Solutions Lecture 11

Exercise 1

Question 1.1

In order to rewrite the `for` loop with `replicate( )`, we first of all need to create a function that performs a single replication of our MC experiment:

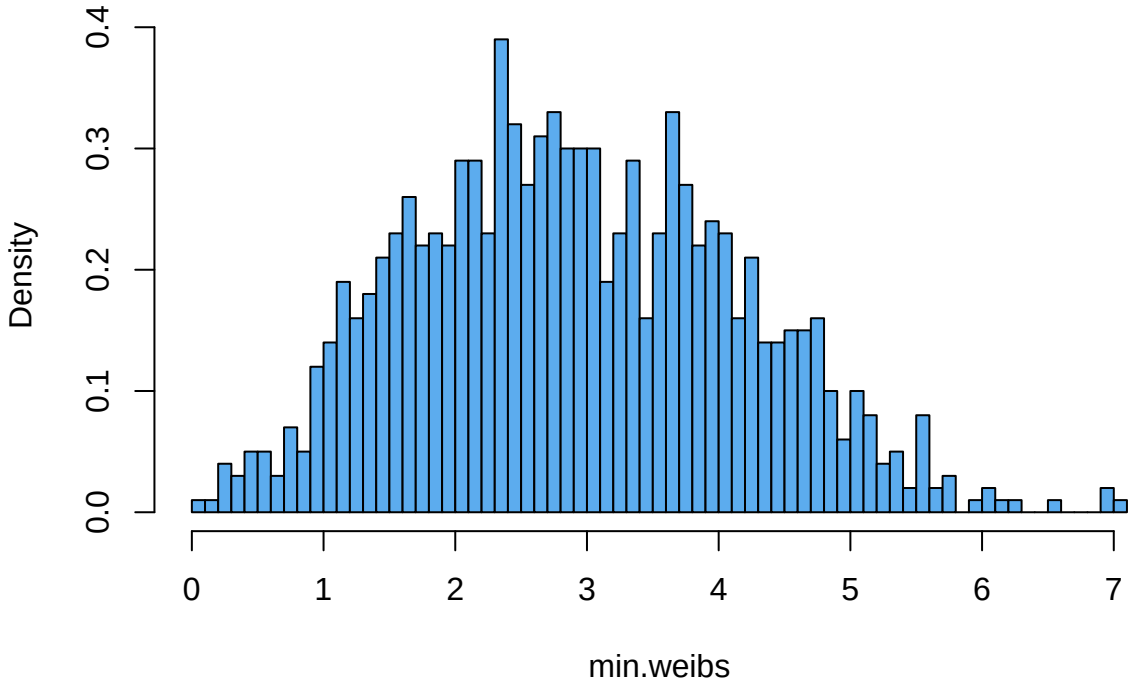
```
MC_experiment =  function(weib.shape, weib.scale){
  x1 = rweibull(1, shape = weib.shape[1], scale = weib.scale[1])
  x2 = rweibull(1, shape = weib.shape[2], scale = weib.scale[2])
  min.weibs = min(x1, x2)
  return(min.weibs)
}
```

Once we have done this, we can rewrite the MC experiment using `replicate( )`:

```
set.seed(13)
n.repl = 1000
weib.shape = c(2, 3)
weib.scale = c(5, 4)
min.weibs = replicate(n.repl, MC_experiment(weib.shape, weib.scale))
```

We can now check whether the experiment yields the same result after the rewriting by drawing a histogram and comparing it to the one given in the text of the exercise:

```
hist(min.weibs, 50, prob = T, main = '', col = 'steelblue2')
```



We can observe that the histogram is indeed identical to the one given. Note that this is possible because we have set the same random seed through `set.seed( )`.

Question 1.2

We can use `benchmark( )` to compare the execution time of the two approaches. We can set the number of replications = 100:

```
library(rbenchmark)
t.eval = benchmark(
  'replicate' = {
    min.weibs1 = replicate(n.repl, MC_experiment(weib.shape, weib.scale))
  },
  'for-loop' = {
    min.weibs2 = rep(NA, n.repl)
    for(i in 1:n.repl){
      x1 = rweibull(1, shape = weib.shape[1],
                    scale = weib.scale[1])
      x2 = rweibull(1, shape = weib.shape[2],
                    scale = weib.scale[2])
      min.weibs2[i] = min(x1, x2)
    }
  },
  replications = 100
)
```

```
t.eval
```

```
##        test replications elapsed relative user.self sys.self user.child
## 2  for-loop          100    1.96    1.202      1.70     0.19         NA
## 1 replicate          100    1.63    1.000      1.25     0.23         NA
##   sys.child
## 2        NA
## 1        NA
```

We can observe that the `replicate( )` implementation is somewhat faster than the `for` loop one.

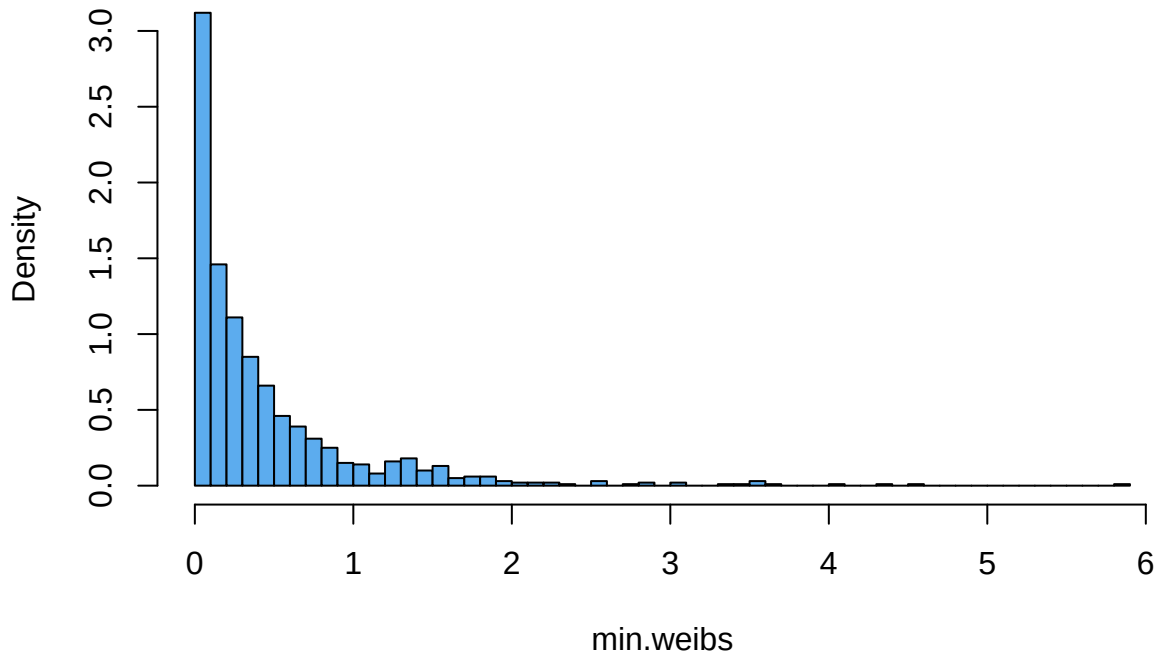
Question 1.3a

All we need to do here is to change the values of the shape and scale parameters:

```
set.seed(13)
weib.shape2 = c(0.7, 1)
weib.scale2 = c(0.5, 2)

min.weibs = replicate(
  n.repl, MC_experiment(weib.shape2, weib.scale2))

hist(min.weibs, 50, prob = T, main = '', col = 'steelblue2')
```

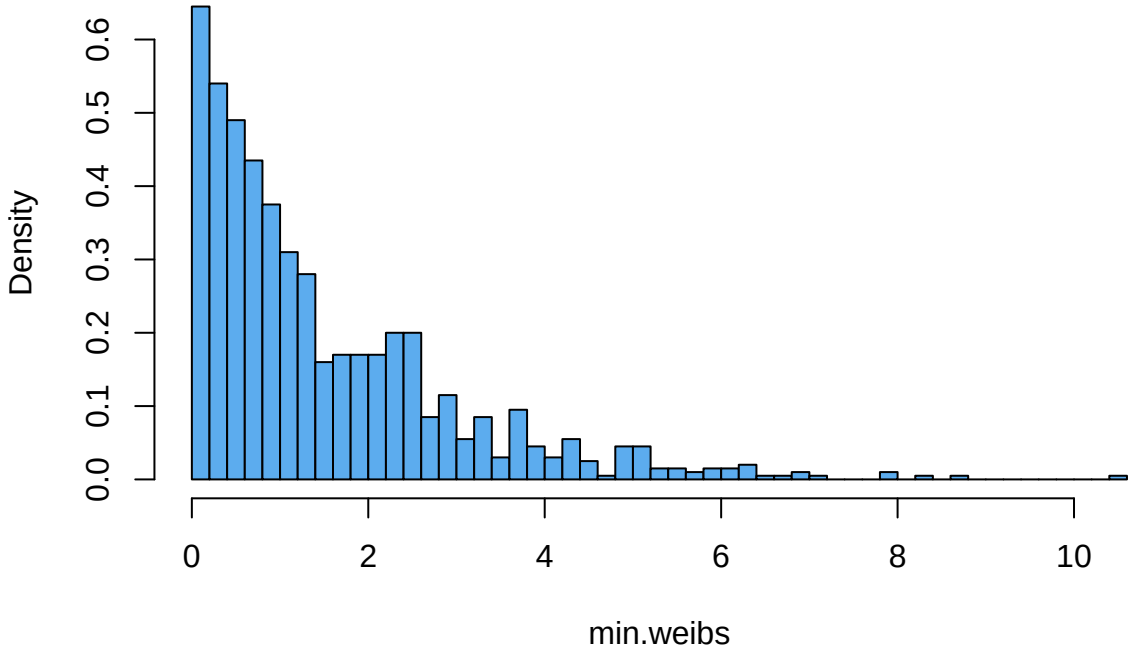


Question 1.3b

```
set.seed(13)
weib.shape3 = c(1, 1)
weib.scale3 = c(3, 3)

min.weibs = replicate(
  n.repl, MC_experiment(weib.shape3, weib.scale3))

hist(min.weibs, 50, prob = T, main = '', col = 'steelblue2')
```



Exercise 2

Question 2.1

Note that this problem can be solved by adapting the negative log-likelihood function used in the slides. We can do this by letting the input parameter `theta` include both the means μ_k and the standard deviations σ_k. Because the standard deviation cannot be negative, here we apply a log transformation to the input so $\tau_k = \log(\sigma_k)$ and, conversely, $\sigma_k = \exp(\tau_k)$. By doing so we can apply the `Nelder-Mead` algorithm in the maximization phase as we did in Lecture 10. Note that alternatively we could use the `L-BFGS-B` as a method in `optim( )`, and proceed with the untransformed σ_ks.

Additionally, also the function `f.x` needs to be rewritten to incorporate the variance parameters:

```
neg.logl = function(theta, pi1, w1, x) {
  mu1 = theta[1]
  mu2 = theta[2]
  sigma1 = exp(theta[3]) # transformation!
  sigma2 = exp(theta[4]) # transformation!
  # density of the mixture model:
  f.x1 = pi1 * dnorm(x, mu1, sd = sigma1)
  f.x2 = (1-pi1) * dnorm(x, mu2, sd = sigma2)
  # negative log-likelihood:
  - sum(w1 * log(f.x1) + (1 - w1) * log(f.x2))
  }
```

Questions 2.2 & 2.3

The code below is taken from the question, and used to generate our data:

```
set.seed(13)
n = 2000; pi1 = 0.35
mu1 = 0.8; mu2 = 2.5
sigma1 = 0.8; sigma2 = 0.6
group = sample(1:2, n, replace = T, prob = c(pi1, 1-pi1))
table(group)
```

```
## group
##    1    2
##  698 1302
```

```
x = rep(NA, n)
x[group == 1] = rnorm(sum(group == 1), mu1, sd = sigma1)
x[group == 2] = rnorm(sum(group == 2), mu2, sd = sigma2)
```

We can now proceed to implement the EM algorithm. Let's first of all initialize two matrices, one where we will store the probability that observations come from the first component at each iteration, `p1hat`, and another where we will save the corresponding parameter values, `theta_hat`. Let's set 200 as number of iterations (we will check convergence later:

```
n.iter = 200
# pi1hat: vector where 1 value = 1 iteration of the EM algorithm
```

```
pi1hat = rep(NA, n.iter) # estimates of \pi_1
# p1hat and theta_hat: matrices where 1 row = 1 iteration of the EM
p1hat = matrix(NA, n.iter, n) # estimates of Pr(Z_i = 1)
theta_hat = matrix(NA, n.iter, 4) # We add sigma1 and sigma2, so 4 columns
```

Next, we need to provide a starting point for the algorithm - specifically, the probability that each observation comes from the first (and the second) component:

```
pi1hat[1] = 0.45

p1hat[1, ] = runif(n, pi1hat[1] - 0.35, pi1hat[1] + 0.35)
```

We are now ready to run the first M step:

```
theta_hat[1, ] = optim(c(-0.5, 0.5, -0.2, 0.2), function(theta)
  neg.logl(theta, pi1hat[1], p1hat[1,], x))$par
```

Now we can alternate E and M steps for the desired number of iterations:

```
for (t in 2:n.iter) {

  # E step: update individual probability memberships

  # Note the use of exp(theta_hat[t-1, 3]) and exp(theta_hat[t-1, 4]) to represent sigma

  p.temp = cbind(
    pi1hat[t-1] * dnorm(x, theta_hat[t-1, 1], exp(theta_hat[t-1, 3])),
    (1 - pi1hat[t-1]) * dnorm(x, theta_hat[t-1, 2], exp(theta_hat[t-1, 4])))

  p1hat[t, ] = p.temp[ , 1]/rowSums(p.temp)


  # M step: update parameter estimates

  pi1hat[t] = mean(p1hat[t, ])

  theta_hat[t, ] = optim(theta_hat[t-1, ], function(theta)
    neg.logl(theta, pi1hat[t], p1hat[t,], x))$par

  }
```

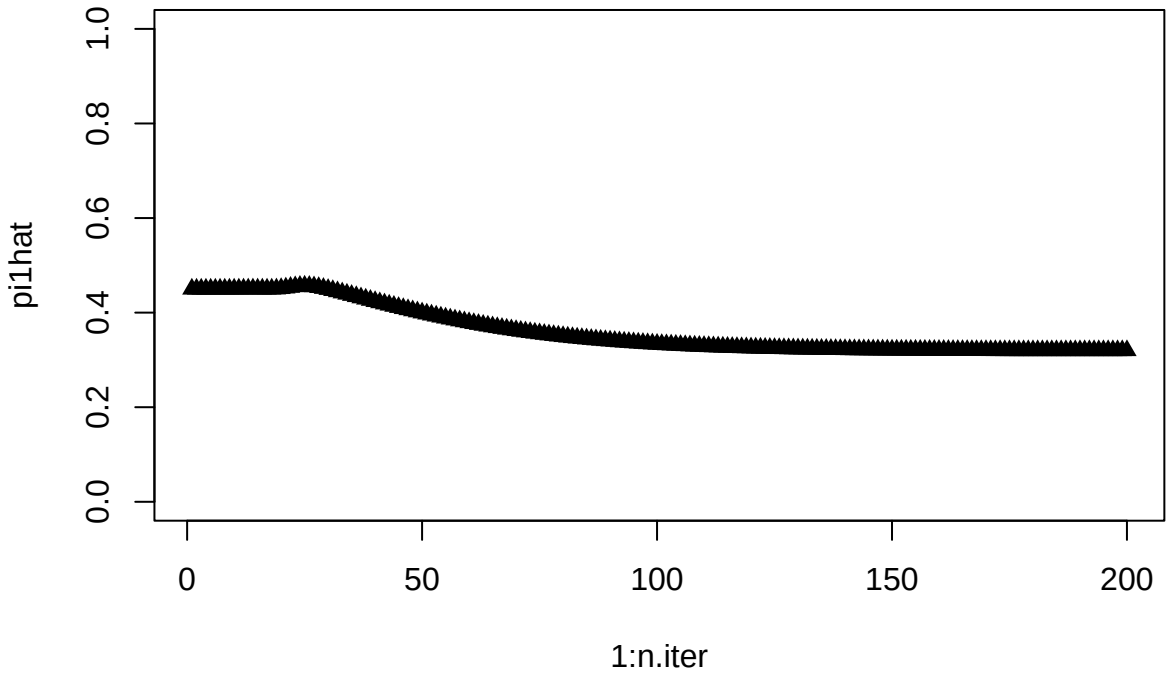
Before we have a look at the parameter estimates, let's check whether the algorithm seems to have converged:

```
tail(theta_hat, 20)
```

```
##             [,1]     [,2]       [,3]       [,4]
## [181,] 0.6929737 2.484827 -0.3291722 -0.5175389
## [182,] 0.6929737 2.484827 -0.3291722 -0.5175389
## [183,] 0.6929737 2.484827 -0.3291722 -0.5175389
## [184,] 0.6929737 2.484827 -0.3291722 -0.5175389
## [185,] 0.6929737 2.484827 -0.3291722 -0.5175389
```

```
## [186,] 0.6929737 2.484827 -0.3291722 -0.5175389
## [187,] 0.6929737 2.484827 -0.3291722 -0.5175389
## [188,] 0.6929737 2.484827 -0.3291722 -0.5175389
## [189,] 0.6929737 2.484827 -0.3291722 -0.5175389
## [190,] 0.6929737 2.484827 -0.3291722 -0.5175389
## [191,] 0.6929737 2.484827 -0.3291722 -0.5175389
## [192,] 0.6929737 2.484827 -0.3291722 -0.5175389
## [193,] 0.6929737 2.484827 -0.3291722 -0.5175389
## [194,] 0.6929737 2.484827 -0.3291722 -0.5175389
## [195,] 0.6929737 2.484827 -0.3291722 -0.5175389
## [196,] 0.6929737 2.484827 -0.3291722 -0.5175389
## [197,] 0.6929737 2.484827 -0.3291722 -0.5175389
## [198,] 0.6929737 2.484827 -0.3291722 -0.5175389
## [199,] 0.6929737 2.484827 -0.3291722 -0.5175389
## [200,] 0.6929737 2.484827 -0.3291722 -0.5175389
```

```r
plot(1:n.iter, pi1hat, pch = 17, ylim = c(0, 1))
```



We can observe that after 200 iterations, the algorithm is not moving much anymore and has thus converged. The ML estimates from this EM run are:

```r
c('mu1' = theta_hat[n.iter, 1],
  'mu2' = theta_hat[n.iter, 2],
  'sigma1' = exp(theta_hat[n.iter, 3]),
  'sigma2' = exp(theta_hat[n.iter, 4]),
```

```
  'pi1' = pi1hat[n.iter],
  'pi2' = 1 - pi1hat[n.iter])
```

```
##       mu1       mu2    sigma1    sigma2       pi1       pi2
## 0.6929737 2.4848272 0.7195191 0.5959855 0.3198633 0.6801367
```

The final membership probabilities are stored in `p1hat`, which contains the estimated probability that each of the 2000 observations comes from the first estimated component of the mixture. Below we display the first 30 values:

```
p1hat[n.iter, ][1:30]
```

```
##  [1] 0.995779128 0.045902506 0.008435082 0.034168521 0.264422324 0.004167770
##  [7] 0.231283357 0.187690900 0.962408461 0.007343170 0.830537108 0.999745571
## [13] 0.979611789 0.271576051 0.002719070 0.003886254 0.153653915 0.021035660
## [19] 0.994600768 0.869359614 0.007250351 0.002969545 0.080694842 0.004078211
## [25] 0.002886803 0.002363655 0.013775203 0.112610819 0.031498855 0.001186620
```

Question 2.4

For simplicity, let's create a function that performs a single run of the EM. We will reuse the code we used before, adjusting it to a function:

```
EM_function = function(x, n.iter = 200) {

  # 1. Preallocate objects:
  # These are the same objects we used before.

  pi1hat = rep(NA, n.iter) # estimates of \pi_1
  p1hat = matrix(NA, n.iter, n) # estimates of Pr(Z_i = 1)
  theta_hat = matrix(NA, n.iter, 4) # We add sigma1 and sigma2, so 4 columns
  n = length(x)

  # 2. Determine starting point of EM:
  # Again, same objects.

  pi1hat[1] = 0.45
  p1hat[1, ] = runif(n, pi1hat[1] - 0.35, pi1hat[1] + 0.35)

  theta_hat[1, ] = optim(c(-0.5, 0.5, -0.2, 0.2), function(theta)
    neg.logl(theta, pi1hat[1], p1hat[1,], x))$par

  # 3. Run the EM algorithm

  for (t in 2:n.iter) {

  # E step: update individual probability memberships

  p.temp = cbind(
    pi1hat[t-1] * dnorm(x, theta_hat[t-1, 1], exp(theta_hat[t-1, 3])),
    (1 - pi1hat[t-1]) * dnorm(x, theta_hat[t-1, 2], exp(theta_hat[t-1, 4])))
```

```
  p1hat[t, ] = p.temp[ , 1]/rowSums(p.temp)

  # M step: update parameter estimates

  pi1hat[t] = mean(p1hat[t, ])

  theta_hat[t, ] = optim(theta_hat[t-1, ], function(theta)
    neg.logl(theta, pi1hat[t], p1hat[t,], x))$par

  }

  # Save last log likelihood:

  logl.last = -neg.logl(theta_hat[n.iter, ], pi1hat[n.iter],
                        p1hat[n.iter,], x)

  # 4: Define the exports

  out = list('logl' = logl.last,
             'theta_hat' = c(theta_hat[n.iter, 1],
                             theta_hat[n.iter, 2],
                             exp(theta_hat[n.iter, 3]),
                             exp(theta_hat[n.iter, 4])),
             'pi' = c(pi1hat[n.iter],
                      1 - pi1hat[n.iter]),
             'p1_last' = p1hat[n.iter, ])

  return(out)

}
```

We can use `replicate( )` to fit the EM from 10 different starting points:

```
set.seed(13)
my_results = replicate(10, EM_function(x, n.iter = 200),
                       simplify = F)
```

We can now compare solutions, and pick the one with the highest loglikelihood:

```
logl.vals = sapply(my_results, function(x) x$logl)
logl.vals
```

```
##  [1] -3177.085 -3176.730 -3176.550 -3176.841 -3176.664 -3177.117 -3176.660
##  [8] -3176.701 -3176.508 -3176.736
```

```
which(logl.vals == max(logl.vals))
```

```
## [1] 9
```

Our final estimates are:

```
EM_estimates = my_results[[which(logl.vals == max(logl.vals))]]
EM_estimates$logl
```

```
## [1] -3176.508
```

```
EM_estimates$theta_hat
```

```
## [1] 0.6923311 2.4846158 0.7192253 0.5961318
```

```
EM_estimates$pi
```

```
## [1] 0.3196752 0.6803248
```

Question 2.5

The estimated means and standard deviations are:

```
EM_estimates$theta_hat
```

```
## [1] 0.6923311 2.4846158 0.7192253 0.5961318
```

```
EM_estimates$pi
```

```
## [1] 0.3196752 0.6803248
```

The true parameter values are:

```
c(mu1, mu2, sigma1, sigma2)
```

```
## [1] 0.8 2.5 0.8 0.6
```

```
c(pi1, 1-pi1)
```

```
## [1] 0.35 0.65
```

We can see that estimated component 1 roughly matches the true component 1 from which we simulated the data, and that the estimated component 2 roughly matches the true component 2. So, in this case there is no label switching.

As concerns the mixing proportions, our estimate is that about 32% of the observed data come from the first component - not too far from the true parameter value $\pi_1 = 0.35$.

Question 2.6

Based on the observation that inferred component 1 corresponds to the true first component, we can compute the predicted group memberships using:

```
predicted.group = ifelse(EM_estimates$p1_last > 0.5, 1, 2)
```

The comparison of real vs predicted group yields

```
table(predicted.group, group)
```

```
##                group
## predicted.group    1    2
##               1  551   50
##               2  147 1252
```

meaning that 1252 + 551 observations have been correctly classified, and the remaining ones are wrongly classified. The misclassification rate is thus equal to

```
(147 + 50)/n
```

```
## [1] 0.0985
```