
plot(data$year, data$temperature,
     xlab = 'Year', ylab = 'Global surface temperature anomaly',
     main = 'Global surface temperature anomaly by year', pch = 19)
lines(data$year, y_hat, col = 2, lwd = 3)
legend('topleft', legend = c('Data', 'Polynomial fit'),
    pch = c(19, -1), lwd = c(0, 3), col = c(1, 2))
```



Sometimes, the data are plotted as blues and reds

```
image(x = data$year, z = matrix(data$temperature, ncol = 1),
      col = hcl.colors(11, 'Blue-Red 3'), xlab = 'Year', ylab = '',
      main = 'Color-based global surface temperature anomaly by year')
```



as in the plot above.)

---

4 Chapter 4 exercises

1. For $f(x) = \mathrm{e}^{2x}$ approximate $f'(x)$ by finite-differencing with $\delta = 10^{-6}$. Use $x = -1.0, -0.9, \ldots, 0.9, 1.0$ and find the difference between your approximations and the true derivatives.

Solution

The following calculates the finite-difference approximation to the derivative for $x = -1.0, -0.9, \ldots, 0.9, 1.0$ and $\delta = 10^{-6}$.

```
x <- seq(-1, 1, by = .1)
delta <- 1e-6
fd <- (exp(2 * (x + delta)) - exp(2 * x)) / delta
```

Then the following calculates the true derivative for the stated x values as `true` and calculates its difference from the finite-difference approximations.

```
true <- 2 * exp(2 * x)
fd - true
```

```
##  [1] 2.706593e-07 3.306135e-07 4.038067e-07 4.932252e-07 6.023919e-07 7.357217e-07
##  [7] 8.986189e-07 1.097651e-06 1.340685e-06 1.637477e-06 2.000013e-06 2.442704e-06
## [13] 2.983658e-06 3.644212e-06 4.450992e-06 5.436657e-06 6.640438e-06 8.110137e-06
## [19] 9.906360e-06 1.210034e-05 1.477673e-05
```

Looking at the range of the differences

```
range(fd - true)
```

```
## [1] 2.706593e-07 1.477673e-05
```

we see that finite-differencing overestimates the true derivative for all the x values, but at most only by a small amount, i.e. $< 2 \times 10^{-5}$.

---

2. The normal pdf, denoted $\phi(y; \mu, \sigma^2)$, where μ is its mean and σ^2 is its variance, is defined as

$$\phi(y; \mu, \sigma^2) = \frac{\mathrm{d}\Phi(y; \mu, \sigma^2)}{\mathrm{d}y}$$

where $\Phi(y; \mu, \sigma^2)$ denotes its corresponding cdf. Confirm this result by finite-differencing for $y = -2, -1, 0, 1, 2$, $\mu = 0$ and $\sigma^2 = 1$ using `pnorm()` to evaluate $\Phi(y; \mu, \sigma^2)$.

Solution

We'll start by defining the y values

```
y <- seq(-2, 2)
```

Then we'll evaluate $\Phi(y; \mu, \sigma^2)$ and $\Phi(y + \delta; \mu, \sigma^2)$ for these y values, choosing $\delta = 10^{-6}$.

```
Fy <- pnorm(y)
delta <- 1e-6
Fy2 <- pnorm(y + delta)
```

The finite-difference approximation is then given by $[\Phi(y+\delta;\mu,\sigma^2) - \Phi(y;\mu,\sigma^2)]/\delta$, which we can calculate in R with the following.

```
fy_fd <- (Fy2 - Fy) / delta
fy_fd
```

```
## [1] 0.05399102 0.24197085 0.39894228 0.24197060 0.05399091
```

We can then compare this to R's `dnorm()` function

```
fy <- dnorm(y)
all.equal(fy, fy_fd)
```

```
## [1] "Mean relative difference: 3.53179e-07"
```

and see that the two are approximately equal. (We shouldn't expect exact equality because `pnorm()` is only an approximation to $\Phi(y;\mu,\sigma^2)$, since the normal distribution's cdf doesn't have closed form, and because finite-differencing is only an approximation.)

---

3. The Rosenbrock function (sometimes called the banana function) is given by

$$f(\mathbf{x}) = (a - x_1)^2 + b(x_2 - x_1^2)^2,$$

where $\mathbf{x} = (x_1, x_2)^{\mathrm{T}}$.

(a) Write a function, `rosenbrock(x, a, b)`, to evaluate the Rosenbrock function in R for vector $\mathbf{x} =$ `x` and scalars $a =$ `a` and $b =$ `b`, with default values `a = 1` and `b = 100`, and evaluate the function at $\mathbf{x}_0 = (1, 2)^{\mathrm{T}}$ with the default values of a and b.

Solution

```
rosenbrock <- function(x, a = 1, b = 100) {
# function to evaluate Rosenbrock's banana function
# x, a and b are scalars
# returns a scalar
(a - x[1])^2 + b * (x[2] - x[1]^2)^2
}
x0 <- c(1, 2)
rosenbrock(x0)
```

```
## [1] 100
```

It's also useful to plot the function, so that we know what we're dealing with. The following gives a contour plot and coloured surface plot.

```
par(mfrow = 1:2)
x1 <- seq(-2, 2, by = .1)
x2 <- seq(-1, 1.5, by = .1)
x12_grid <- expand.grid(x1 = x1, x2 = x2)
f12 <- matrix(apply(x12_grid, 1, rosenbrock), length(x1))
contour(x1, x2, f12, nlevels = 20)
x1_mat <- matrix(x12_grid$x1, length(x1))
x2_mat <- matrix(x12_grid$x2, length(x1))
plot3D::surf3D(x1_mat, x2_mat, f12, colvar = f12, colkey = TRUE,
  box = TRUE, bty = "b", phi = 20, theta = 15)
```

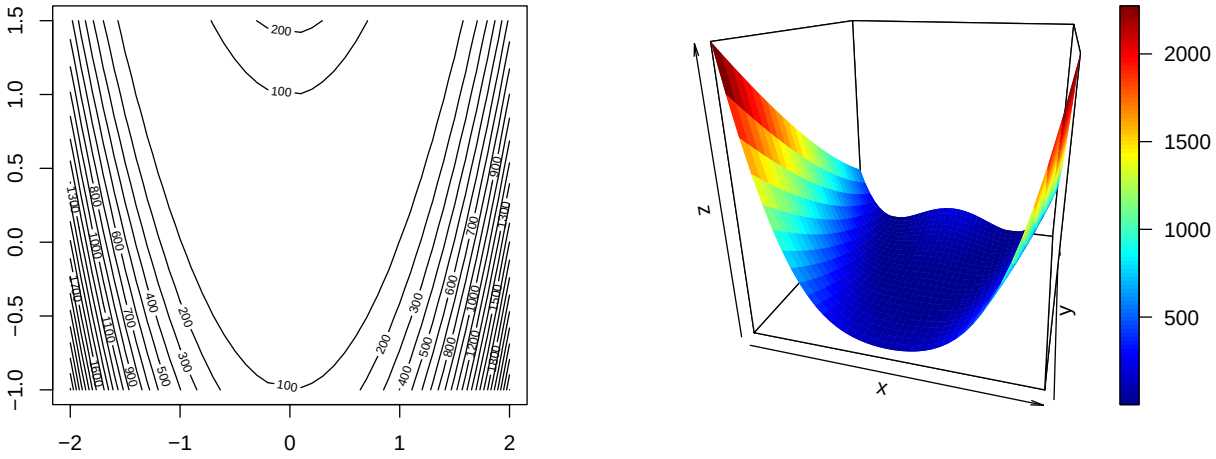


Figure 3: Rosenbrock's function as a contour plot (left) and surface plot (right).

As we see, Rosenbrock's function is a rather complicated function.

(b) Find $\partial f(\mathbf{x})/\partial x_1$ and $\partial f(\mathbf{x})/\partial x_2$ and then write a function in `R`, `rosenbrock_d1(x, a, b)`, that returns the vector $(\partial f(\mathbf{x})/\partial x_1, \partial f(\mathbf{x})/\partial x_2)^{\mathrm{T}}$ and has the same default values as `rosenbrock()`, and then evaluate this gradient operator for $\mathbf{x}_0 = (1, 2)^{\mathrm{T}}$ and the default values of a and b.

Solution

The partial derivatives are

$$\frac{\partial f(\mathbf{x})}{\partial x_1} = -2(a - x_1) - 4bx_1(x_2 - x_1^2) \text{ and } \frac{\partial f(\mathbf{x})}{\partial x_2} = 2b(x_2 - x_1^2),$$

which can be evaluated in `R` with

```
rosenbrock_d1 <- function(x, a = 1, b = 100) {
# function to evaluate first partial derivatives
# of Rosenbrock's banana function
# x, a and b are scalars
# returns a 2-vector
d_x2 <- 2 * b * (x[2] - x[1]^2)
c(-2 * (a - x[1]) - 2 * x[1] * d_x2, d_x2)
}
```

and for $\mathbf{x}_0$ and the default values of a and b we get

```
rosenbrock_d1(x0)
```

```
## [1] -400  200
```

(c) Find $\partial^2 f(\mathbf{x})/\partial x_1^2$, $\partial^2 f(\mathbf{x})/\partial x_1 \partial x_2$ and $\partial^2 f(\mathbf{x})/\partial x_2^2$ and then write a function in `R`, `rosenbrock_d2(x, a, b)`, that returns the 2×2 Hessian matrix of $f(\mathbf{x})$ and has the same default values as `rosenbrock()`, and then evaluate this Hessian matrix for $\mathbf{x}_0 = (1, 2)^{\mathrm{T}}$ and the default values of a and b.

Solution

The second partial derivatives are

$$\frac{\partial f(\mathbf{x})}{\partial^2 x_1^2} = 2 - 4b(x_2 - 3x_1^2), \quad \frac{\partial f(\mathbf{x})}{\partial x_1 \partial x_2} = -4bx_1 \text{ and } \frac{\partial^2 f(\mathbf{x})}{\partial x_2^2} = 2b,$$

which can be evaluated in `R` with

```
rosenbrock_d2 <- function(x, a = 1, b = 100) {
# function to evaluate second partial derivatives
# of Rosenbrock's banana function
# x, a and b are scalars
# returns a 2 x 2 matrix
d_x1x1 <- 2 - 4 * b * (x[2] - 3 * x[1]^2)
d_x1x2 <- -4 * b * x[1]
d_x2x2 <- 2 * b
matrix(c(d_x1x1, d_x1x2, d_x1x2, d_x2x2), 2, 2)
}
```

and for $\mathbf{x}_0$ and the default values of a and b we get

```
rosenbrock_d2(x0)
```

```
##      [,1] [,2]
## [1,]  402 -400
## [2,] -400  200
```

(d) Use `rosenbrock_d1()` to confirm that $\tilde{\mathbf{x}} = (1, 1)^{\mathrm{T}}$ is a minimum of $f(\mathbf{x})$, i.e. that gradient operator is approximately a vector of zeros at $\tilde{\mathbf{x}} = (1, 1)^{\mathrm{T}}$.

Solution

We'll call $\tilde{\mathbf{x}}$, `x_tilde`,

```
x_tilde <- c(1, 1)
```

and then evaluating the first derivatives

```
rosenbrock_d1(x_tilde)
```

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

we see that these are zero, and to confirm that the Hessian matrix is positive definite, we can confirm that all its eigenvalues are positive

```
H <- rosenbrock_d2(x_tilde) # Hessian matrix
lambda <- eigen(H, only.values = TRUE, symmetric = TRUE)$values
all(lambda > 0)
```

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

which they are, and so $\tilde{\mathbf{x}}$ is a minimum (although we don't know whether it's a local or global minimum from these two calculations).

4. Recall the $MVN_p(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ log-likelihood, e.g. from Example 3.2.

   (a) Find $\partial \log f(\mathbf{y} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma})/\partial \boldsymbol{\mu}$ analytically and evaluate this for $\mathbf{y}$, $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ as given in Example 3.2.

Solution

$$\frac{\partial \log f(\mathbf{y} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma})}{\partial \boldsymbol{\mu}} = \boldsymbol{\Sigma}^{-1}(\mathbf{y} - \boldsymbol{\mu}).$$

Evaluating this for $\mathbf{y}$, $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ as given can be done with the following code

```
y <- c(.7, 1.3, 2.6)
mu <- 1:3
Sigma <- matrix(c(4, 2, 1, 2, 3, 2, 1, 2, 2), 3, 3)
(d_mu <- solve(Sigma, y - mu))
```

```
## [1]  0.08 -0.38  0.14
```

so that

$$\frac{\partial \log f(\mathbf{y} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma})}{\partial \boldsymbol{\mu}} \simeq \begin{pmatrix} 0.08 \\ -0.38 \\ 0.14 \end{pmatrix}.$$

   (b) Confirm the above result by finite-differencing.

Solution

We'll use function `fd()` again from Example 4.4 of the notes.

```
fd <- function(x, f, delta = 1e-6, ...) {
# Function to evaluate derivative by finite-differencing
# x is a p-vector
# fn is the function for which the derivative is being calculated
# delta is the finite-differencing step, which defaults to 10^{-6}
# returns a vector of length x
f0 <- f(x, ...)
p <- length(x)
f1 <- numeric(p)
for (i in 1:p) {
  x1 <- x
  x1[i] <- x[i] + delta
  f1[i] <- f(x1, ...)
}
(f1 - f0) / delta
}
```

We'll also need function `dmvn3()` from Example 3.13.

```
dmvn3 <- function(y, mu, Sigma, log = TRUE) {
  # Function to evaluate multivariate Normal pdf by solving
```

```
  # a system of linear equations via Cholesky decomposition
  # y and mu are vectors
  # Sigma is a square matrix
  # log is a logical
  # Returns scalar, on log scale, if log == TRUE.
  p <- length(y)
  res <- y - mu
  L <- t(chol(Sigma))
  out <- - sum(log(diag(L))) - 0.5 * p * log(2 * pi) -
    0.5 * sum(forwardsolve(L, res)^2)
  if (!log)
    out <- exp(out)
  out
}
```

Then the following approximates $\partial \log f(\mathbf{y} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) / \partial \boldsymbol{\mu}$ by finite-differencing

```
d_mu_fd <- fd(mu, dmvn3, y = y, Sigma = Sigma)
```

Note above that we've given `y = y` and `Sigma = Sigma`, then the first free argument to `dmvn3()` is `mu`, and hence `fd()` takes this as `x`. The finite-differencing approximation is

```
d_mu_fd
```

```
## [1]  0.0799998 -0.3800007  0.1399992
```

and is the same as the analytical result

```
all.equal(d_mu, d_mu_fd)
```

```
## [1] "Mean relative difference: 2.833149e-06"
```

once we allow for error in the finite-difference approximation.

---

5. The Gamma(α, β) pdf is given by

$$f(y \mid \alpha, \beta) = \frac{\beta^\alpha}{\Gamma(\alpha)} y^{\alpha-1} \exp(-\beta y) \quad \text{for } y > 0, \tag{6}$$

with $\alpha, \beta > 0$. The gamma distribution's cdf, however, has no closed form. Approximate its cdf $F(y \mid \alpha, \beta)$ for $y = 1.5$, $\alpha = 1$ and $\beta = 2$ using the midpoint rule with $N = 14$ and compare your result to `R`'s `pgamma(y, alpha, beta)` function.

Solution

We can use `dgamma(y, alpha, beta)` to evaluate the Gamma(α, β) pdf (although it's fine to explicitly write a function in `R` to evaluate $f(y \mid \alpha, \beta)$). Then we want to specify the integration nodes for $N = 14$. These will be $0.05, 0.15, \ldots, 1.35, 1.45$, if they're equally spaced, and hence $h = 0.1$.

The following then approximates the integral as `I_midpoint`.

```
nodes <- seq(.05, 1.45, by = .1)
h <- .1
```

```
I_midpoint <- h * sum(dgamma(nodes, 1, 2))
I_midpoint
```

```
## [1] 0.9486311
```

Using R's `pgamma()` function we get

```
I_pgamma <- pgamma(1.5, 1, 2)
I_pgamma
```

```
## [1] 0.9502129
```

Assuming R's approximation to be the more accurate (which we'll assume in MTH3045 is always a good assumption for R's functions for evaluating cdfs), we'll compares the relative absolute error of the midpoint approximation to `pgamma()`, which we'll call `rel_err`

```
rel_err <- abs((I_pgamma - I_midpoint) / I_pgamma)
rel_err
```

```
## [1] 0.001664724
```

and we see is about 0.17%.

---

6. Consider integrating a function $f(x)$ over and arbitrary interval $[a, b]$.

   (a) Write a function to integrate $f(x)$ over $[a, b]$ using the midpoint rule with N midpoints.

Solution

The following function, `integrate_midpoint()`, integrates a function over $[a, b]$ using the midpoint rule with N midpoints.

```
integrate_midpoint <- function(f, a, b, N) {
  # Function to approximate integral of f by midpoint rule over [a, b]
  # f is a function
  # a and b are scalars
  # N is integer number of midpoints
  # returns scalar
  ends <- seq(0, 1, l = N + 1)
  h <- ends[2] - ends[1]
  mids <- ends[-1] - .5 * h
  h * sum(f(mids))
}
```

   (b) Then use this to approximate $\int_0^1 f(x)\mathrm{d}x$, where

$$f(x) = 1 + 2\cos(2x) + 2\sin(2x)$$

   using the midpoint rule with $N = 8$.

Solution

We'll first create function $f(x)$, which we'll call `f`,

```
f <- function(x) 1 + 2 * cos(2 * x) + 2 * sin(2 * x)
```

and then we'll use `integrate_midpoint()` to approximate its integral over $[0, 1]$ with $N = 8$.

```
midpoint <- integrate_midpoint(f, 0, 1, 8)
midpoint
```

```
## [1] 3.331511
```

(c) Compare your integral approximation to the exact result in terms of its relative absolute error.

Solution

As

$$\begin{aligned}\int_0^1 1 + 2\cos(2x) + 2\sin(2x)\mathrm{d}x &= [x + \sin(2x) - \cos(2x)]_0^1 \\ &= [1 + \sin(2) - \cos(2)] - [\sin(0) - \cos(0)] \\ &= 2 + \sin(2) - \cos(2),\end{aligned}$$

then the true integral is $2 + \sin(2) - \cos(2) \simeq 3.3254$, which we'll store as `true`. The relative absolute error, `rel_err`, is then

```
true <- 2 + sin(2) - cos(2)
rel_err <- abs((true - midpoint) / true)
rel_err
```

```
## [1] 0.001824388
```

and on this occasion we see a relative absolute error of approximately 0.18%, for just $N = 8$.

---

7. The composite trapezium rule for integration is given by

$$\int_a^b f(x)\mathrm{d}x \simeq \frac{h}{2}\left[f(x_1^*) + f(x_N^*) + 2\sum_{i=2}^{N-1} f(x_i^*)\right],$$

for N integration nodes $x_i^* = a + (i-1)h$ where $h = (b-a)/(N-1)$.

(a) Write a function, `integrate_trapezium(f, a, b, N)`, to implement the trapezium rule for an integral $\int_a^b f(x)\mathrm{d}x$ with N integration nodes.

Solution

```
integrate_trapezium <- function(f, a, b, N) {
  # Function to approximate integral of f by trapezium rule over [a, b]
  # f is a function
  # a and b are scalars
  # N is integer number of integration nodes
```

```
  # returns scalar
  nodes <- seq(a, b, l = N)
  h <- (b - a) / (N - 1)
  int <- 2 * sum(f(nodes[2:(N - 1)]))
  int <- int + sum(f(nodes[c(1, N)]))
  .5 * h * int
}
```

(b) Then use `integrate_trapezium()` to approximate $F(y \mid \alpha, \beta) = \int_0^y f(z \mid \alpha, \beta)\mathrm{d}z$ using $N = 14$, where $f(y \mid \alpha, \beta)$ is the gamma pdf as given in Equation (6).

Solution

We'll start with a function to evaluate the Gamma(1, 2) pdf, `dgamma2(y)`.

```
dgamma2 <- function(y) dgamma(y, 1, 2)
```

Then the trapezium rule approximation to $F(1.5 \mid 1, 2)$ is given below as `I_trapezium`.

```
I_trapezium <- integrate_trapezium(dgamma2, 0, 1.5, 14)
I_trapezium
```

```
## [1] 0.9544261
```

(c) Compare your integral approximation by the trapezium rule to that of the midpoint rule by using `pgamma()` as the benchmark of 'the truth'.

Solution

```
rel_err <- c(midpoint = abs((I_pgamma - I_midpoint) / I_pgamma),
             trapezium = abs((I_pgamma - I_trapezium) / I_pgamma))
rel_err
```

```
##    midpoint   trapezium
## 0.001664724 0.004433936
```

We see that the relative absolute error of the trapezium rule is 0.44% and so is greater than that of the midpoint rule, given the same number of integration nodes.

---

8. Repeat Question 5 using Simpson's rule and $N = 14$, comparing your approximation to the actual integral *and* your midpoint rule approximation.

Solution

For Simpson's composite rule, the N integration nodes are given by $x_{1i} = a + h(2i - 1)/2$ and $x_{2i} = a + ih$ where $h = (b - a)/N$.

```
a <- 0
b <- 1.5
```

```r
N <- 140
h <- (b - a) / N
nodes1 <- a + h * (2*c(1:N) - 1) / 2
nodes2 <- a + h * c(1:(N - 1))
I_simpson <- dgamma2(a) + dgamma2(b)
I_simpson <- I_simpson+ 4 * sum(dgamma2(nodes1)) + 2 * sum(dgamma2(nodes2))
I_simpson <- h * I_simpson / 6
I_simpson
```

```
## [1] 0.9502129
```

```r
rel_err <- abs((c(I_pgamma, I_midpoint) - I_simpson) / c(I_pgamma, I_midpoint))
rel_err
```

```
## [1] 7.321086e-11 1.667500e-03
```

---

9. Repeat Question 6 using Simpson's rule and $N = 8$, comparing your approximation to the actual integral *and* your midpoint rule approximation.

Solution

```r
integrate_simpson <- function(f, a, b, N) {
  # Function to approximate integral of f by Simpson's rule over [a, b]
  # f is a function
  # a and b are scalars
  # N is integer number of integration nodes
  # returns scalar
  h <- (b - a) / N
  nodes1 <- a + h * (2 * c(1:N) - 1) / 2
  nodes2 <- a + h * c(1:(N - 1))
  h * (f(a) + 4 * sum(f(nodes1)) + 2 * sum(f(nodes2)) + f(b)) / 6
}
simpson <- integrate_simpson(f, 0, 1, 8)
simpson
```

```
## [1] 3.325447
```

```r
rel_err <- abs((c(true, midpoint) - simpson) / c(true, midpoint))
rel_err
```

```
## [1] 9.502376e-07 1.820118e-03
```

---

10. Simpson's composite 3/8 rule approximates an integral as

$$\int_a^b f(x)\mathrm{d}x \simeq \frac{3h}{8}\left(f(a) + 3\sum_{i=1}^{N/3} f(x^*_{3i-2}) + 3\sum_{i=1}^{N/3} f(x^*_{3i-1}) + 2\sum_{i=1}^{N/3-1} f(x^*_{3i}) + f(b)\right),$$

for integration nodes $x^*_i = a + ih$, $i = 1, \ldots, N-1$, with $h = (b-a)/N$ and where N must be a multiple of three. Approximate $I = \int_0^1 \mathrm{e}^x \mathrm{d}x$ using Simpson's 3/8 rule with $N = 9$ nodes and find the relative absolute error of your approximation.

Solution

```
f <- function(x) exp(x)
true <- exp(1) - 1
N <- 9
a <- 0
b <- 1
h <- (b - a) / N
nodes <- a + h * (1:(N - 1))
id2 <- 3 * 1:((N / 3) - 1) # multiply by two
id3 <- c(3 * (1:(N / 3)) - 1, 3 * (1:(N / 3)) - 2) # multiply by three
simpson38 <- f(a) + 3 * sum(f(nodes[id3])) + 2 * sum(f(nodes[id2])) + f(b)
(simpson38 <- 3 * h * simpson38 / 8)
```

```
## [1] 1.718285
```

```
rel_err <- abs((true - simpson38) / true)
rel_err
```

```
## [1] 1.899613e-06
```

---

11. Repeat Question 5 using Gaussian quadrature with $N = 7$ integration nodes, comparing your approximation to the actual integral *and* your midpoint *and* Simpson's rule approximations of questions 5 and 9, respectively.

Solution

```
N <- 7
gq_nodes <- pracma::gaussLegendre(N, 0, 1.5)
I_gq <- sum(gq_nodes$w * dgamma(gq_nodes$x, 1, 2))
I_gq
```

```
## [1] 0.9502129
```

```
I_vec <- c(I_pgamma, I_midpoint, I_simpson)
rel_err <- abs((I_vec - I_gq) / I_vec)
rel_err
```

```
## [1] 2.281873e-13 1.667500e-03 7.343905e-11
```

---

12. Approximate

$$I = \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1}(x_1x_2x_3)^2\mathrm{d}x_1\mathrm{d}x_2\mathrm{d}x_3$$

using the midpoint rule with $N = 20$ and evaluate the approximation's relative absolute error.

Solution

Let's start by finding the true integral

$$I = \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1}(x_1x_2x_3)^2 \mathrm{d}x_1\mathrm{d}x_2\mathrm{d}x_3 = \int_{-1}^{1}\int_{-1}^{1}\left[\frac{x_1^3}{3}\right]_{-1}^{1}\mathrm{d}x_2\mathrm{d}x_3$$

$$= \int_{-1}^{1}\int_{-1}^{1}\left[\frac{1}{3} - \frac{-1}{3}\right]_{-1}^{1}\mathrm{d}x_2\mathrm{d}x_3 = \frac{2}{3}\int_{-1}^{1}\int_{-1}^{1}(x_2x_3)^2\mathrm{d}x_2\mathrm{d}x_3 = \ldots = \left(\frac{2}{3}\right)^3.$$

```
true <- (2 / 3)^3
```

For the midpoint rule, we'll first find the midpoints, which can be the same for each dimension in this case.

```
N <- 20
edges <- seq(-1, 1, l = N + 1)
h <- edges[2] - edges[1]
mids <- edges[-1] - .5 * h
```

Then we'll find all combinations of these, `x123`, and evaluate the integrand at them, `f123`.

```
x123 <- expand.grid(x1 = mids, x2 = mids, x3 = mids)
f123 <- apply(x123, 1, prod)^2
```

We'll call our midpoint approximation `midpoint`, which can be calculated with

```
midpoint <- sum(h^3 * f123)
midpoint
```

```
## [1] 0.2940796
```

Then its relative absolute error, `rel_err_mid`, is

```
rel_err_mid <- abs((true - midpoint) / true)
rel_err_mid
```

```
## [1] 0.007481266
```

which is reasonably small.

---

13. Approximate the integral of Question 12 by applying Simpson's rule with $N = 20$ to each dimension, instead of the midpoint rule.

Solution

The following create the nodes and weights needed to apply Simpson's rule.

```
N <- 20
a <- -1
b <- 1
h <- (b - a) / N
x1i <- a + h * (2 * (1:N) - 1) / 2
x2i <- a + h * (1:(N - 1))
weights <- rep(c(1, 4, 2, 1), c(1, length(x1i), length(x2i), 1))
nodes <- c(a, x1i, x2i, b)
weights <- h * weights / 6
```

```
x123 <- expand.grid(x1 = nodes, x2 = nodes, x3 = nodes)
f123 <- apply(x123, 1, prod)^2
w123 <- weights %x% weights %x% weights
```

We'll call our Simpson's rule approximation `simpson`, which can be calculated with

```
simpson <- sum(w123 * f123)
simpson
```

```
## [1] 0.2962963
```

Then its relative absolute error, `rel_err_simp`, is

```
rel_err_simp <- abs((true - simpson) / true)
rel_err_simp
```

```
## [1] 5.620504e-16
```

which is very small.

---

14. Approximate the integral

$$I = \int_0^2 \int_{-\pi}^{\pi} [\sin(x_1) + x_2 + 1] \, \mathrm{d}x_1 \mathrm{d}x_2$$

using Gaussian quadrature with $N = 9$ integration nodes per dimension and estimate the relative absolute error of your approximation.

Solution

We'll start by calculating I, the true integral,

$$\begin{aligned} I &= \int_0^2 \int_{-\pi}^{\pi} [\sin(x_1) + x_2 + 1] \, \mathrm{d}x_1 \mathrm{d}x_2 \\ &= \int_0^2 \Big\{ \big[-\cos(x_1) + x_1 x_2 + x_2 \big]_{-\pi}^{\pi} \Big\} \mathrm{d}x_2 \\ &= \int_0^2 \big[(1 + \pi x_2 + \pi) - (1 - \pi x_2 - \pi) \big] \mathrm{d}x_2 \\ &= \int_0^2 2\pi(x_2 + 1) \mathrm{d}x_2 = 2\pi \left[\frac{x_2^2}{2} + x_2 \right]_0^2 = 8\pi, \end{aligned}$$

and then store this as `true`.

```
(true <- 8 * pi)
```

```
## [1] 25.13274
```

To calculate this is `R` we want to set our number of nodes, `N`, and write a function, `f()`, to evaluate the integrand.

```
N <- 9
f <- function(x1, x2) sin(x1) + x2 + 1
```

Then we want to set our integration nodes for $x1$, `gq1`, and for $x2$, `gq2`.

```
gq1 <- pracma::gaussLegendre(N, -pi, pi)
gq2 <- pracma::gaussLegendre(N, 0, 2)
```

Then we want to evaluate `f()` at each combination of our x_1 and x_2 integration nodes, which we'll call `f12`.

```
f12 <- outer(gq1$x, gq2$x, f)
```

The weights corresponding to these nodes, `w12`, are

```
w12 <- gq1$w %o% gq2$w
```

and then the integral approximation is

```
Ihat <- sum(w12 * f12)
Ihat
```

```
## [1] 25.13274
```

Its relative absolute error, `rel_err`, is

```
rel_err <- abs((true - Ihat) / true)
rel_err
```

```
## [1] 2.82716e-16
```

and again is incredibly small.

---

15. Repeat examples 4.10 and 4.11 by avoiding `for()` loops. [You might want to consider the functions `expand.grid()` or `outer()`.]

Solution

We'll start with Example 4.10 and will use `expand.grid()`. Then we'll set the dimension `d`, the number of integration nodes per dimension, `N`, the integration nodes, `x1` and `x2`, and calculate the true value of the integral, `true`, as in the Lecture Notes.

```
d <- 2
true <- (exp(1) - 1)^d
N <- 10
x1 <- x2 <- (1:N - .5) / N
```

Then we want to use `expand.grid()` to give us all combinations of the integration nodes for both dimensions, and we'll quickly use `head()` to see what it's done.

```
x12 <- expand.grid(x1 = x1, x2 = x2)
head(x12)
```

We'll calculate the integration weights, `w`, as in the Lecture Notes,

```
w <- 1/(N^d)
```

and then approximate the integral, which we'll call `midpoint`, by taking the first column of `x12` as the x_{1i} nodes, and the second column as the x_{2j} nodes, which gives

```
midpoint <- w * sum(exp(x12[, 1] + x12[, 2]))
midpoint
```

```
## [1] 2.950033
```

and is the same as in the Lecture Notes.

For Example 4.11 we'll use `outer()` instead, using `d` and `true` from above. Then we need to set the number of integration nodes, `N`, and find the nodes for each margin, i.e. x_{1i} and x_{2j} for $i, j = 1, \ldots, N$, which we'll store as `xw`.

```
N <- 4
xw <- pracma::gaussLegendre(N, 0, 1)
```

Next we want to calculate $\exp(x_{1i} + x_{2j})$ for $i, j = 1, \ldots, N$, which we'll store as `f12`.

```
f12 <- exp(outer(xw$x, xw$x, FUN = '+'))
f12
```

```
##          [,1]     [,2]     [,3]     [,4]
## [1,] 1.148967 1.490991 2.094725 2.718282
## [2,] 1.490991 1.934829 2.718282 3.527458
## [3,] 2.094725 2.718282 3.818971 4.955800
## [4,] 2.718282 3.527458 4.955800 6.431040
```

And then we want to calculate the integration weights, $w_{ij} = w_{1i}w_{2j}$, for $i, j = 1, \ldots, N$, where the w_{1i}s and w_{2j} can both be taken from `xw$w`, and will store these as `w12`, obtaining them through `outer()`.

```
w12 <- outer(xw$w, xw$w)
w12
```

```
##            [,1]       [,2]       [,3]       [,4]
## [1,] 0.03025075 0.05671296 0.05671296 0.03025075
## [2,] 0.05671296 0.10632333 0.10632333 0.05671296
## [3,] 0.05671296 0.10632333 0.10632333 0.05671296
## [4,] 0.03025075 0.05671296 0.05671296 0.03025075
```

which uses that `FUN = '*'` is `outer()`'s default. Then we'll calculate the integral approximation, and store this as `gq`

```
gq <- sum(w12 * f12)
gq
```

```
## [1] 2.952492
```

which is the same approximation as in the Lecture Notes. [On this occasion `outer()` is perhaps a bit tidier, but is less easy to implement for $(d > 2)$-dimensional integral approximations.]

---

16. Consider a random vector $\mathbf{X} = (X_1, X_1)^{\mathrm{T}}$ from the standard bivariate normal distribution, which we write as

$$\mathbf{X} = \begin{pmatrix} X_1 \\ X_2 \end{pmatrix} \sim BVN\left(\begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 & \rho \\ \rho & 1 \end{pmatrix}\right)$$

so that $-1 \leq \rho \leq 1$ is the correlation between X_1 and X_2. The standard bivariate normal pdf is given by

$$f(\mathbf{x}; \rho) = \frac{1}{2\pi\sqrt{1-\rho^2}} \exp\left(-\frac{x_1^2 - 2\rho x_1 x_2 + x_2^2}{2(1-\rho^2)}\right)$$

but its cdf, which is given by

$$\Pr(a_1 \leq x_1 \leq b_1, a_2 \leq x_2 \leq b_2; \rho) = \int_{a_1}^{b_1} \int_{a_2}^{b_2} \frac{1}{2\pi} f(\mathbf{x}; \rho) \mathrm{d}x_1 \mathrm{d}x_2,$$

doesn't have closed form. Estimate $\Pr(-1 \leq x_1 \leq 1, -1 \leq x_2 \leq 1; 0.9)$ using the midpoint rule with $N = 10$ integration nodes per dimension, and then using Gaussian quadrature with $N = 7$ integration nodes per dimension.

Solution

We'll start with a function to evaluate $f(\mathbf{x}; \rho)$, which we'll call `f_bvn()`.

```
f_bvn <- function(x1, x2, rho) {
  # Function to evaluate bivaraiate Normal pdf
  # x1, x2 can be scalar, vector or matrix
  # rho is a scalar
  # Returns object in same format as x1
  temp <- 1 - rho^2
  out <- x1^2 - 2 * rho * x1 * x2 + x2^2
  out <- out / (2 * temp)
  exp(-out) / (2 * pi * sqrt(temp))
}
```

Then we'll write a generic function to implement the two-dimensional midpoint rule, which we'll call `integrate_bvn_midpoint()`, which is only useful for the case when we want the same integration nodes for each dimension.

```
integrate_bvn_midpoint <- function(N, a, b, f, ...) {
  # Function to evaluate a cdf by midpoint rule
  # N is an integer giving the number of integration nodes per dimension
  # a and b are scalars
  # f is the corresponding pdf
  # Returns a scalar
  edges <- seq(a, b, l = N + 1)
  h <- edges[2] - edges[1]
  mids <- edges[-1] - .5 * h
  grid <- expand.grid(x1 = mids, x2 = mids)
  sum(h * h * f(grid$x1, grid$x2, ...))
}
I_bvn_midoint <- integrate_bvn_midpoint(10, -1, 1, f_bvn, rho = .9)
I_bvn_midoint
```

```
## [1] 0.5986239
```

Next we'll repeat the process, writing a function `integrate_bvn_gq()` to approximate the integral by Gaussian quadrature.

```
integrate_bvn_gq <- function(N, a, b, f, ...) {
  # Function to evaluate a cdf by Gaussian quadrature
  # N is an integer giving the number of integration nodes per dimension
  # a and b are scalars
  # f is the corresponding pdf
  # Returns a scalar
```

```
  x1 <- x2 <- pracma::gaussLegendre(N, a, b)
  x_grid <- expand.grid(x1 = x1$x, x2 = x2$x)
  w_vec <- x1$w %x% x2$w
  sum(w_vec * f(x_grid$x1, x_grid$x2, ...))
}
I_bvn_gq <- integrate_bvn_gq(7, -1, 1, f_bvn, rho = .9)
I_bvn_gq
```

```
## [1] 0.5963602
```

Note that on this occasion, we don't know the true integral, and so can't calculate the relative absolute errors.

---

17. Repeat Example 4.12 using the midpoint rule and $N = 10$.

Solution

On this occasion, we can effectively merge code from Question 15 and Example 4.12.

```
N <- 10
d <- 5
x <- (1:N - .5) / N
xx <- lapply(1:d, function(i) x)
X <- expand.grid(xx)
w <- 1/(N^d)
f <- sum(w * exp(rowSums(X)))
true <- (exp(1) - 1)^d
c(true = true, midpoint = f, rel.err = abs((true - f) / true))
```

```
##         true     midpoint      rel.err
## 14.978626322 14.947455928  0.002080991
```

We've now got a relative absolute error about six orders of magnitude larger than for Gaussian quadrature in Example 4.12, and yet here we've had to evaluate the integrand 10^5 times, as opposed to 1024 times in Example 4.12. Here we see that when dealing with high-dimensional integrals, we may want to carefully choose our integration scheme, especially if the integrand is an expensive function to evaluate.

---

18. Consider a single random variable $Y \mid \mu \sim \mathrm{N}(\mu, \sigma^2)$, where we can characterise our prior beliefs about μ as $\mu \sim \mathrm{N}(\alpha, \beta)$. The marginal distribution of y is given by

$$f(y) = \int_{-\infty}^{\infty} f(y \mid \mu) f(\mu) \mathrm{d}\mu,$$

and that the marginal distribution of Y is $Y \sim \mathrm{N}(\alpha, \sigma^2 + \beta)$ can be derived in various ways. Approximate $f(y)$ using Laplace's method for $\sigma = 1.5$, $\alpha = 2$ and $\beta = 0.5$, and then compare the relative absolute error of your approximation to the true marginal pdf for $y = -2, 0, 2, 4, 6$.

Solution

For the Laplace approximation, we take $n = 1$ and $f()$ from Equation (4.4) as $-\log\phi(y;\mu,\sigma^2) - \log\phi(\mu;\alpha,\beta)$, where we swap x for μ. Its first derivative w.r.t. μ is $-(y-\mu)/\sigma^2 + (\mu-\alpha)/\beta$ and its second derivative w.r.t. μ is $1/\sigma^2 + 1/\beta$. Setting the first derivative to zero gives $\hat{\mu} = (y\beta + \alpha\sigma^2)/(\beta + \sigma^2)$. Our Laplace approximation then

$$\begin{aligned}\hat{f}(y) &\simeq \sqrt{\frac{2\pi}{\frac{1}{\sigma^2} + \frac{1}{\beta}}} \exp\left\{-\log\phi(y;\hat{\mu},\sigma^2) - \log\phi(\hat{\mu};\alpha,\beta)\right\} \\ &= \sqrt{\frac{2\pi}{\frac{1}{\sigma^2} + \frac{1}{\beta}}} \phi(y;\hat{\mu},\sigma^2)\phi(\hat{\mu};\alpha,\beta).\end{aligned}$$

We can evaluate this in R with the following. We'll start by setting σ, α and β.

```
alpha <- 2
beta <- .5
sigma <- 1.5
```

Then we'll create `fy()`, which evaluates the *true* marginal pdf, i.e. $f(y)$, given y, σ, α and β,

```
fy <- function(y, sigma, alpha, beta) {
  # Function to evaluate Normal(alpha, beta = sigma^2) pdf
  # y is scalar, vector or matrix
  # sigma, alpha and beta are scalars
  # returns objects in same format as y
  dnorm(y, alpha, sqrt(beta + sigma^2))
}
```

evaluate this for the y values specified, `y_vals`,

```
y_vals <- seq(-2, 6)
```

and store this as `f_true`

```
f_true <- fy(y_vals, sigma, alpha, beta)
```

The following function calculates the Laplace approximation, by finding $\hat{\mu}$, called `mu_hat`, and then substituting this into the approximation above.

```
fla <- function(y, sigma, alpha, beta) {
  # Function to compute Laplace approximate for Normal
  # prior on mean  mu and Normal likelihood on data
  # y is scalar or vector
  # sigma, alpha and beta are scalars
  # Returns object of same format as y
  mu_hat <- (y * beta + alpha * sigma^2) / (beta + sigma^2)
  mult <- 2 * pi / abs(1 / beta + 1 / sigma^2)
  sqrt(mult) * dnorm(y, mu_hat, sigma) * dnorm(mu_hat, alpha, sqrt(beta))
}
```

Then we use this to evaluate the approximation for the specified y values, `f_la`.

```
f_la <- fla(y_vals, sigma, alpha, beta)
```

We'll look at these alongside each other

```
rbind(true = f_true, laplace = f_la)
```

```
##                [,1]       [,2]      [,3]     [,4]      [,5]     [,6]      [,7]
## true    0.01311721 0.04683602 0.1162501 0.200577 0.2405712 0.200577 0.1162501 0.0468
## laplace 0.01311721 0.04683602 0.1162501 0.200577 0.2405712 0.200577 0.1162501 0.0468
##                [,9]
## true    0.01311721
## laplace 0.01311721
```

and then by calculating the relative absolute error, i.e.

```
abs((f_true - f_la) / f_true)
```

```
## [1] 6.612393e-16 0.000000e+00 1.193788e-16 1.383787e-16 1.153736e-16 1.383787e-16
## [7] 0.000000e+00 0.000000e+00 1.322479e-16
```

we see that the relative absolute error is effectively zero, once we allow for R's machine tolerance. [In fact if we carry on with the algebra above by substituting in the expression for the Normal pdfs, i.e.

$$I_n = \sqrt{\frac{2\pi}{\frac{1}{\sigma^2}+\frac{1}{\beta}}}\left(\frac{1}{\sqrt{2\pi\sigma^2}}\exp\left\{-\frac{(y-\hat{\mu})^2}{2\sigma^2}\right\}\right)\left(\frac{1}{\sqrt{2\pi\beta}}\exp\left\{-\frac{(\hat{\mu}-\alpha)^2}{2\beta}\right\}\right),$$

we eventually find that on this occasion the Laplace approximation is exact (which you're welcome to confirm in your own time).]

---

19. The negative binomial distribution, denoted $\text{NegBin}(r,p)$, has probability mass function (pmf)

$$f(y) = \frac{\Gamma(y+r)}{y!\Gamma(r)}p^y(1-p)^r.$$

If $Y \mid \lambda \sim \text{Poisson}(\lambda)$ and $\lambda \sim \text{Gamma}(r,[1-p]/p)$, then $Y \sim \text{NegBin}(r,p)$, so that the marginal distribution of Y is negative binomial, i.e.

$$f(y) = \int_{-\infty}^{\infty} f(y \mid \lambda) f(\lambda) \mathrm{d}\lambda \tag{7}$$

if $f(y \mid \lambda)$ is the $\text{Poisson}(\lambda)$ pmf and $f(\lambda)$ is the $\text{Gamma}(\alpha,\beta)$ pdf with $\alpha = r$ and $\beta = (1-p)/p$. Use Laplace's method to approximate the marginal pmf, i.e. to approximate $f(y)$, for $y = 0,1,\ldots,20$, $\alpha = 5$ and $\beta = 0.5$.

Solution

For Laplace's method, we first note that

$$\begin{aligned} f(y,\lambda) &= f(y \mid \lambda) f(\lambda) \\ &= \mathrm{e}^{-\lambda}\frac{\lambda^y}{y!} \times \frac{\lambda^{\alpha-1}\mathrm{e}^{-\beta\lambda}\beta^\alpha}{\Gamma(\alpha)} = \mathrm{e}^{-\lambda(1+\beta)}\frac{\lambda^{y+\alpha-1}\beta^\alpha}{y!\Gamma(\alpha)} \end{aligned}$$

Then, considering only λ,

$$\log f(y,\lambda) = -\lambda(\beta+1) + (y+\alpha-1)\log(\lambda) + \text{constant}$$

and so

$$\frac{\partial \log f(y,\lambda)}{\partial \lambda} = -(\beta+1) + (y+\alpha-1)/\lambda$$

which we set to zero at $\tilde{\lambda}$ so that $\beta + 1 = (y + \alpha - 1)/\tilde{\lambda}$, i.e.

$$\tilde{\lambda} = \frac{y + \alpha - 1}{\beta + 1}.$$

Then

$$\frac{\partial^2 \log f(y, \lambda)}{\partial \lambda^2} = -(y + \alpha - 1)/\lambda^2.$$

Using the formula for Laplace's method from the notes, we take $-nf(x)$ as $\log f(y, \lambda)$ so that

$$f(y) \simeq \mathrm{e}^{\log f(y, \tilde{\lambda})} \sqrt{\frac{2\pi}{nf''(\tilde{\lambda})}}$$

where $f''(\tilde{\lambda}) = -\partial^2 \log f(y, \lambda)/\partial \lambda^2$. Therefore

$$\begin{aligned} f(y) &\simeq \mathrm{e}^{-\tilde{\lambda}(1+\beta)} \frac{\tilde{\lambda}^{y+\alpha-1} \beta^\alpha}{y! \Gamma(\alpha)} \sqrt{\frac{2\pi \tilde{\lambda}^2}{y + \alpha - 1}} \\ &= \mathrm{e}^{-\tilde{\lambda}(1+\beta)} \frac{\tilde{\lambda}^{y+\alpha} \beta^\alpha}{y! \Gamma(\alpha)} \sqrt{\frac{2\pi}{y + \alpha - 1}}. \end{aligned}$$

Next we'll set the y values and α and β.

```
y <- 0:20
alpha <- 5
beta <- .5
```

The following function can then give the Laplace approximation to $f(y)$ for given y, α and β.

```
fhat <- function(y, alpha, beta) {
  # Function to give Laplace approximation to marginal distribution
  # y can be scalar or vector
  # alpha and beta are scalar
  # returns scalar if y scalar, and vector if y vector
  tlambda <- (y + alpha - 1) / (beta + 1)
  dpois(y, tlambda) * dgamma(tlambda, alpha, beta) *
    tlambda * sqrt(2 * pi / (y + alpha - 1))
}
```

Then we can evaluate this for the specific values of y.

```
fhat(y, alpha, beta)
```

```
##  [1] 0.004030551 0.013490989 0.027056782 0.042171720 0.056312548 0.067653206 0.07523
##  [8] 0.078882315 0.078932092 0.076049269 0.071011807 0.064581799 0.057425971 0.05007
## [15] 0.042936547 0.036266361 0.030228594 0.024899075 0.020291794 0.016378531 0.01310
```

It's useful to plot this against the NegBin(α, p) pmf. First we'll write a function `dnbinom2(y, r, p)` to evaluate this pmf

```
dnbinom2 <- function(y, r, p)
  gamma(y + r) * p^y * (1 - p)^r / factorial(y) / gamma(r)
```

and then we'll plot it against the Laplace approximation.

```
fy <- dnbinom2(y, alpha, 1 / (beta + 1))
plot(y, fy, ylim = c(0, max(fy)))
lines(y, fhat(y, alpha, beta), col = 2)
```

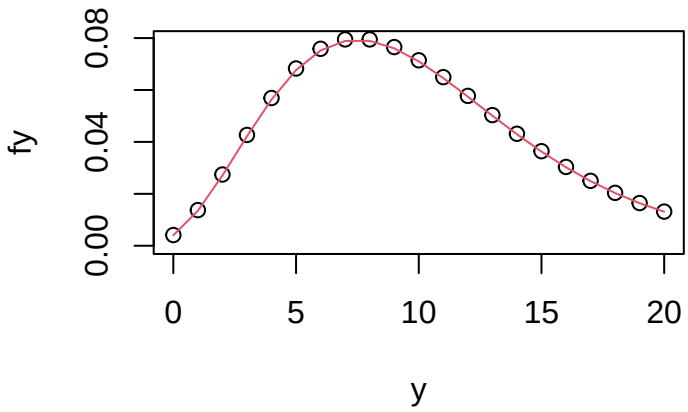


Figure 4: Comparison of true negative binomial marginal pmf with its Laplace approximation.

The plot shows agreement between the true marginal pmf and its Laplace approximation to be very good.

---

20. Consider an observation y from the pdf

$$f(y \mid \omega) = \frac{1}{\sqrt{2\pi\omega}} \exp\left(-\frac{y^2}{2\omega}\right),$$

where $\omega > 0$ with

$$f(\omega) = \frac{1}{\omega}\sqrt{\frac{2}{\pi}} \exp\left(-2(\log\omega)^2\right).$$

Approximate $f(y) = \int f(y, \omega)\mathrm{d}\omega$ using Laplace's method for $y = -3.0, -2.5, -2.0, \ldots, 2.0, 2.5, 3.0$.

Solution

For Laplace's method, we first note that

$$\begin{aligned} f(y, \omega) &= f(y \mid \omega) f(\omega) \\ &= \frac{1}{\sqrt{2\pi\omega}} \exp\left(-\frac{y^2}{2\omega}\right) \cdot \frac{1}{\omega}\sqrt{\frac{2}{\pi}} \exp\left\{-2(\log\omega)^2\right\} \\ &= \frac{\omega^{-3/2}}{\pi} \exp\left\{-\frac{y^2}{2\omega} - 2(\log\omega)^2\right\}. \end{aligned}$$

Then

$$\log f(y, \omega) = -\frac{3}{2}\log\omega - \log\pi - \frac{y^2}{2\omega} - 2(\log\omega)^2.$$

can be evaluated in R with

```
d0 <- function(omega, y) {
  # function to evaluate log joint pdf
  # omega and y are scalars
  # returns a scalar
```

```
  lomega <- log(omega)
  - 1.5 * lomega - log(pi) - .5 * y^2 / omega - 2 * lomega^2
}
```

Then $\tilde{\omega} = \arg\max_\omega \log f(y, \omega)$. However, the first deriative

$$\frac{\partial \log f(y,\omega)}{\partial \omega} = \frac{y^2}{2\omega^2} - \frac{1}{\omega}\left[\frac{3}{2} + 4\log\omega\right]$$

does not have a closed-form solution to its root. Instead, we find its root, $\tilde{\omega}$, numerically. We'll write an `R` function for this

```
d1_omega <- function(omega, y) {
  # function to calculate first derivative of joint pdf w.r.t. omega
  # omega and y are scalars
  # returns a scalar
  y^2 / (2 * omega^2) - (1.5 + 4 * log(omega)) / omega
}
give_tilde_omega <- function(y) {
  # function to give root of first derivative of joint pdf w.r.t. omega
  # y is a scalar
  # returns a scalar
  uniroot(d1_omega, c(.001, 10), y = y)$root
}
```

which recognises that we expect ω to be in $[0.001, 10]$ (which might sometimes require some trial and error, or athematical derivation) and we'll later use inside a function to calculate the Laplace approximation. For the Laplace approximation, we'll need a function to evaluate the second derivative of the joint pdf w.r.t. ω, which, given that

$$\frac{\partial^2}{\partial \omega^2} \log f(\omega \mid y) = \frac{1}{\omega^2}\left[4\log(\omega) - \frac{5}{2}\right] - \frac{y^2}{\omega^3}$$

can be evaluated in `R` with

```
d2_omega <- function(omega, y) {
  # function to calculate second derivative of joint pdf w.r.t. omega
  # omega and y are scalars
  # returns a scalar
  3 / (2 * omega^2) - y^2 / omega^3 - 4 / omega^2 + 4 * log(omega) / omega^2
}
```

Then our Laplace approximation can be calculated with the function

```
la_val <- function(y) {
  tilde_omega <- give_tilde_omega(y)
  exp(d0(tilde_omega, y)) * sqrt(2 * pi / abs(d2_omega(tilde_omega, y)))
}
```

which we might want to put inside a wrapper that supports a vector of y values

```
la <- function(y) {
  sapply(y, la_val)
}
```

Finally, we can evaluate the Laplace approximation at the specified values with

```
y_vals <- seq(-3, 3, by = .5)
la(y_vals)
```

```
##  [1] 0.008649151 0.021233632 0.049663850 0.106991453 0.201798613 0.310870633 0.36323
##  [8] 0.310870633 0.201798613 0.106991453 0.049663850 0.021233632 0.008649151
```

A slightly useful code check is to plot the Laplace approximation against the joint pdf for a specified value of ω, which in this case will be $\omega = 1$, i.e.

```
y_vals <- seq(-3, 3, by = .5)
matplot(y_vals, cbind(dnorm(y_vals, 0, 1), la(y_vals)), type = 'l', lty = 1)
```

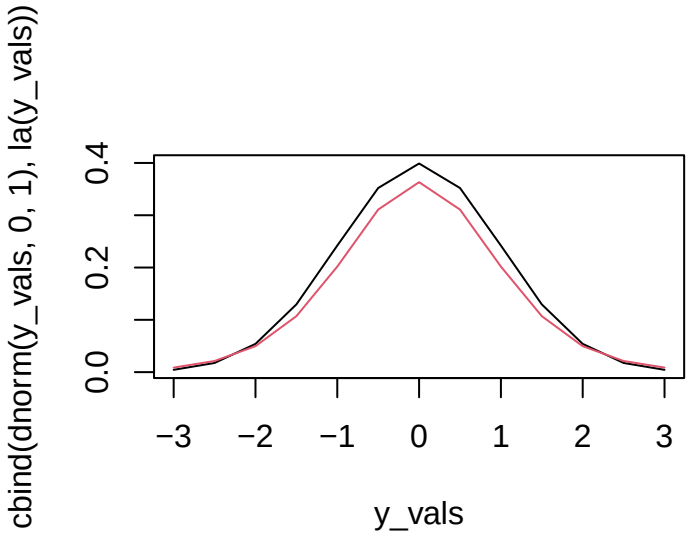


Figure 5: To do.

and we see that they're close, but the Laplace approximation has more spread, which is to be expected because ω is a random variable, and not fixed at one.

---

21. Recall the integral

$$I = \int_0^1 [1 + 2\cos(2x) + 2\sin(2x)]\mathrm{d}x$$

from above. Use Monte Carlo integration to approximate I with Monte Carlo samples of size $N = 100$, 1000 and 10000 and calculate the relative absolute error of each of your approximations.

Solution

For a given value of N, we want to generate N Uniform($[0, 1]$) random variates, evaluate the integrand at each, and take the mean as the approximation to I. The following code approximates the integral as `I_hat` for for $N = 100$

```
N <- 100
x <- runif(N)
fx <- 1 + 2 * cos(2 * x) + 2 * sin(2 * x)
I_hat <- mean(fx)
I_hat
```

```
## [1] 3.330527
```

and its relative absolute error, `rel_err`, is

```
true <- 2 + sin(2) - cos(2)
rel_err <- abs((true - I_hat) / true)
rel_err
```

```
## [1] 0.001528359
```

The following code then loops this over the N values $N = 100$, 1000 and 10000.

```
N_vals <- c(100, 1000, 10000)
I_hat <- rel_err <- numeric(length(N_vals))
for (i in 1:length(N_vals)) {
  N <- N_vals[i]
  x <- runif(N)
  fx <- 1 + 2 * cos(2 * x) + 2 * sin(2 * x)
  I_hat[i] <- mean(fx)
  rel_err[i] <- abs((true - I_hat[i]) / true)
}
```

which gives the integral approximations

```
I_hat
```

```
## [1] 3.316098 3.311553 3.325624
```

and corresponding relative absolute errors

```
rel_err
```

```
## [1] 2.810404e-03 4.177179e-03 5.404947e-05
```

We might expect that our relative absolute errors will decrease as N increases. Probabilistically they will: but our approximations are subject to the variability in the random samples, and so sometimes a smaller Monte Carlo sample size may give a better approximation than a larger sample size. In general, though, we should rely on the probabilistic properties, and use the largest possible Monte Carlo sample size that is feasible.

---

22. If $Y \sim \text{Uniform}([a, b])$ then

$$E(Y) = \int_a^b \frac{u}{b-a} \mathrm{d}u = \frac{1}{2}(a+b).$$

Use the mean of $m = 1000$ Monte Carlo replicates of Monte Carlo samples of size $N = 500$ to approximate the above integral, and hence to approximate $E(Y)$, for $a = -1$ and $b = 1$.

Solution

We'll start by setting the Monte Carlo sample size, `N`, and then the number of replicates, `m`.

```
N <- 500
m <- 1e3
```

For a given replicate we need to generate N points from the Uniform[-1, 1] distribution, $u_1^*, \ldots, u_N^*$, say, and then our approximation to the integral is given by $N^{-1} \sum_{i=1}^N u_i^*/2$, as $b - a = 2$.

The following calculates this for a single replicate, where $a =$ `a` and $b =$ `b`,

```
a <- -1
b <- 1
u <- runif(N, -1, 1)
mean(u) / (b -a)
```

```
## [1] -0.001125121
```

and then `replicate()` can perform the m replicates

```
uu <- replicate(m, runif(N, -1, 1))
dim(uu)
```

```
## [1]  500 1000
```

which stored the sampled values in a $N \times m$ matrix `uu`. Then we can use `colMeans()` to calculate the mean of each sample, and `mean()` to take the mean over replicates.

```
mean(colMeans(uu) / (b - a))
```

```
## [1] 0.0005700557
```

[Note that due to the independence of the samples, this is equivalent to

```
mean(uu) / (b - a)
```

```
## [1] 0.0005700557
```

and so m Monte Carlo samples of size N is equivalent to one Monte Carlo sample of size mN.]

---

23. Approximate the integral the earlier integral

$$I = \int_0^2 \int_{-\pi}^{\pi} [\sin(x_1) + x_2 + 1] \, \mathrm{d}x_1 \mathrm{d}x_2$$

using Monte Carlo with samples of size $N = 100$, 200 and 400 and $m = 1000$ replicates for each. Calculate the mean and variance of your approximations for each value of N, i.e. across replicates. What do you observe?

Solution

We'll start with a single replicate for $N = 100$, generating x_1 from the Uniform($[-\pi, \pi]$) distribution and x_2 from the Uniform($[0, 2]$) distribution. Let $x^*_{11}, \ldots, x^*_{1N}$ and $x^*_{21}, \ldots, x^*_{2N}$ denote these respective samples. Then the integral approximation is given by

$$\hat{I} = \frac{4\pi}{N} \sum_{i=1}^{N} \left[\sin(x^*_{1i}) + x^*_{2i} + 1 \right]$$

as our domain for (x_1, x_2), denoted by $\mathcal{X}$ in the Lecture Notes, is $[0, 2] \times [-\pi, \pi]$, and hence $V(\mathcal{X}) = 4\pi$. The following then calculates this in `R`.

```
N <- 100
x1 <- runif(N, -pi, pi)
x2 <- runif(N, 0, 2)
f <- function(x1, x2) sin(x1) + x2 + 1
4 * pi * mean(f(x1, x2))
```

```
## [1] 26.53961
```

Now we can to replicate this m times for the different N values. For explicitness, we'll use loops.

```
N_vals <- c(100, 200, 400)
m <- 1000
I_hat <- matrix(NA, length(N_vals), m)
for (i in 1:length(N_vals)) {
  N <- N_vals[i]
  for (j in 1:m) {
    x1 <- runif(N, -pi, pi)
    x2 <- runif(N, 0, 2)
    I_hat[i, j] <- 4 * pi * mean(f(x1, x2))
  }
}
```

The following give the mean and variance across replicates.

```
rowMeans(I_hat)
```

```
## [1] 25.06577 25.08629 25.16046
```

```
apply(I_hat, 1, sd)
```

```
## [1] 1.1376463 0.8392819 0.6005032
```

Our main observation here is that the variance decreases as N increases, and roughly halves as N doubles, which is consistent with the formula for $\text{Var}(\hat{I}_{\mathcal{X}})$ given in the Lecture Notes, i.e. being inversely proportional to N.

---

24. Approximate

$$I = \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1}(x_1x_2x_3)^2\mathrm{d}x_1\mathrm{d}x_2\mathrm{d}x_3$$

using a Monte Carlo sample of size $N = 10^4$ and evaluate the approximation's relative absolute error.

Solution

Let's start by finding the true integral

$$I = \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1}(x_1x_2x_3)^2\mathrm{d}x_1\mathrm{d}x_2\mathrm{d}x_3 = \int_{-1}^{1}\int_{-1}^{1}\frac{2}{3}\mathrm{d}x_2\mathrm{d}x_3 = \ldots = \left(\frac{2}{3}\right)^3 = \frac{8}{27}.$$

We'll store this as `true`.

```
true <- 8 / 27
```

Then we want to generate three sets of N draws from the Uniform([-1, 1]) distribution, representing x_1, x_2 and x_3, which we'll store as `x123`.

```
N <- 1e4
x123 <- matrix(runif(3 * N, -1, 1), N)
```

Our domain is $[-1, 1]^3$ which has volume $2^3 = 8$. Our approximation to the integral, `I_hat`, is

```
I_hat <- 8 * mean(apply(x123^2, 1, prod))
I_hat
```

```
## [1] 0.2993306
```

which has a relative absolute error, `rel_err`, of

```
rel_err <- abs((true - I_hat) / true)
rel_err
```

```
## [1] 0.01024071
```

i.e. about 1%.

---

25. Approximate the negative binomial marginal distribution given in Equation (7) using Monte Carlo sampling with $N = 10^3$, $\alpha = 5$, $\beta = 0.5$ and for $y = 0, 1, \ldots, 20$, and calculate the relative absolute error of your approximation.

Solution

We'll start by storing N, α, β and $y = 0, 1, \ldots, 20$.

```
N <- 1e3
alpha <- 5
beta <- .5
y_vals <- 0:20
```

Then we want to generate N values for λ from the Gamma(α, β) distribution, which we'll call `lambda_samp`.

```
lambda_samp <- rgamma(N, alpha, beta)
```

Our approximation to the marginal distribution is obtained by evaluating the Poisson(λ) distribution for the y values specified at each sampled λ value, which we'll call `fy_samp`,

```
fy_samp <- sapply(lambda_samp, function(lambda) dpois(y_vals, lambda))
```

and then averaging over the samples, which we'll call `fy_hat`.

```
fy_hat <- rowMeans(fy_samp)
fy_hat
```

```
##  [1] 0.003646912 0.012868946 0.026607133 0.042330466 0.057387922 0.069712704 0.07805
##  [8] 0.081994808 0.081816047 0.078298252 0.072440391 0.065222290 0.057453215 0.04971
## [15] 0.042382039 0.035655861 0.029631056 0.024332628 0.019746861 0.015838362 0.01255
```

The relative absolute error, `rel_err`, relative to the true marginal pdf, `fy`, can be caluclated with

```
fy <- dnbinom2(y_vals, alpha, 1 / (beta + 1))
rel_err <- abs((fy - fy_hat) / fy)
rel_err
```

```
##  [1] 0.113800441 0.061853809 0.030169988 0.008106467 0.008541485 0.020948770 0.02882
```

```
##  [8] 0.031614646 0.029365575 0.022995284 0.014064485 0.004323243 0.004722164 0.01209
## [15] 0.017473383 0.021136090 0.023842795 0.026619715 0.030533783 0.036477436 0.04499
```

and is reasonably small for each y value. Finally, as in question 19, we'll plot the approximation against the true marginal pdf

```
plot(y_vals, fy, ylim = c(0, max(fy_hat)), xlab = 'y', ylab = 'f(y)')
lines(y, fy_hat, col = 2)
```

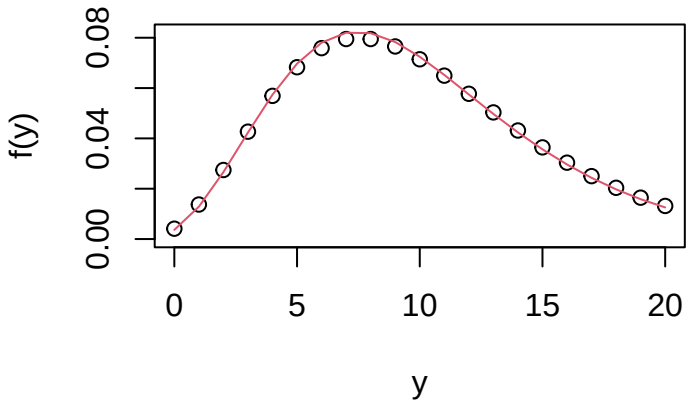


Figure 6: Comparison of true negative binomial marginal pmf with its Monte Carlo approximation.

and see that the approximation (in red) closely follows the true marginal pdf.

---

26. Repeat Question 20 by Monte Carlo integration using a Monte Carlo sample of size $N = 1000$, given that $f(y \mid \omega)$ is the $N(0, \omega)$ pdf and that draws from $f(\omega)$ can be obtained with

```
romega <- function(n) exp(rnorm(n, 0, 0.5))
```

Solution

```
N <- 1e3
omegas <- romega(N)
y_vals <- seq(-3, 3, by = .5)
fy <- sapply(y_vals, function(y) mean(dnorm(y, 0, omegas)))
fy
```

```
##  [1] 0.01797765 0.03123875 0.05602439 0.10343296 0.19313821 0.34219194 0.45626573 0.
##  [9] 0.19313821 0.10343296 0.05602439 0.03123875 0.01797765
```

A quick plot confirms what we saw in Question 20.

```
matplot(y_vals, cbind(dnorm(y_vals, 0, 1), fy),
        xlab = 'y', ylab = 'pdf', type = 'l', lty = 1)
legend('topright', legend = c('N(0, 1) pdf', 'Laplace approx.'), lty = 1, col = 1:2)
```

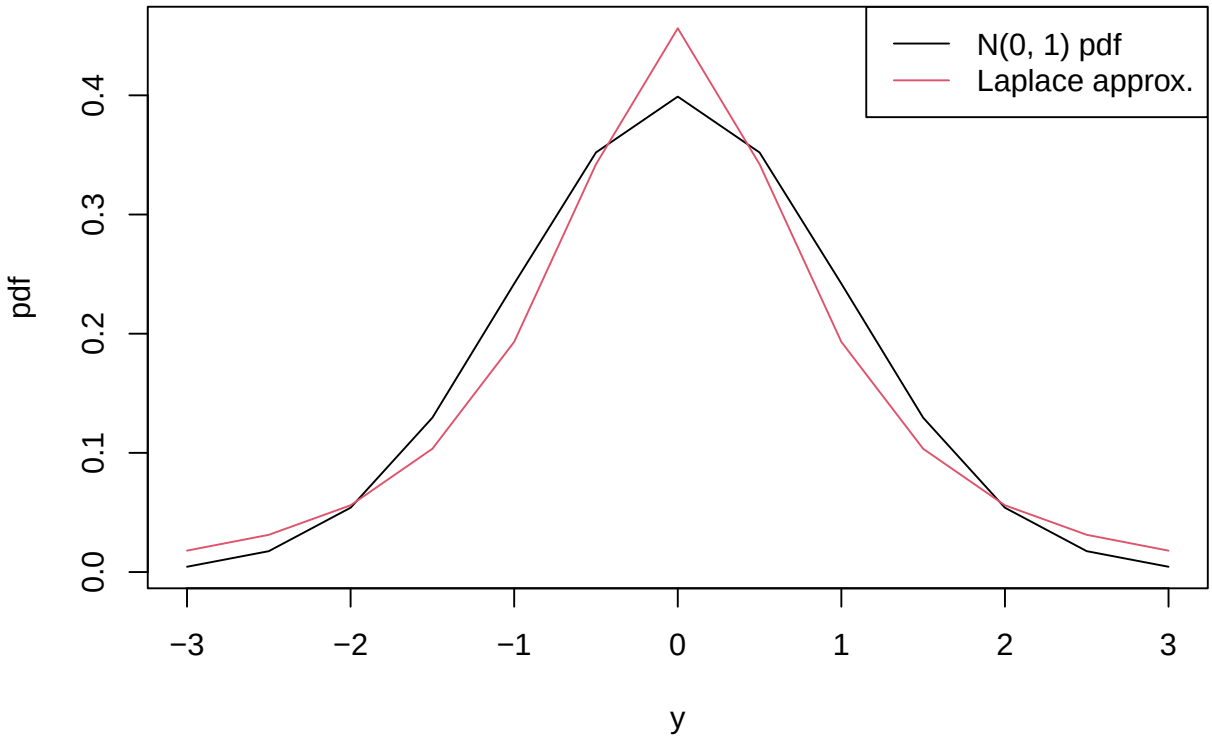


Figure 7: Comparison of N(0, 1) pdf to Laplace approximation of marginal pdf.

5 Chapter 5 exercises

1. Suppose that we want to generate random variables from a distribution with cdf

$$F(y) = \frac{2}{\pi} \arcsin(\sqrt{y}) \quad \text{for } y \in [0, 1].$$

We can generate a single random variable, Y^*, say, by generating a random Uniform([0, 1]) random variable, U^*, say, and then finding Y^* such that $F(Y^*) = U^*$. Write a function in `R`, `rarcsine(n)`, that generates n random variables with cdf $F(y)$ above using `R`'s `uniroot()` function, and then generate $n = 10$ variates. Note that `asin(y)` evaluates $\arcsin(y)$ in `R` for $y =$ `y`.

Solution

We'll start by writing a function to evaluate $F()$, called `F`.

```
F <- function(y) 2 * asin(sqrt(y)) / pi
```

Then we want the function for which we want the root, which we'll call `F_root()`.

```
F_root <- function(y, u) F(y) - u
```

Next we'll use `uniroot()` to find y such that $F(y) = u$.

```
uniroot(F_root, c(0, 1), u = runif(1))$root
```

```
## [1] 0.9324378
```

Putting this together into function `rarcsine(n)`, we get

```r
qarcsin <- function(n) {
  replicate(n, uniroot(F_root, c(0, 1), u = runif(1))$root)
}
```

and so

```r
qarcsin(10)
```

```
##  [1] 0.0236656809 0.5095609153 0.0001006283 0.5565401351 0.6430583868 0.0258865154
##  [7] 0.0388274516 0.0058215727 0.9420697860 0.6765783694
```

gives $n = 10$ random variates.

---

2. Consider that the independent sample $\mathbf{y} = (y_1, \ldots, y_n)$ is from the pdf

$$f(y) = 2\theta y \exp\left\{-\theta y^2\right\} \quad \text{for } y > 0$$

with parameter $\theta > 0$.

(a) Show that the maximum likelihood estimate of θ is given by $\hat{\theta} = n/(\sum_{i=1}^n y_i^2)$.

Solution

The log-likelihood is

$$\log f(\mathbf{y} \mid \theta) = n \log(\theta) - \theta \sum_{i=1}^n y_i^2 + \text{constant}$$

and so

$$\frac{\partial \log f(\mathbf{y} \mid \theta)}{\partial \theta} = \frac{n}{\theta} - \sum_{i=1}^n y_i^2.$$

Setting $\partial \log f(\mathbf{y} \mid \theta)/\partial\theta = 0$ gives $\hat{\theta} = n/\sum_{i=1}^n y_i^2$.

(b) Given that $y_1, \ldots, y_n$ take the values

$$0.15, 0.24, 0.33, 0.43, 0.49, 0.57, 0.57, 0.63, 0.71, 0.93, 1.15, 1.22, 1.23, 1.23, 1.28$$

find $\hat{\theta}$.

Solution

We can calculate this in R with

```r
# y <- c(...) # read in y
theta_hat <- length(y) / sum(y^2)
```

and so $\hat{\theta} = 1.428$.

(c) Use R's `optimize()` function to verify $\hat{\theta}$, assuming $\hat{\theta} \in [0.1, 4]$.

Solution

We now need a function to evaluate the negative log-likelihood (ignoring the constant, because this doesn't vary with θ), as we want to find θ that minimises this. We'll call this function `nd0(theta, y)`.

```
nd0 <- function(theta, y) theta * sum(y^2) - n * log(theta)
```

Then we call `optimize()` accordingly

```
optimize(nd0, c(.1, 4), y = y)
```

```
## $minimum
## [1] 1.427901
##
## $objective
## [1] 9.656731
```

and we see that element `minimum` in the list confirms $\hat{\theta}$.

---

3. Consider using Newton's method to find $\hat{\theta}$ from Question 2.

   (a) Find the second derivative of the log-likelihood w.r.t. θ.

Solution

The second derivative of the log-likelihood is

$$\frac{\partial^2 \log f(\mathbf{y} \mid \theta)}{\partial \theta^2} = -\frac{n}{\theta^2}.$$

   (b) To find maximum likelihood estimates using Newton's method, we want to minimise the negative log-likelihood. Hence show that if we want to minimise $-\log f(\mathbf{y} \mid \theta)$ then a step of Newton's method, given θ, is given by

$$\frac{\partial[-\log f(\mathbf{y} \mid \theta)]/\partial\theta}{\partial^2[-\log f(\mathbf{y} \mid \theta)]/\partial\theta^2} = \frac{\sum_{i=1}^n y_i^2 - n/\theta}{n/\theta^2}.$$

Solution

The step is

$$\frac{\partial[-\log f(\mathbf{y} \mid \theta)]/\partial\theta}{\partial^2[-\log f(\mathbf{y} \mid \theta)]/\partial\theta^2} = \frac{\sum_{i=1}^n y_i^2 - n/\theta}{n/\theta^2}$$

as required.

   (c) Write functions in `R`, `nd1(theta, y)` and `nd2(theta, y)`, that return the first and second derivatives of $-\log f(\mathbf{y} \mid \theta)$ w.r.t. θ based on analytical expressions for the derivatives.

Solution

Here we'll ensure that `nd1()` and `nd2()` return a vector and matrix, respectively, for generality if we're working with higher dimensions, but returning scalars is also fine here.

```
nd1 <- function(theta, y) {
  # Function to evaluate first derivative w.r.t. theta
  # theta is a scalar
  # y is a vector
  # returns a 1-vector
  as.vector(sum(y^2) - n / theta)
}

nd2 <- function(theta, y) {
  # Function to evaluate second derivative w.r.t. theta
  # theta is a scalar
  # y is a vector
  # returns a 1x1 matrix
  matrix(n / theta^2, 1, 1)
}
```

(d) Use your functions from Question 3(c) to find the first Newton step, given $\theta_0 = 1$ and the sample of data from Question 2(b).

Solution

We'll load the data

```
# y <- ...
theta_0 <- 1
```

and then first step is

```
-solve(nd2(theta_0, y), nd1(theta_0, y))
```

```
## [1] 0.29968
```

(e) Perform four further steps of Newton's method. After how many steps does Newton's method agree with $\hat{\theta}$ from Question 2(b) to within three decimal places.

Solution

The following code performs five iterations of Newton's method in total, starting at θ_0.

```
iterations <- 5
theta_i <- numeric(iterations + 1)
theta_i[1] <- 1
for (i in 1 + 1:iterations) {
  theta_i[i] <- theta_i[i - 1] - solve(nd2(theta_i[i - 1], y), nd1(theta_i[i - 1],
```

```
}
theta_i
```

```
## [1] 1.000000 1.299680 1.416402 1.427826 1.427919 1.427919
```

```
theta_hat <- n / sum(y^2)
theta_hat
```

```
## [1] 1.427919
```

If we compare these to $\hat{\theta}$ by calculating the absolute difference we get

```
abs(theta_i - theta_hat)
```

```
## [1] 4.279187e-01 1.282387e-01 1.151687e-02 9.288927e-05 6.042653e-09 0.000000e+0
```

which is less than 10^{-3} by its fourth element, i.e. the third iteration of Newton's method.

(f) Evaluate the Hessian of the negative log-likelihood at $\hat{\theta}$. This should be positive definite if $\hat{\theta}$ is the maximum likelihood estimate, which can be assessed by all of its eigenvalues being positive. Check whether this is the case.

Solution

The following calculates the Hessian and its eigenvalues

```
H <- nd2(theta_hat, y)
ev <- eigen(H, symmetric = TRUE, only.values = TRUE)
```

and we can then check whether they're all positive

```
all(ev$values > 0)
```

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

which they are.

---

4. Suppose that the sample of data

$$-7.7, -0.5, -0.1, 1.2, 1.3, 2.4, 3.7, 5.7, 6.2, 8.4, 13.9, 24.4$$

can be modelled as independent realisations from the pdf

$$f(y \mid \mu) = \frac{2}{\pi\left[(y-\mu)^2 + 4\right]},$$

where $-\infty < \mu < \infty$ is an unknown parameter.

(a) Find the maximum likelihood estimate of μ, denoted $\hat{\mu}$, using Newton's method with a suitable numbers of iterations.

Solution

We'll start by finding the log-likelihood and its first and second derivatives w.r.t. μ. The log-likelihood is given by

$$\log f(\mathbf{y} \mid \mu) = n \log 2 - n \log \pi - \sum_{i=1}^{n} \log \left([y_i - \mu]^2 + 4 \right)$$

and so

$$\frac{\partial \log f(\mathbf{y} \mid \mu)}{\partial \mu} = 2 \sum_{i=1}^{n} \frac{y_i - \mu}{(y_i - \mu)^2 + 4}$$

and therefore

$$\frac{\partial^2 \log f(\mathbf{y} \mid \mu)}{\partial \mu^2} = 4 \sum_{i=1}^{n} \left[\frac{y_i - \mu}{(y_i - \mu)^2 + 4} \right]^2 - 2 \sum_{i=1}^{n} \frac{1}{(y_i - \mu)^2 + 4}.$$

Then we'll write functions in `R` to evaluate these, `d0(mu, y, mult = 1)`, `d1(mu, y, mult = 1)` and `d2(mu, y, mult = 1)`, respectively, where `mult` can be set to `-1` when we want to deal with the negative log-likelihood.

```
d0 <- function(mu, y, mult = 1) {
  # function to evaluate log-likelihood
  # mu is a scalar
  # y is a vector
  # returns a scalar
  n <- length(y)
  mult * (n * log(2) - n * log(pi) - sum(log((y - mu)^2 + 4)))
}

d1 <- function(mu, y, mult = 1) {
  # function to evaluate first derivative of log-likelihood w.r.t. mu
  # mu is a scalar
  # y is a vector
  # returns a scalar
  n <- length(y)
  mult * (2 * sum((y - mu) / ((y - mu)^2 + 4)))
}

d2 <- function(mu, y, mult = 1) {
  # function to evaluate second derivative of log-likelihood w.r.t. mu
  # mu is a scalar
  # y is a vector
  # returns a scalar
  n <- length(y)
  n * log(2) - n * log(pi) - sum(log((y - mu)^2 + 4))
  2 * mult * (2 * sum(((y - mu) / ((y - mu)^2 + 4))^2) -
    sum(1 / ((y - mu)^2 + 4)))
}
```

Next we'll perform our iterations of Newton's method. We'll deem a 'suitable' number of iterations to be when the parameter estimates for one iteration are within 10^{-4} of those of the previous iteration.

```
theta_i <- 1 # set theta_0
while(TRUE) {
  theta_last <- theta_i[length(theta_i)]
  theta_next <- theta_last - solve(d2(theta_last, y2, -1), d1(theta_last, y2, -1))
  theta_i <- c(theta_i, theta_next)
  if (abs(theta_last - theta_next) < 1e-4)
    break
}
theta_i
```

```
## [1] 1.000000 2.099045 2.215480 2.219332 2.219336
```

We see that our 4th iteration has met our stopping criteria. Hence we have $\hat{\theta} = 2.2193$, to four decimal places.

(b) Then use `optimize()` to verify your estimate of μ based on Newton's method from Question 4(a).

Solution

For `optimize()` we can use the following call, and will assume $\hat{\theta} \in [1, 3]$.

```
optimize(d0, c(1, 3), y = y2, mult = -1)
```

```
## $minimum
## [1] 2.219318
##
## $objective
## [1] 41.79389
```

Looking at the `minimum` element of the list that `optimize()` returns, we see agreement between its estimates of θ and that of Question 4(a).

---

5. Consider a log-Normal model for the wind speeds of Example 5.5, so that $Y \sim \text{log-Normal}(\mu, \sigma^2)$ and hence

$$f_Y(y) = \frac{1}{y\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{[\log(y) - \mu]^2}{2\sigma^2}\right\} \quad y > 0$$

and for $\sigma^2 > 0$. The log-likelihood for an independent sample $\mathbf{y} = (y_1, \ldots, y_n)$ is given by

$$\log f(\mathbf{y} \mid \mu, \sigma^2) = -\frac{n}{2}\log(2\pi) - \frac{n}{2}\log\sigma^2 - \sum_{i=1}^{n}\log(y_i) - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(\log y_i - \mu)^2$$

and its gradient operator is given by

$$\begin{pmatrix} \dfrac{\partial \log f(\mathbf{y} \mid \mu, \sigma^2)}{\partial \mu} \\ \dfrac{\partial \log f(\mathbf{y} \mid \mu, \sigma^2)}{\partial \sigma^2} \end{pmatrix} = \begin{pmatrix} \dfrac{1}{\sigma^2}\sum_{i=1}^{n}(\log y_i - \mu) \\ -\dfrac{n}{2\sigma^2} + \dfrac{1}{2\sigma^4}\sum_{i=1}^{n}(\log y_i - \mu)^2 \end{pmatrix}.$$

(a) For the wind speed data,

$$\sum_{i=1}^{n} \log y_i = -12.755, \quad \sum_{i=1}^{n} (\log y_i)^2 = 155.675$$

and $n = 31$. Write $\log f(\mathbf{y} \mid \mu, \sigma^2)$ in terms of the summary statistics s_1 and s_2, so that $\log f(\mathbf{y} \mid \mu, \sigma^2)$ can be evaluated without knowing $y_1, \ldots, y_n$.

Solution

$$\log f(\mathbf{y} \mid \mu, \sigma^2) = -\frac{n}{2}\log(2\pi) - \frac{n}{2}\log\sigma^2 - s_1 - \frac{1}{2\sigma^2}(s_2 - 2\mu s_1 + n\mu^2).$$

(b) Write a function in R, `ln(pars, s1, s2, n, mult = 1)`, that evaluates $m \log f(\mathbf{y} \mid \mu, \sigma^2)$, where `pars` is a 2-vector such that `pars[1]` $= \mu$ and `pars[2]` $= \sigma^2$, `s1` $= s_1$, `s2` $= s_2$, `n` $= n$ and `mult` $= m$. Your function should ensure that $\sigma^2 > 0$.

Solution

```
mu <- 1
sigsq <- 2
ln <- function(pars, s1, s2, n, mult = 1) {
  # Function to evaluate log-Normal(mu, sig^2) log-likelihood
  # pars is a 2-vector: pars[1] = mu, pars[2] = sig^2
  # s1 and s2 are scalars
  # n is an integer
  # mult is a scalar; defaults to 1
  # returns a scalar
  mu <- pars[1]
  sigsq <- pars[2]
  if (sigsq <= 0)
    return(mult * -1e8)
  out <- -.5 * n * (log(2 * pi) + log(sigsq))
  out <- out - .5 * (s2 - 2 * mu * s1 + n * mu^2) / sigsq
  mult * (out - s1)
}
```

(c) Find

$$\begin{pmatrix} \dfrac{\log f(\mathbf{y} \mid \mu, \sigma^2)}{\partial \mu} \\ \dfrac{\log f(\mathbf{y} \mid \mu, \sigma^2)}{\partial \sigma^2} \end{pmatrix}$$

in terms of s_1 and s_2.

Solution

$$\begin{pmatrix} \dfrac{\partial \log f(\mathbf{y} \mid \mu, \sigma^2)}{\partial \mu} \\ \dfrac{\partial \log f(\mathbf{y} \mid \mu, \sigma^2)}{\partial \sigma^2} \end{pmatrix} = \begin{pmatrix} \dfrac{1}{\sigma^2}(s_1 - n\mu^2) \\ -\dfrac{n}{2\sigma^2} + \dfrac{1}{2\sigma^4}(s_2 - 2\mu s_1 + n\mu^2) \end{pmatrix}.$$

(d) Write a function in R, `ln_d1(pars, s1, s2, n, mult = 1)`, that evaluates the first derivative of $m \log f(\mathbf{y} \mid \mu, \sigma^2)$ w.r.t. (μ, σ^2), where `pars` is a 2-vector such that `pars[1]` $= \mu$ and `pars[2]` $= \sigma^2$, `s1` $= s_1$, `s2` $= s_2$, `n` $= n$ and `mult` $= m$.

Solution

```
ln_d1 <- function(pars, s1, s2, n, mult= 1) {
  # Function to evaluate first derivative of log-Normal(mu, sig^2)
  # log-likelihood w.r.t (\mu, \sigma^2)
  # pars is a 2-vector: pars[1] = mu, pars[2] = sig^2
  # s1 and s2 are scalars
  # n is an integer
  # mult is a scalar; defaults to 1
  # returns a 2-vector
  mu <- pars[1]
  sigsq <- pars[2]
  out <- numeric(2)
  out[1] <- (s1 - n * mu) / sigsq
  out[2] <- -.5 * n / sigsq +
    .5 * (s2 - 2 * mu * s1 + n * mu^2) / (sigsq^2)
  mult * out
}
```

(e) Using values for (μ, σ^2) of $(\mu_0, \sigma_0^2) = (1, 2)$, check your function `ln_d1()` by finite-differencing.

Solution

We'll use function `fd()` from the lecture notes.

```
fd <- function(x, f, delta = 1e-6, ...) {
  # Function to evaluate derivative by finite-differencing
  # x is a p-vector
  # fn is the function for which the derivative is being calculated
  # delta is the finite-differencing step, which defaults to 10^{-6}
  # returns a vector of length x
  f0 <- f(x, ...)
  p <- length(x)
  f1 <- numeric(p)
  for (i in 1:p) {
    x1 <- x
    x1[i] <- x[i] + delta
    f1[i] <- f(x1, ...)
  }
```

```
  (f1 - f0) / delta
}
```

Then we'll load μ_0 and σ_0^2.

```
mu0 <- 1
sigsq0 <- 2
```

Finally we'll evaluate the gradient and its finite-differencing counterpart

```
ln_d1(c(mu0, sigsq0), s1, s2, n)
```

```
## [1] -21.87750  18.77313
```

```
fd(c(mu0, sigsq0), ln, s1 = s1, s2 = s2, n = n)
```

```
## [1] -21.87751  18.77311
```

which are both the same, so it looks as if our function to evaluate the gradient is returning the correct values. (We could check with more values of (μ, σ^2) if we're really keen, but one check is usually sufficient!)

(f) Using (μ_0, σ_0^2) from Question 5(e) as starting values, use `optim()` together with `ln()` and `ln_d1()` to find the maximum likelihood estimates of (μ, σ^2), $(\hat{\mu}, \hat{\sigma}^2)$, via the BFGS method.

Solution

```
s1 <- -12.755
s2 <- 155.675
n <- 31
hats <- optim(c(mu0, sigsq0), ln, ln_d1, s1 = s1, s2 = s2, n = n,
      mult = -1, method = 'BFGS')$par
      hats
```

```
## [1] -0.4114503  4.8524782
```

(g) Evaluate the first derivative of $\log f(\mathbf{y} \mid \mu, \sigma^2)$ at $(\hat{\mu}, \hat{\sigma}^2)$ and comment on whether $(\hat{\mu}, \hat{\sigma}^2)$ are likely to be maximum likelihood estimates.

Solution

The first derivatives at $(\hat{\mu}, \hat{\sigma}^2)$ are given by

```
ln_d1(hats, s1, s2, n)
```

```
## [1] -8.394066e-06  2.346390e-06
```

which are very closely to zero, indicating we're likely to have found the maximum likelihood estimates.

---

6. For the log-Normal wind speed model of Question 5(a), the Hessian matrix of second derivatives is given by

$$\nabla^2 \log f(\mathbf{y} \mid \mu, \sigma^2) = \begin{pmatrix} \dfrac{\partial^2 \log f(\mathbf{y} \mid \mu, \sigma^2)}{\partial \mu^2} & \dfrac{\partial^2 \log f(\mathbf{y} \mid \mu, \sigma^2)}{\partial \mu \partial \sigma^2} \\ \dfrac{\partial^2 \log f(\mathbf{y} \mid \mu, \sigma^2)}{\partial \mu \partial \sigma^2} & \dfrac{\partial^2 \log f(\mathbf{y} \mid \mu, \sigma^2)}{\partial (\sigma^2)^2} \end{pmatrix}$$

where

$$\begin{aligned} \frac{\partial^2 \log f(\mathbf{y} \mid \mu, \sigma^2)}{\partial \mu^2} &= -\frac{n}{\sigma^2} \\ \frac{\partial^2 \log f(\mathbf{y} \mid \mu, \sigma^2)}{\partial \mu \partial \sigma^2} &= -\frac{1}{\sigma^4} \sum_{i=1}^{n} (\log y_i - \mu) \\ \frac{\partial^2 \log f(\mathbf{y} \mid \mu, \sigma^2)}{\partial (\sigma^2)^2} &= \frac{n}{2\sigma^4} - \frac{1}{\sigma^6} \sum_{i=1}^{n} (\log y_i - \mu)^2. \end{aligned}$$

(a) Write a function, `nl_d2(pars, s1, s2, n, mult = -1)`, that evaluates the Hessian matrix in terms of s_1 and s_2 given in Question 5(a) and assuming that same arguments as `nl_d1()` of Question 5(d).

Solution

```
ln_d2 <- function(pars, s1, s2, n, mult= 1) {
  # Function to evaluate second derivatives of log-Normal(mu, sig^2)
  # log-likelihood w.r.t (\mu, \sigma^2)
  # pars is a 2-vector: pars[1] = mu, pars[2] = sig^2
  # s1 and s2 are scalars
  # n is an integer
  # mult is a scalar; defaults to 1
  # returns a 2x2 matrix
  mu <- pars[1]
  sigsq <- pars[2]
  out <- matrix(0, 2, 2)
  out[1, 1] <- - n / sigsq
  out[1, 2] <- out[2, 1] <- -(s1 - n * mu) / (sigsq^2)
  out[2, 2] <- .5 * n / sigsq^2 - (s2 - 2 * mu * s1 + n * mu^2) / (sigsq^3)
  mult * out
}
```

(b) Using `nl()` and `nl_d1()` from Question 5(a) and `nl_d2()` from Question 6(a), find $(\hat{\mu}_2, \hat{\sigma}_2^2)$, the maximum likelihood estimates, using Newton's method via `nlminb()` in R.

Solution

```
hats2 <- nlminb(c(mu0, sigsq0), ln, ln_d1, ln_d2, s1 = s1, s2 = s2, n = n,
        mult = -1)$par
hats2
```

```
## [1] -0.4114516  4.8524818
```

(c) Evaluate the first derivative of $\log f(\mathbf{y} \mid \mu, \sigma^2)$ at $(\hat{\mu}_2, \hat{\sigma}_2^2)$ from Question 6(b) and comment on whether $(\hat{\mu}_2, \hat{\sigma}_2^2)$ are likely to be maximum likelihood estimates.

Solution

We can proceed as in question 5g.

```
ln_d1(hats2, s1, s2, n)
```

```
## [1] -1.098215e-15  5.506706e-14
```

which are still very closely to zero, indicating we're likely to have found the maximum likelihood estimates.

(d) Evaluate the Hessian matrix of $\log f(\mathbf{y} \mid \mu, \sigma^2)$ at $(\hat{\mu}_2, \hat{\sigma}_2^2)$ from Question 6(b) and comment on whether $(\hat{\mu}_2, \hat{\sigma}_2^2)$ are likely to be maximum likelihood estimates.

Solution

We'll calculate the Hessian

```
H <- ln_d2(hats2, s1, s2, n)
```

and then, after it's been negated, want all of its eigenvalues to be positive

```
ev <- eigen(-H, symmetric = TRUE, only.value = TRUE)
all(ev$values > 0)
```

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

which they are.

---

7. Recall the wind speed data of Example 5.5. Consider these as independent realisations from the Gamma(α, β) distribution, i.e. with pdf

$$f_Y(y \mid \alpha, \beta) = \frac{y^{\alpha-1} e^{-\beta x} \beta^\alpha}{\Gamma(\alpha)}, \quad y > 0,$$

for parameters $\alpha, \beta > 0$.

(a) Find the maximum likelihood estimates of (α, β) using the BFGS method with `optim()` in `R` by supplying a function that evaluates the negative log-likelihood's gradient vector w.r.t. (α, β). [Note that in `R` `lgamma(x)` evaluates $\log \Gamma(x)$ and `digamma(x)` evaluates $\mathrm{d} \log \Gamma(x)/\mathrm{d}x$ for `x` $= x$, where `digamma()` relies on the so-called polygamma function.]

Solution

We'll load the wind speed data as `y0`.

```
y0 <- c(3.52, 1.95, 0.62, 0.02, 5.13, 0.02, 0.01, 0.34, 0.43, 15.5,
  4.99, 6.01, 0.28, 1.83, 0.14, 0.97, 0.22, 0.02, 1.87, 0.13, 0.01,
  4.81, 0.37, 8.61, 3.48, 1.81, 37.21, 1.85, 0.04, 2.32, 1.06)
```

Then the following functions evaluate the negative log-likelihood and its gradient, respectively.

```
nldgamma <- function(pars, y) {
  # Function to evaluate Gamma(alpha, beta) negative log-likelihood
  # pars is a 2-vector
  # y is a vector
  # returns a scalar
  alpha <- pars[1]
  beta <- pars[2]
  if (min(c(alpha, beta)) <= 0)
    return(1e8)
  n <- length(y)
  - n * alpha * log(beta) + n * lgamma(alpha) - (alpha - 1) * sum(log(y)) +
    beta * sum(y)
}

nldgamma_d1 <- function(pars, y) {
  # Function to evaluate first derivative of Gamma(alpha, beta)
  # negative log-likelihood w.r.t. (alpha, beta)
  # pars is a 2-vector
  # y is a vector
  # returns a 2-vector
  alpha <- pars[1]
  beta <- pars[2]
  n <- length(y)
  out <- numeric(2)
  out[1] <- - n * log(beta) + n * digamma(alpha) - sum(log(y))
  out[2] <- - n * alpha / beta + sum(y)
  out
}
```

We'll choose starting values for (α, β) as $(\alpha_0, \beta_0) = (1, 1)$ and store these as `pars0`

```
pars0 <- c(1, 1)
```

and then call `optim()` to implement the BFGS method

```
pars_bfgs <- optim(pars0, nldgamma, nldgamma_d1, y = y0, method = 'BFGS')$par
pars_bfgs
```

```
## [1] 0.4017348 0.1179670
```

storing the results maximum likelihood estimates as `pars_bfgs`.

(b) Find the maximum likelihood estimates of (α, β) using Newton's method with `nlminb()` in `R` by supplying the function that evaluates the negative log-likelihood's gradient vector w.r.t. (α, β) created in Question `ref qgbfgs`, and by supplying a function that evaluates the negative log-likelihood's Hessian matrix w.r.t. (α, β).

[Note that in R `trigamma(x)` evaluates $\mathrm{d}^2 \log \Gamma(x)/\mathrm{d}x^2$ for `x` $= x$.]

Solution

We can re-use a few functions and objects from above, so next we'll write a function to evaluate the second derivative of the log-likelihood w.r.t. (α, β).

```
nldgamma_d2 <- function(pars, y) {
  # Function to evaluate second derivative of Gamma(alpha, beta)
  # negative log-likelihood w.r.t. (alpha, beta)
  # pars is a 2-vector
  # y is a vector
  # returns a 2x2 matrix
  alpha <- pars[1]
  beta <- pars[2]
  n <- length(y)
  out <- matrix(0, 2, 2)
  out[1, 1] <- n * trigamma(alpha)
  out[1, 2] <- out[2, 1] <- - n / beta
  out[2, 2] <- n * beta^2
  out
}
```

We supply this to `nlminb()` with `pars0` from above

```
pars_newt <- nlminb(pars0, nldgamma, nldgamma_d1, nldgamma_d2, y = y0)$par
pars_newt
```

```
## [1] 0.4017348 0.1179670
```

and store the maximum likelihood estimates as `pars_newt`.

(c) By considering the gradient at your estimates in Questions 7(a) and 7(b), verify that on both occasions the maximum likelihood estimates have been reached.

Solution

We'll evaluate the gradient of the negative log-likelihood at `pars_bfgs` and `pars_newt`

```
nldgamma_d1(pars_bfgs, y0)
```

```
## [1] -1.200319e-06  8.312922e-06
```

```
nldgamma_d1(pars_newt, y0)
```

```
## [1]  1.999826e-06 -1.875220e-06
```

and find both are sufficiently close to the zero vector that we think we've reached the maximum likelihood estimates.

---

8. Use the Nelder-Mead method to find the maximum likelihood estimates of (μ, σ^2) for the log-Normal model, starting values and data of Question 5.

Solution

```
hats3 <- optim(c(mu0, sigsq0), ln, s1 = s1, s2 = s2, n = n, mult = -1)$par
hats3
```

```
## [1] -0.4113588  4.8537235
```

As a quick aside we'll have a look at the derivative of our estimates

```
ln_d1(hats3, s1, s2, n)
```

```
## [1] -0.0005929934 -0.0008170016
```

which we see are still close to zero, but not as close as with the gradient-based optimisation methods. This is something to expect with the Nelder-Mead method because it doesn't use gradients in its iterations, and doesn't use gradient-based criteria to determine convergence, whereas the gradient-based methods do (amongst other criteria).

---

9. Find the maximum likelihood estimates for the wind speed data of Example 5.5 based on the Gamma(α, β) model of Question 7 using the Nelder-Mead method.

Solution

We'll use the following call to `optim()`

```
pars_nm <- optim(pars0, nldgamma, y = y0)$par
pars_nm
```

```
## [1] 0.4017200 0.1179219
```

and see that we get similar estimates to question 7.

---

10. Use simulated annealing with $N = 1000$ iterations (i.e. `control = list(maxit = 1e3)`) to approximate the maximum likelihood estimates of (μ, σ^2) for the wind speed data and log-Normal model of Question 5.

Solution

```
hats4 <- optim(c(mu0, sigsq0), ln, s1 = s1, s2 = s2, n = n, mult = -1,
           method = 'SANN', control = list(maxit = 1e3))
hats4
```

```
## $par
## [1] -0.394841  4.795629
##
## $value
## [1] 55.71617
##
## $counts
## function gradient
##     1000       NA
```

```
##
## $convergence
## [1] 0
##
## $message
## NULL
```

---

11. Find the maximum likelihood estimates for the wind speed data of Example 5.5 based on the Gamma(α, β) model of Question 7 using simulated annealing with $N = 10^4$ iterations, and then evaluate the gradient w.r.t. (α, β) at the maximum likelihood estimates using your gradient function from Question 7(a).

Solution

We'll use the following call to `optim()`

```
pars_sann <- optim(pars0, nldgamma, y = y0, method = 'SANN')$par
pars_sann
```

```
## [1] 0.3995571 0.1175635
```

which recognises that $N = 10^4$ iterations is `optim()`'s default. We get similar estimates to questions 7 and 9. The following evaluates the gradient

```
nldgamma_d1(pars_sann, y0)
```

```
## [1] -0.3835148  0.2118473
```

which we see is near zero, but nowhere near as near as for estimates we've seen previously. The nature of simulated annealing means we're only going to get a final gradient incredibly close to zero by chance, or if we use *lots* of iterations *and* allow the temperature to decrease sufficiently. In practice, we'd probably not want to wait this long.
