TMA4300 Project 3

Martin Ludvigsen & Sivert Selnes

April 9, 2019

A.1

The bootstrap method in genereal relies on independently and identically distributed samples to resample (if they were correlated they would of course not show the same properties after random reordering). For data that origins from time series or regression problems that is an issue, since the very core of the time series and regression is to predict the response given the covariate(s), hence some correlation must take place. That implies, nesseccerily, some relationship between the responses of similar covariates that are close in time or space.

If the chosen model is the appropriate one, however, it should capture all dependencies in the model and the deviations from the models are (ideally) uncorrelated white noise, not useful for predictions. Hence, these residuals can be bootstrapped to get an estimate of the variability and bias of the model parameters. In this particular exercise the carrying assumption is that our data origins from an autoregressive process of second order. The residual bootstrap method then goes as follows:

- Estimate the AR(2) coefficients α_1 and α_2.
- Define the innovations $\hat{e}_t = X_t - \hat{\alpha}_1 X_{t-1} - \hat{\alpha}_2 X_{t-2}$.
- Ensure that the residuals have zero mean, $\hat{\epsilon}_t = \hat{e}_t - \bar{e}$.
- Resample $n+1$ values from the (iid) residuals $\{\hat{\epsilon}_2, ..., \hat{\epsilon}_n\}$.
- Reconstruct a pseudo data series with the resampled residuals, $X_0^* = \epsilon_0^*, \quad X_t^* = \hat{\alpha}_1 X_{t-1} + \hat{\alpha}_2 X_{t-2} + \epsilon_t^*$.
- Estimate new coefficients α_1 and α_2 for each pseudo data series to get an idea of the parameter variance and bias (which can be calculated by the plug-in estimator.)

```
# Bootstrapping the AR(2)-model

ts = data3A$x
n = length(ts)

beta_estimate = ARp.beta.est(ts,2) # get AR(2) parameters estimates
res_LS = ARp.resid(ts, beta_estimate$LS) # LS residuals
res_LA = ARp.resid(ts, beta_estimate$LA) # LA residuals

#mean(res_LS) # mean is zero - OK
#mean(res_LA) #

B = 1500

LS <- function(res, beta){
  e_sample = sample(res, length(res), replace=T) # resample residuals
  init = sample(1:(n-1), 1, replace = T) # random intializing index
  x0 = ts[init:(init+1)]
  x_pseudo = ARp.filter(x0, beta, e_sample) # generate pseudo time series
  beta_boot = ARp.beta.est(x_pseudo,2)$LS # get coefficients
  return(beta_boot)
}
```

```r
LA <- function(res, beta){
  e_sample = sample(res, length(res), replace=T) # resample
  init = sample(1:(n-1), 1, replace = T) # random intializing index
  x0 = ts[init:(init+1)]
  x_pseudo = ARp.filter(x0, beta, e_sample) # generate pseudo time series
  beta_boot = ARp.beta.est(x_pseudo,2)$LA # get coefficient
  return(beta_boot)
}

boot_LS = replicate(B,LS(res_LS, beta_estimate$LS))
boot_LA = replicate(B,LA(res_LA, beta_estimate$LA))
```

```r
# Bias estimates

mean(boot_LS[1,])-beta_estimate$LS[1] # bias of beta1, LS-method
## [1] -0.01629063
mean(boot_LS[2,])-beta_estimate$LS[2] # bias of beta2, LS-method
## [1] 0.01047407
mean(boot_LA[1,])-beta_estimate$LA[1] # bias of beta1, LA-method
## [1] -0.003281431
mean(boot_LA[2,])-beta_estimate$LA[2] # bias for beta2, LA-method
## [1] 0.002704513

# Variances estimates

sum((boot_LS[1,]-beta_estimate$LS[1])^2)/B # variance of beta1, LS-method
## [1] 0.005598091
sum((boot_LS[2,]-beta_estimate$LS[2])^2)/B # variance of beta2, LS-method
## [1] 0.005281759
sum((boot_LA[1,]-beta_estimate$LA[1])^2)/B # variance of beta1, LA-method
## [1] 0.0004155088
sum((boot_LA[2,]-beta_estimate$LA[2])^2)/B # variance for beta2, LA-method
## [1] 0.000404486
```

The bootstrap bias and bootstrap variance are easiliy computed by subtracting the β-estimate from the bootstrap mean, and taking the sum of squares for the variance, respectively. **Both variance and bias are considerably lower for the estimators computed by least absolute residuals than for the least squared residuals.** Thus, for this problem the absolute residuals method give the better estimates.

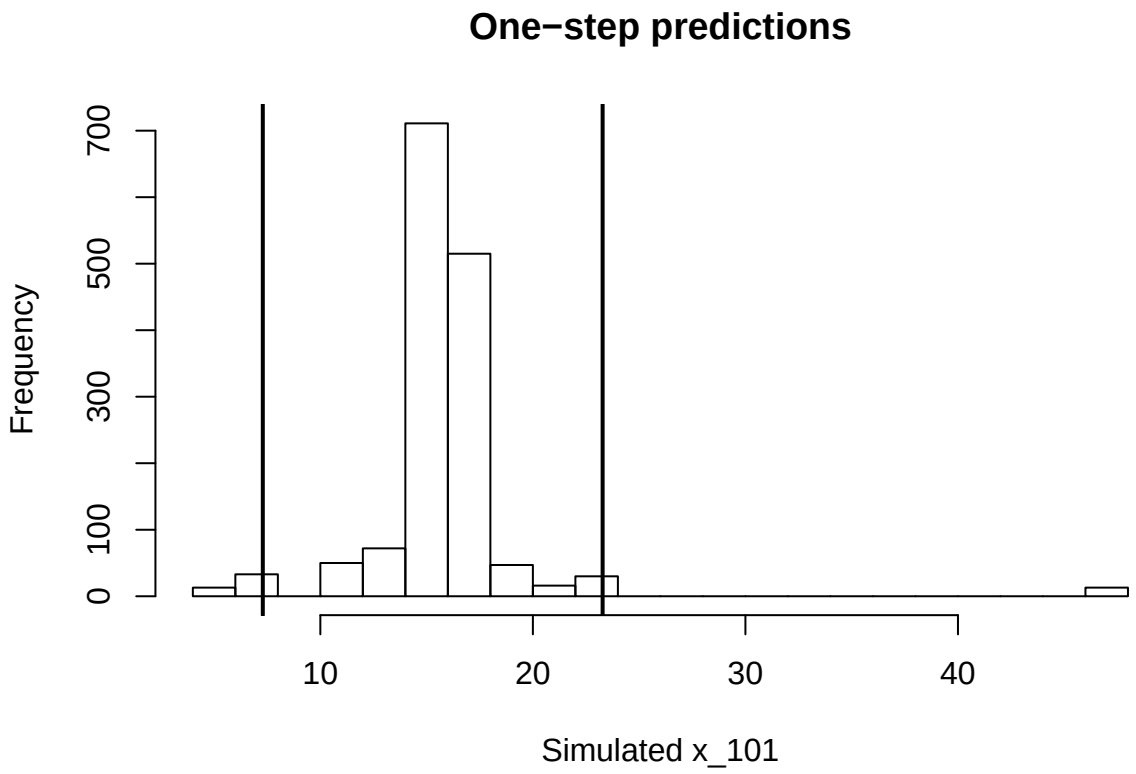

Figure 1: Histogram of bootstrapped one-step predictions and the 0.025 and 0.975 quantiles.

A.2

We predict the next value in the time series, x_{101}, by using every pair of bootstrapped parameter values and a randomly chosen residual to generate the innovations that yields the empirical distribution. After the bootstrapping of the parameters, the distribution contains B predictions of x_{101}. To construct an estimated 95 % confidence interval we choose the $\alpha_{2.5}$ and the $\alpha_{97.5}$-quantiles in the empirical distribution.

```
pred_new <- function(boot, residuals){
  return(boot[1,]*ts[100] + boot[2,]*ts[99] + sample(residuals, B, replace=T))
}

x101_LA = pred_new(boot_LA,res_LA)
x101_LS = pred_new(boot_LS,res_LS)
quantile(x101_LA, c(0.025,0.975))
```

```
##      2.5%     97.5%
##  7.291044 23.280584
```

```
quantile(x101_LS, c(0.025,0.975))
```

```
##      2.5%     97.5%
##  7.218947 23.132199
```

```
hist(x101_LA, 30, xlab = "Simulated x_101", ylab = "Frequency",  main = "One-step predictions")
abline(v = quantile(x101_LA, c(0.025,0.975))
, col=1, lwd=2)
```

```
hist(x101_LS, 30, xlab = "Simulated x_101", ylab = "Frequency",  main = "One-step predictions")
abline(v = quantile(x101_LS, c(0.025,0.975)),col=1, lwd=2)
```

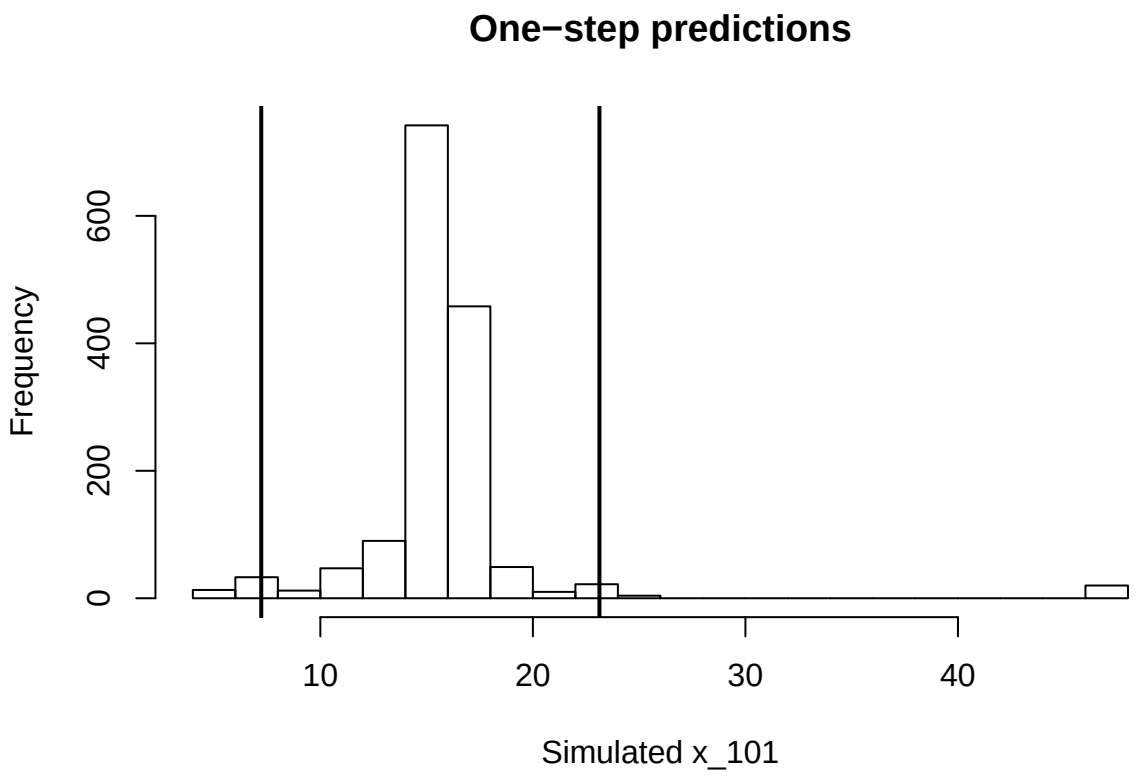

Figure 2: Histogram of bootstrapped one-step predictions and the 0.025 and 0.975 quantiles.

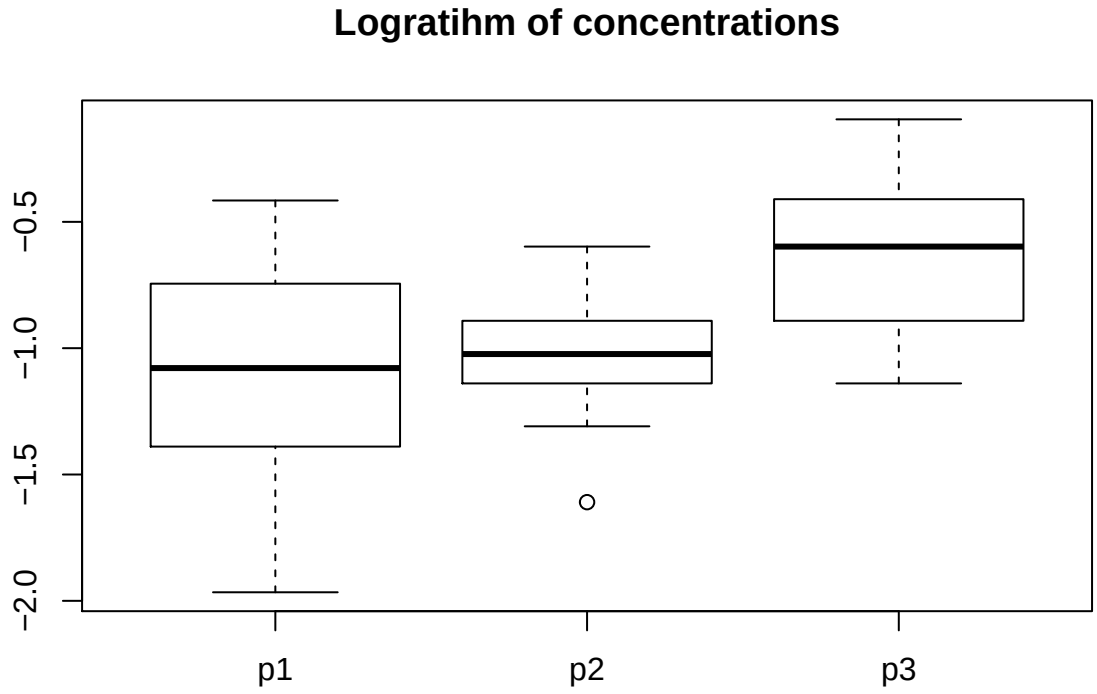

Figure 3: Box plot of bilirubin data. At first glance, it looks like person 3 has a higher bilirubin concentration than the two others.

B.1, B.2, B.3

In this task we have labeled measured data of three different individuals. A box plot of the data is shown in figure 3.

We want to fit a linear model

$$\log Y_{ij} = \beta_i + \varepsilon_{ij}$$

, and test

$$H_0 : \beta_0 = \beta_1 = \beta_2, \quad H_1 : \text{At least one } \beta_i \text{ different from the others.} \tag{1}$$

Under the null hypothesis, the data should be iid, meaning that labels should not matter. This is the idea of a permutation test. If the null hypothesis is correct, changing labels should not matter.

In the permutation test, we create a permuted dataset $\mathbf{x}^*$, from our original dataset $\mathbf{x}$, and calculate our test statistic $t(\mathbf{x}^*)$. We do this B times, and the p-value can then be approximated by $\frac{1}{B}$ times the number of statistics satisfying $t(\mathbf{x}^*) \geq t(\mathbf{x})$. In other words, we approximate the probability of obtaining a test statistic ast least as extreme as the "original" test statistic $t(\mathbf{x})$.

```
set.seed(314)

linearmodel = lm(log(meas)~pers, data = bilirubin)
Fval = summary(linearmodel)$fstatistic[1] # Value of F-statistic
Fval
```

```
##    value
## 3.669775
```

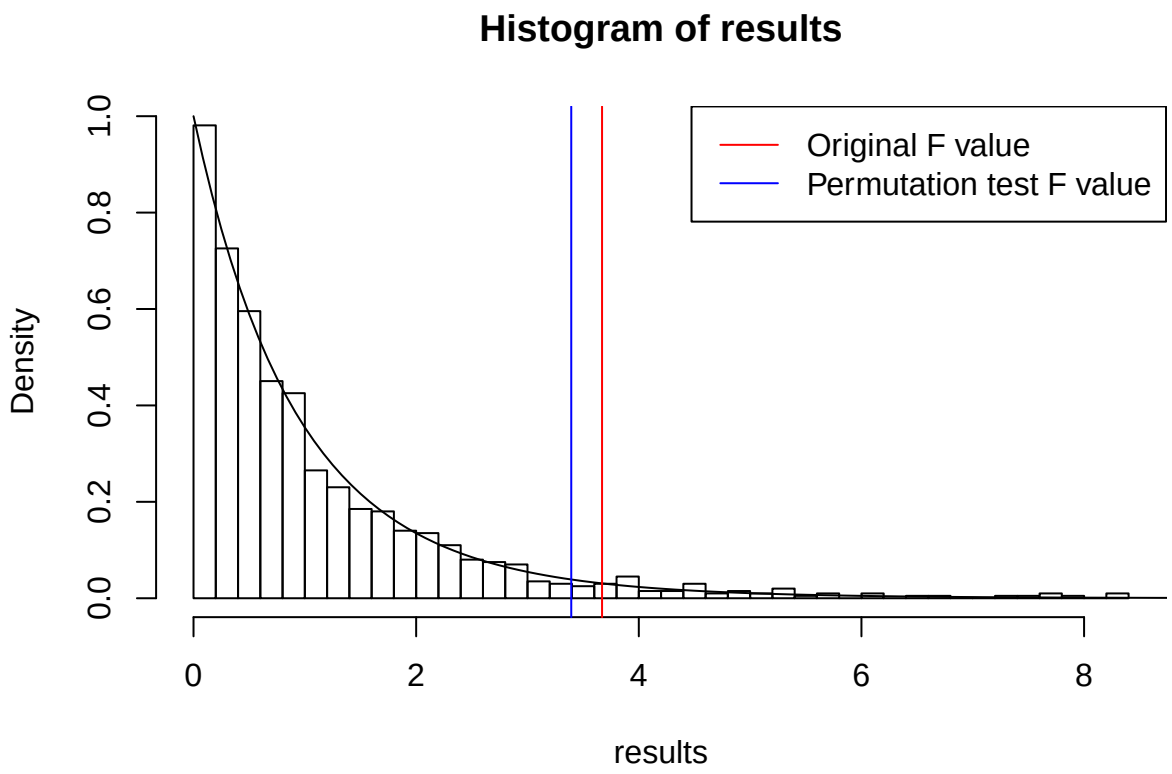

Figure 4: Result from running the permutation test

```
n1 = 11; n2 = 10; n3 = 8; n = n1 + n2 + n3
permTest <- function(){
  permbilirubin = copy(bilirubin)
  permbilirubin$meas = sample(bilirubin$meas,n, replace = F)
  linearmodel = lm(log(meas)~pers, data = permbilirubin)
  return(summary(linearmodel)$fstatistic[1])
}
```

Fitting the linear regression on the original dataset gives a p-value of 0.03946, rather significant. Results from running the permutation test with $B = 999$ is shown in figure 4. re more useful when trying to obtain a p-value when you do not know the distribution.

```
## [1] "Permutation test p-value:"
```

```
## [1] 0.04904905
```

We see that the statistics obtained by permutation nicely fit the distribution they come from. Furthermore, the new p-value is very similar to the old one, and approaches the original one nicely for large B. In this case, we know the distribution under the null hypothesis, and therefore the correct p-value. Permutation tests are more useful to approximate p-values when we do not know the null hypothesis distribution. Here we have essentially shown that we can use a permutation test in this case, and it nicely approximates the p-value it is supposed to approximate.

C.1

The log-likelihood function for the complete data $(x_i, y_i), i = 1, ..., n$ is based on data points that are inaccessible directly, "latent", or in some way not observed. Given these, the log-likelihood can be maximized to yield the most likely distribution parameters. We known their respective distributions,

$$x_i \sim \exp(\lambda_0) = \lambda_1 e^{-y_i \lambda_1},$$
$$y_i \sim \exp(\lambda_1) = \lambda_1 e^{-y_i \lambda_1}.$$

They are in addition indepentent of each other, which means the each data pair can be written as a product. Furthermore, the likelihood function of the joint distribution is just the product of each pair, since they are i.i.d.:

$$(x_i, y_i) \sim \lambda_0 \lambda_1 e^{-x_i \lambda_0} e^{-y_i \lambda_1},$$
$$f(\mathbf{x}, \mathbf{y} | \lambda_0, \lambda_1) = \prod_{i=1}^{n} \lambda_0 \lambda_1 e^{-x_i \lambda_0 - y_i \lambda_1},$$
$$\ln f(\mathbf{x}, \mathbf{y}) | \lambda_0, \lambda_1) = n \ln(\lambda_0 \lambda_1) - \lambda_0 \sum_{i=1}^{n} x_i - \lambda_1 \sum_{i=1}^{n} y_i.$$

If the latent variables $\mathbf{x}, \mathbf{y}$ were observed and known, the parameter optimization would be easy to execute in the classical likelihood-manner: take the derivative of the log-likelihood (given the exact observations) with resepct to λ_0 and λ_1 and set to zero to find to maximum likelihood.

The difference to this situation is that the data points to be used in the likelihood is not observed directly, but approximated. To approximate the data points $\mathbf{x}, \mathbf{y}$ needed to maximize the likelihood, we need (ironically) to use the parameters. These are of course not known either, since the whole point of giving exact data to the likelihood is to find the right parameters for the distribution.

This is were the iterative EM-style comes into play. Since we need to know our data $\mathbf{x}, \mathbf{y}$ in order to maximize the parameters, and we need the parameters to obtain the best guess on the unobserved data, we do exactly that:

- Approximate the expectations of $\mathbf{x}$ and $\mathbf{y}$ with the current parameter guess to maximize a likelihood that is as correct as possible.
- Maximize the likelihood as normal, using the approximated values for the data and obtaining new maximized parameters.
- Repeat the E- and M- steps to get increasingly better approximated data and hence increasingly better parameter estimates.
- Stop the algorithm when the new parameter guesses in the M-step and the parameters on which the data is based on is the E-step are coinciding/converged.

To approximate the latent data in the E-step we are conditioning on the observed data and the current parameter values:

$$z_i = \max(x_i, y_i) \quad \text{for} \quad i = 1, ..., n$$
$$u_i = \mathrm{I}(x_i \geq y_i) \quad \text{for} \quad i = 1, ..., n.$$

and

$$\hat{\lambda}_0 = \lambda_0^{(t)}, \quad \hat{\lambda}_1 = \lambda_1^{(t)}.$$

By conditioning on z_i and u_i and assuming we know some (approximate) values for the parameters as well, λ_0, λ_1, we can compute the expected value of the resulting log-likelihood:

$$\mathbb{E}[\ln f(\mathbf{x},\mathbf{y}|\mathbf{z},\mathbf{u},\lambda_0^{(t)},\lambda_1^{(t)})] = n\ln(\lambda_0\lambda_1) - \lambda_0\sum_{i=1}^{n}\mathbb{E}[x_i] - \lambda_1\sum_{i=1}^{n}\mathbb{E}\big[y_i\big]$$

We rewrite x_i and y_i in terms of the variables we are conditioning on get the expectation. For each of the unobserved variables there are two cases: the first one is that the variable is the greater or equal to the other, in which case it is known and it is simply the max-value z_i times the indicator function u_i for x_i, or $1-u_i$ for y_i:

$$\mathbb{E}[x_i|u_i=1,z_i] = u_i z_i.$$

The second case is where the latent variable in question is the smaller one. The expectation is taken of an exponential distribution, bounded above by the maximum variable and normalized to suit the given area, as follows:

$$\begin{aligned}\mathbb{E}\big[x_i\big|u_i=0,z_i,\lambda_0,\lambda_1\big] &= \mathbb{E}\left[\frac{\lambda_0 e^{-x_i\lambda_0}}{\int_0^{z_i}\lambda_0 e^{-x_i\lambda_0}\,dx_i}\right]\\ &= \frac{\int_0^{z_i} x_i\lambda_0 e^{-x_i\lambda_0}\,dx_i}{(1-e^{-z_i\lambda_0})}\\ &= \frac{\int_0^{z_i} x_i\lambda_0 e^{-x_i\lambda_0}\,dx_i}{1-e^{-z_i\lambda_0}}\end{aligned}$$

The integral is easily evaluated using integration by parts:

$$\begin{aligned}\int_0^{z_i}\lambda_0 x_i e^{-x_i\lambda_0}\,dx_i &= -\lambda_0 x_i e^{-\lambda_0 x_i}\Big|_0^{z_i} - \int_0^{z_i}\lambda_0 e^{-x_i\lambda_0}\,dx_i\\ &= -\lambda_0 z_i e^{-\lambda_0 x_i} + e^{-\lambda_0 z_i} - 1\\ &= e^{-\lambda_0 z_i}(1-\lambda_0 z_i) - 1.\end{aligned}$$

Substituting into the expectation for $x_i|u_i=0$ we get

$$\begin{aligned}\mathbb{E}\big[x_i\big|u_i=0,z_i,\lambda_0,\lambda_1\big] &= \frac{\int_0^{z_i} x_i\lambda_0 e^{-x_i\lambda_0}\,dx_i}{1-e^{-z_i\lambda_0}}\\ &= \frac{e^{-\lambda_0 z_i}(1-\lambda_0 z_i)-1}{1-e^{-z_i\lambda_0}}\\ &= \frac{1}{\lambda_0} - \frac{z_i}{e^{\lambda_0 z_i}-1}\end{aligned}$$

Hence the full expectations for x_i and y_i are

$$\begin{aligned}\mathbb{E}[\ln f(\mathbf{x},\mathbf{y}|\mathbf{z},\mathbf{u},\lambda_0^t,\lambda_1^t)] &= n\ln(\lambda_0\lambda_1) - \lambda_0\sum_{i=1}^{n}\mathbb{E}[x_i] - \lambda_1\sum_{i=1}^{n}\mathbb{E}\big[y_i\big]\\ &= n(\ln\lambda_0+\ln\lambda_1) - \lambda_0\sum_{i=1}^{n}\Big[\mathbb{E}[x_i|x_i\ge y_i] + \mathbb{E}[x_i|x_i<y_i]\Big] - \lambda_1\sum_{i=1}^{n}\Big[\mathbb{E}[y_i|y_i>x_i] + \mathbb{E}[y_i|y_i\le x_i]\Big]\\ &= n(\ln\lambda_0+\ln\lambda_1) - \lambda_0\sum_{i=1}^{n}\left[u_i z_i + (1-u_i)\left(\frac{1}{\lambda_0^{(t)}} - \frac{z_i}{e^{\lambda_0^{(t)} z_i}-1}\right)\right]\\ &\quad - \lambda_1\sum_{i=1}^{n}\left[(1-u_i)z_i + u_i\left(\frac{1}{\lambda_1^{(t)}} - \frac{z_i}{e^{\lambda_1^{(t)} z_i}-1}\right)\right].\end{aligned}$$

This is the best log-likelihood we have available for a given parameter guess $\hat{\lambda}_0 = \lambda_0^{(t)}, \quad \hat{\lambda}_1 = \lambda_1^{(t)}$. We carry on with the standard procedure for maximizing the likelihood; we optimize with respect to the parameters, using the first order necessary condition of optimization.

$$\begin{aligned}
\frac{Q(\boldsymbol{\lambda}|\boldsymbol{\lambda}^{(\mathbf{t})})}{\partial\lambda_0} &= \frac{n}{\lambda_0} - \sum_{i=1}^{n}\left[u_i z_i + (1-u_i)\left(\frac{1}{\lambda_0^{(t)}} - \frac{z_i}{e^{\lambda_0^{(t)} z_i} - 1}\right)\right], \\
\frac{Q(\boldsymbol{\lambda}|\boldsymbol{\lambda}^{(\mathbf{t})})}{\partial\lambda_1} &= \frac{n}{\lambda_1} - \sum_{i=1}^{n}\left[(1-u_i) z_i + u_i\left(\frac{1}{\lambda_1^{(t)}} - \frac{z_i}{e^{\lambda_1^{(t)} z_i} - 1}\right)\right], \\
&\Updownarrow \\
\lambda_0^{(t+1)} &= \frac{n}{\sum_{i=1}^{n}\left[u_i z_i + (1-u_i)\left(\frac{1}{\lambda_0^{(t)}} - \frac{z_i}{\exp(\lambda_0^{(t)} z_i) - 1}\right)\right]}, \\
\lambda_1^{(t+1)} &= \frac{n}{\sum_{i=1}^{n}\left[(1-u_i) z_i + u_i\left(\frac{1}{\lambda_1^{(t)}} - \frac{z_i}{\exp(\lambda_1^{(t)} z_i) - 1}\right)\right]}.
\end{aligned}$$

Furthermore, we can easily calculate the Hessian

$$\nabla^2 \mathbb{E}(\log f(x, y|\lambda_0, \lambda_1)|z, u, \lambda_0^{(t)}, \lambda_1^{(t)}) = n \begin{bmatrix} -\frac{1}{\lambda_0}^2 & 0 \\ 0 & -\frac{1}{\lambda_1}^2 \end{bmatrix},$$

which is negative definite. By the second order sufficient condition our maximum likelihood from earlier is indeed a global maximum.

C.2

Now that we have the log-likelihood we can use the EM-algorithm. In our concrete case, the algorithm is simple as stated earlier:

- Choose initial values $(\lambda_0^{(0)}, \lambda_1^{(0)})$
- Maximize $E(\log f(x, y|\lambda_0, \lambda_1)|z, u, \lambda_0^{(t)}, \lambda_1^{(t)})$ with respect to (λ_0, λ_1), and choose this as $(\lambda_0^{(t+1)}, \lambda_1^{(t+1)})$
- Iterate until convergence

Checking convergence is easy if the desired parameteres are known. In our case, we stop the algorithm if we reach some maximum number of iterations, or if the 2-norm of the difference between the last and current iterates are below some tolerance. Resulting convergence is shown in figure 5. We see that the two values converge rapidly, λ_0 using basically one iteration to converge to the desired value. A better way to check for convergence might be to take the difference between the log-likelihoods of each step, terminating then algorithm when the difference is sufficiently close to zero.

```
z = read.table('z.txt', header = FALSE, sep = "", dec = ".")
u = read.table('u.txt', header = FALSE, sep = "", dec = ".")
n = length(z[,])
lambda_0_next <- function(lambda_0_prev, u, z){
  sum = 0
  for (i in 1:n){
    if (u[i] == 1){sum = sum + z[i]}
    else {sum = sum + 1.0/lambda_0_prev - z[i]/(exp(lambda_0_prev * z[i])-1)}
  }
  return(n/sum)
}
```

```
lambda_1_next <- function(lambda_1_prev, u, z){
  sum = 0
  for (i in 1:n){
    if (u[i] == 0){sum = sum + z[i]}
    else {sum = sum + 1.0/lambda_1_prev - z[i]/(exp(lambda_1_prev * z[i])-1)}
  }
  return(n/sum)
}
EM <- function(u, z, lambda_0_init = 1.0, lambda_1_init = 1.0, plot=0){
  MAXITER = 50
  k = MAXITER
  TOL = 1e-8
  lambda_0_list = numeric(MAXITER)
  lambda_1_list = numeric(MAXITER)
  lambda_0_list[1] = lambda_0_init
  lambda_1_list[1] = lambda_1_init
  for (j in 1:(MAXITER-1)){
      lambda_0_list[j+1] = lambda_0_next(lambda_0_list[j], u, z)
      lambda_1_list[j+1] = lambda_1_next(lambda_1_list[j], u, z)
      if(sqrt((lambda_0_list[j+1]-lambda_0_list[j])^2 + (lambda_1_list[j+1]-lambda_1_list[j])^2) < TOL)
        k = j
        break
      }
  }
  if(plot){
    plot(lambda_1_list[1:k], type="b", main="Convergence of EM-algorithm", xlab = "Iterations", ylab=""
    lines(lambda_0_list[1:k], type="b", col="blue")
    legend("topleft", legend= c(expression(lambda[1]^(t)), expression(lambda[0]^(t))), col=c("red", "blu
  }
  return(list("lambda_0" = lambda_0_list[k], "lambda_1" = lambda_1_list[k]))
}
EM_result = EM(u[,],z[,], plot=1)
```

```
EM_lambda_0 = EM_result$lambda_0
EM_lambda_1 = EM_result$lambda_1
EM_lambda_0 # resulting lambda_0 from the EM algorithm
```

```
## [1] 3.465735
```

```
EM_lambda_1 # resulting lambda_1 from the EM algorithm
```

```
## [1] 9.353215
```

C.3

The EM-algorithm gives us estimates $(\hat{\lambda}_0, \hat{\lambda}_1)$, but says nothing about the standard deviations, bias or correlation of our estimators. Bootstrapping comes to the rescue. The algorithm can be written simply as

- For $b = 1...B$ bootstrap samples
- sample u^b and corresponding z^b from the original dataset
- use EM algorithm to find $\hat{\lambda}_0^b$ and $\hat{\lambda}_1^b$
- optional: Use $\hat{\lambda}_0^{b-1}$ and $\hat{\lambda}_1^{b-1}$ as initial values in the EM algorithm for (hopefully) faster convergence

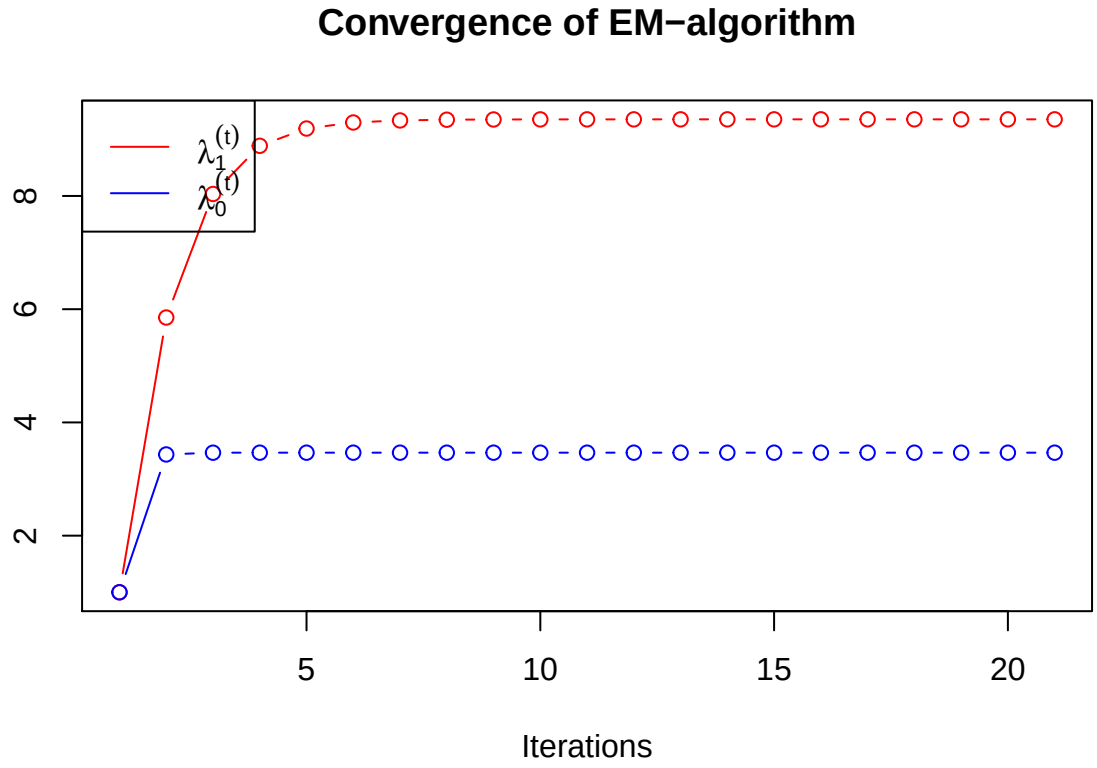

Figure 5: Result from running the EM-algorithm$

- extract some interesting feature (mean, standard deviation and correlation) from the new populations $(\hat{\boldsymbol{\lambda}}_0^*, \hat{\boldsymbol{\lambda}}_1^*) = (\hat{\lambda}_0^1, ...\hat{\lambda}_0^B, \hat{\lambda}_1^1, ...\hat{\lambda}_1^B)$.

```
set.seed(316)
B = 10000 # Number of bootstrap samples
lambda_0_strapped = numeric(B)
lambda_1_strapped = numeric(B)
lambda_0_init = 1.0
lambda_1_init = 1.0
for (i in 1:B){
  strapped_indices = sample(1:n, n, replace=T)
  EM_result = EM(u[strapped_indices,], z[strapped_indices,], lambda_0_init, lambda_1_init, 0)
  lambda_0_strapped[i] = EM_result$lambda_0
  lambda_1_strapped[i] = EM_result$lambda_1
  lambda_0_init = EM_result$lambda_0
  lambda_1_init = EM_result$lambda_1
}
sd(lambda_0_strapped) # standard deviation of lambda_0
```

```
## [1] 0.2480356
```

```
sd(lambda_1_strapped) # standard deviation of lambda_1
```

```
## [1] 0.8035822
```

```
mean(lambda_0_strapped) - EM_lambda_0 # estimated bias of lambda_0
```

```
## [1] 0.01965293
```

```r
mean(lambda_1_strapped) - EM_lambda_1 # estimatedbias of lambda_1
```

```
## [1] 0.07853582
```

```r
cor(lambda_0_strapped, lambda_1_strapped) # correlation between lambda_0 and lambda_1
```

```
## [1] -0.006514024
```

Here we use $B = 10000$ bootstrap samples to arrive at the results. We see that the standard deviation of λ_0 is lower than λ_1, which is probably just beccause λ_1 is larger than λ_0. The bias is similar. We also see that the two parameters are negatively correlated. For the standard deviations, we only need a couple of tens of bootstrap samples to aquire the answer. For the other parameters, we need a lot more samples in order to obtain consistent answers.

When we have an estimate for the bias, we can correct our estimated $(\hat{\lambda}_0, \hat{\lambda}_1)$ by subtracting the bias to obtain an unbiased estimator. However, when we do this, we increase the variance of our estimator. This is easily seen by taking the variance of our bias corrected estimator

$$\hat{\theta}_c = \hat{\theta} - \hat{\text{bias}}(\theta), \quad \text{var}(\theta_c) = \text{var}(\theta) + \text{var}(\hat{\text{bias}}(\theta)) \geq \text{var}(\theta)$$

.

Because the bias is so small relative to the size of the estimators and the variance of the estimators, we would probably avoid correcting the bias here. If the variance is small and the bias large compared to the value of our estimators, we might want to correct for bias. Actually, many statistical methods actually aim to have larger bias, like Ridge regression, in order to reduce the variance.

C.4

Now we try to find the likelihood $f_{Z_i,U_i}(z_i, u_i|\lambda_0, \lambda_1)$ analytically, which we can use to find the maximum likelihood estimators $\hat{\lambda}_0$ and $\hat{\lambda}_1$. We can do this similarily to what we did in C.1. We begin by looking at the cumulative distribution

$$P(Z_i < z, U_i = u|\lambda_0, \lambda_1) = \begin{cases} P(Z_i < z, U_i = 0|\lambda_0, \lambda_1), & \text{for } X_i < Y_i \\ P(Z_i < z, U_i = 1|\lambda_0, \lambda_1), & \text{for } X_i > Y_i \end{cases}.$$

We solve one case by calculating the cumulative distribution and differentiating

$$P(Z_i < z, U_i = 0|\lambda_0, \lambda_1) = P(Z_i < z, X_i < Y_i|\lambda_0, \lambda_1) = \int_0^z \int_0^{Y_i} \lambda_1 \exp(-\lambda_0 Y_i)\lambda_0 \exp(-\lambda_1 X_i) dX_i dY_i$$

$$\int_0^z \lambda_1 \exp(-\lambda_1 Y_i)(1 - \exp(-\lambda_0 Y_i)) = -\exp(-\lambda_1 z) + \frac{\lambda_1}{\lambda_0 + \lambda_1} \exp(-(\lambda_0 + \lambda_1)z) - \frac{\lambda_1}{\lambda_0 + \lambda_1}$$

$$p(Z_i = z, U_i = 0|\lambda_0\lambda_1) = \frac{d}{dz} P(Z_i < z, U_i = 0|\lambda_0, \lambda_1) = \lambda_1 \exp(-\lambda_1 Z_i)(1 - \exp(-\lambda_0 Z_i))$$

using a similar method for $U_i = 1$, we arrive at the density

$$p(Z_i, U_i|\lambda_0, \lambda_1) = U_i \lambda_0 \exp(-\lambda_0 Z_i)(1 - \exp(-\lambda_1 Z_i)) + (1 - U_i)\lambda_1 \exp(-\lambda_1 Z_i)(1 - \exp(-\lambda_0 Z_i)).$$

Because of the indicator variable U_i, we can write the likelihood as

$$L(\lambda_0, \lambda_1 | z_i, u_i) = \prod_i p(Z_i, U_i = 0 | \lambda_0, \lambda_1) \prod_i p(Z_i, U_i = 1 | \lambda_0, \lambda_1)$$

$$\begin{aligned} l(\lambda_0, \lambda_1 | z_i, u_i) &= \sum_i u_i \log(\lambda_0 \exp(-\lambda_0 z_i)(1 - \exp(-\lambda_1 z_i))) + (1 - u_i) \log(\lambda_1 \exp(-\lambda_1 z_i)(1 - \exp(-\lambda_0 z_i))) \\ &= \sum_i u_i (\log(\lambda_0) - \lambda_0 z_i + \log(1 - \exp(-\lambda_1 z_i))) + (1 - u_i)(\log(\lambda_1 - \lambda_1 z_i) + \log(1 - \exp(-\lambda_0 z_i))) \end{aligned}$$

This is a rather nasty log-likelihood, and finding an analytic optimum seems impossible. We can find the gradient

$$\nabla l(\lambda_0, \lambda_1 | z_i u_i) = \left(\sum_i u_i (\frac{1}{\lambda_0} - z_i + (1 - u_i) \frac{z_i}{\exp(lambda_0 z_i)}, \sum_i (1 - u_i)(\frac{1}{\lambda_1} - z_i + u_i \frac{z_i}{\exp(\lambda_1 z_i)} \right).$$

Finding the zeroes of the gradient would most likely require gradients anyways.. We see that there are no cross terms, meaning the hessian will be diagonal, and we see that it will most likely be negative definite, meaning there exists a unique optimal solution. We can use the `optim` function in R to do the dirtywork for us, not using the calculated gradient.

```
u_optim = copy(u[,])
z_optim = copy(z[,])
likelihood <- function(lambda_vec){
  sum = 0
  lambda_0 = lambda_vec[1]; lambda_1 = lambda_vec[2];
  for (i in 1:n){
    if (u_optim[i] == 1){sum = sum + log(lambda_0*exp(-lambda_0*z_optim[i])*(1-exp(-lambda_1*z_optim[i]
    else{ sum = sum + log(lambda_1*exp(-lambda_1*z_optim[i])*(1-exp(-lambda_0*z_optim[i])))}
  }
  return(sum)
}
init = c(3,9)
result = optim(init, likelihood, control = list(fnscale=-1))
optim_lambda_0 = result$par[1]
optim_lambda_1 = result$par[2]
optim_lambda_0 - EM_lambda_0 # Difference from EM algorithm and optim for lambda_0
```

```
## [1] 0.0002109881
```

```
optim_lambda_1 - EM_lambda_1 # Difference from EM algorithm and optim for lambda_1
```

```
## [1] -0.0005564619
```

We see that we obtain similar results as our EM-algorithm. The computation time of both the EM-algorithm and using `optim` is rather non-significant. `optim` would probably work a lot better if we calculated the gradient and chose algorithms more carefully. In this case, the EM-algorithm is probably more effective.

So, what is the difference between using EM and "normal" numerical optimization? `optim` is more tempting to use as it is already implemented, meaning less code. However, in order to use `optim`, we need the likelihood, which was not trivial to find in this setting, and might be impossible in other settings. On larger problems, we probably require the gradient and maybe even hessian as well, or the optimization would be slow and innacurate. In hard problems, you would need an optimization expert in order to find the maximum likelihood estimate, and optimization is hard.

When that is said, we also had to derive some expressions for the EM-algorithm. In this case we we're very lucky, as the optimal values were easy to find analytically. In cases when we cannot do this, we would have

to use numerical optimization anyways. In addition, we saw that we had to implement a bootstrap algorithm in order to approximate the errors of our algorithm. In `optim`, this is already implemented.

In conclusion, the EM-algorithm has its uses, but in some ways, good ol' analytical or numerical optimization is better suited, as long as the situation allows it.