732A91 Lab 3

Fanny Karelius (fanka300), Milda Poceviciute (milpo192)

3 maj 2018

Question 1: Normal model, mixture of normal model with semi-conjugate prior

a) Normal model

The daily precipitation $y = \{y_1, ..., y_n\}$ is assumed to be independently normally distributed, $y|\mu, \sigma^2 \sim N(\mu, \sigma^2)$, where μ, σ^2 are unknown. The priors $\mu \sim N(\mu_0, \tau_0^2)$ and $\sigma^2 \sim Inv - \chi^2(\nu_0, \sigma_0^2)$ are independent. The conditional posteriors are

$$\mu|\sigma^2, y \sim N(\mu_n, \tau_n^2)$$

and

$$\sigma^2|\mu, y \sim Inv - \chi^2(\nu_n, \frac{\nu_0\sigma_0^2 + \sum_{i=1}^{n}(y_i - \mu)^2}{\nu_0 + n})$$

where $\mu_n = w\bar{y} + (1-w)\mu_0$, $w = \frac{n/\sigma^2}{\frac{n}{\sigma^2}+\frac{1}{\tau_0^2}}$, $\frac{1}{\tau_n^2} = \frac{n}{\sigma^2} + \frac{1}{\tau_0^2}$ and $\nu_n = \nu_0 + n$.

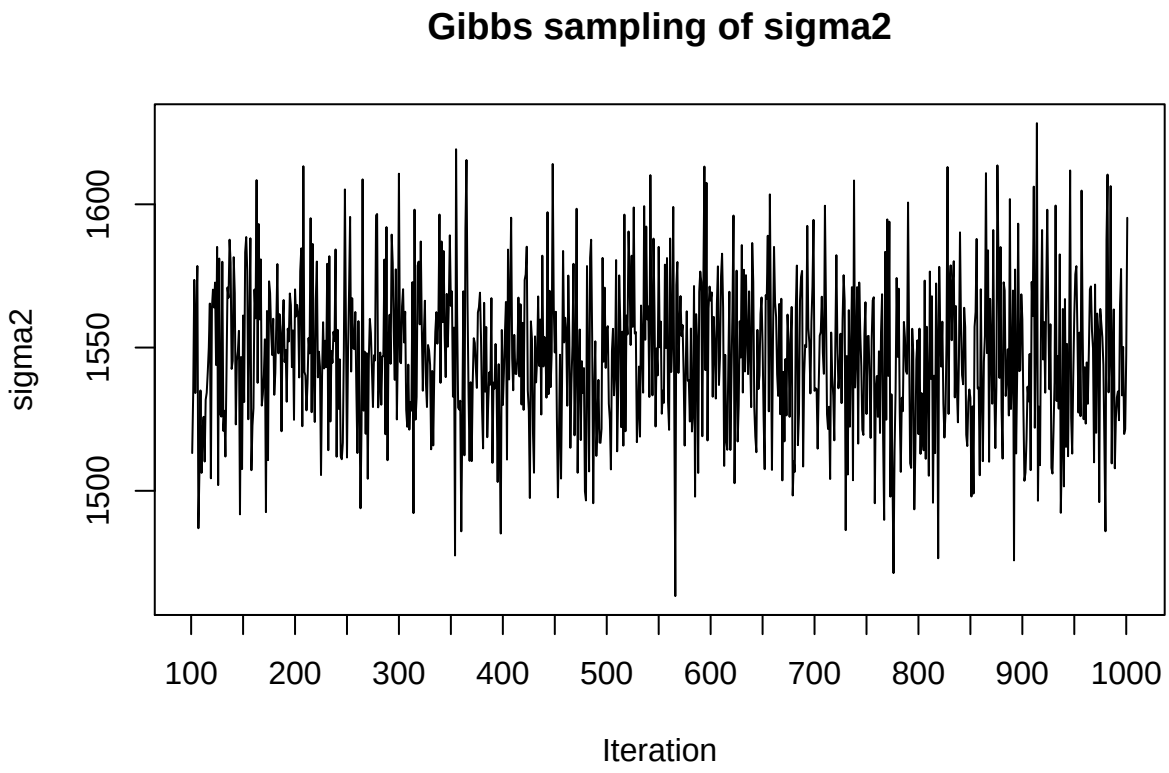
i)

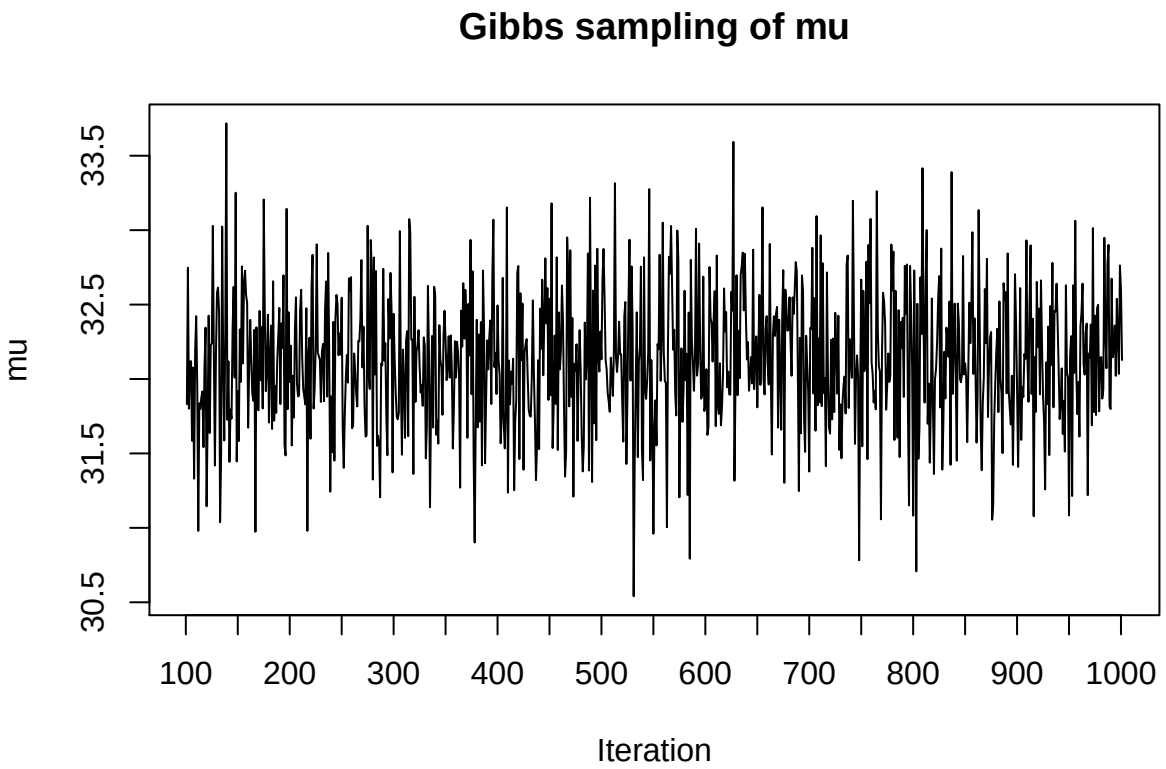
Gibbs sampling implementation:

```
gibbs_sampler <- function(iter, data, mu0, tau20, nu0, sigma20){
  N <- nrow(data)
  x_bar <- mean(data[,1])
  nun <- nu0+N
  mu <- rnorm(1,mu0, sqrt(tau20))
  sigma2 <- nu0*sigma20/rchisq(1,nu0)
  results <- matrix(ncol=2,nrow=iter+1)
  results[1,1]<-mu
  results[1,2]<-sigma2
  colnames(results)<-c("mu", "sigma2")

  for(i in 1:iter){
    w <- (N/results[i,2])/((N/results[i,2])+(1/tau20))
    mun <- w*x_bar+(1-w)*mu0
    tau2n <- 1/((N/results[i,2])+(1/tau20))
    mu <- rnorm(1,mun, sqrt(tau2n))
    para_n <- (nu0*sigma20+sum((data$V1-mu)^2))/nun
    sigma2 <- nun*para_n/rchisq(1,nun)
    results[i+1,]<-c(mu, sigma2)
  }
  results
}
```

ii)

```
rain<-read.table("Rainfall.dat",header=FALSE)

# Weakly informative priors based on our guesses about the possible paramter values
mu0 <-0
tau20 <-50
nu0 <-5
sigma20 <-20

burn_in <- 100
n <- 1000
rain_gibbs <- gibbs_sampler(n, rain, mu0, tau20, nu0, sigma20)
rain_gibbs <- as.data.frame(rain_gibbs)

post_mu <- mean(rain_gibbs$mu[burn_in:n])
post_sigma2 <- mean(rain_gibbs$sigma2[burn_in:n])
```





The plots of the trajectories of the sampled Markov chains of σ^2 and μ seem to be converging well. The Gibbs sampling takes large steps from iteration to iteration and the values does not appear to be (highly) correlated.

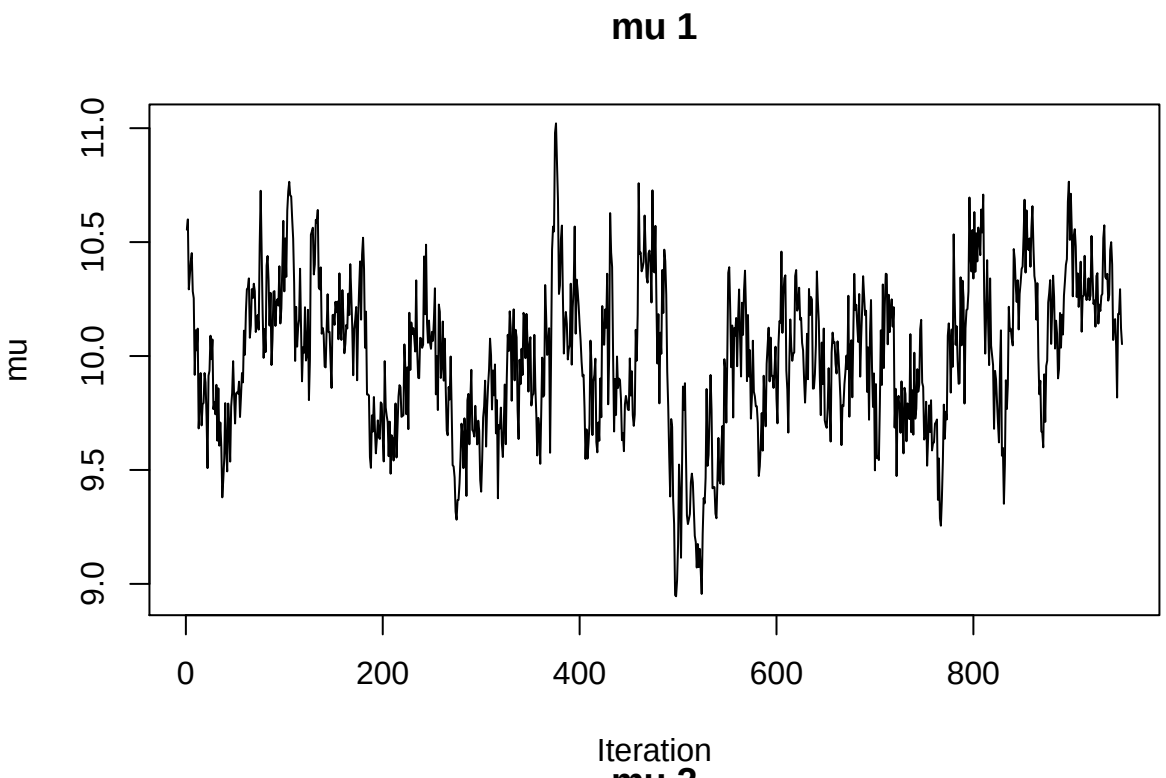
b) Mixture normal model

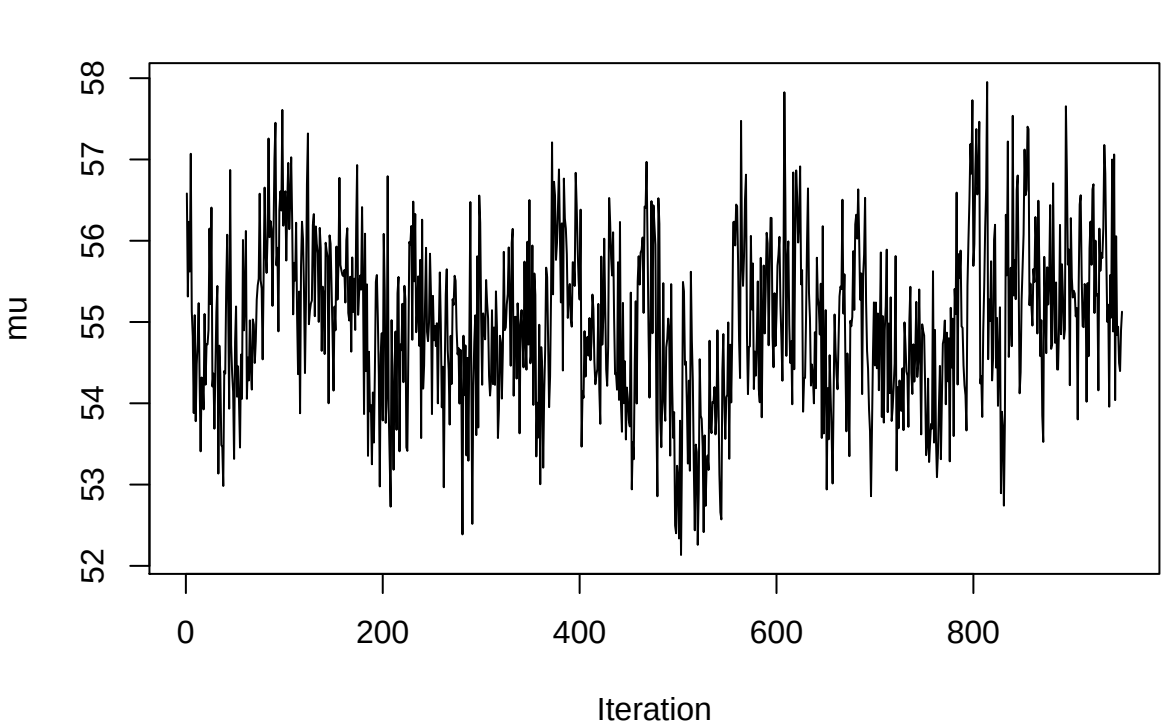
```
rawData <- rain
x <- as.matrix(rawData)

# Model options
nComp <- 2    # Number of mixture components

# Prior options
alpha <- 10*rep(1,nComp) # Dirichlet(alpha)
muPrior <- rep(0,nComp) # Prior mean of mu
tau2Prior <- rep(10,nComp) # Prior std of mu
sigma2_0 <- rep(var(x),nComp) # s20 (best guess of sigma2)
nu0 <- rep(4,nComp) # degrees of freedom for prior on sigma2

# MCMC options
nIter <- 1000 # Number of Gibbs sampling draws

# Plotting options
plotFit <- TRUE
# lineColors <- c("blue", "green", "magenta", 'yellow')
# sleepTime <- 0.1 # Adding sleep time between iterations for plotting
################   END USER INPUT ###############
```

```
###### Defining a function that simulates from the
rScaledInvChi2 <- function(n, df, scale){
  return((df*scale)/rchisq(n,df=df))
}

####### Defining a function that simulates from a Dirichlet distribution
rDirichlet <- function(param){
  nCat <- length(param)
  piDraws <- matrix(NA,nCat,1)
  for (j in 1:nCat){
    piDraws[j] <- rgamma(1,param[j],1)
  }
  piDraws = piDraws/sum(piDraws) # Diving every column of piDraws by the sum of the elements in that co
  return(piDraws)
}

# Simple function that converts between two different representations of the mixture allocation
S2alloc <- function(S){
  n <- dim(S)[1]
  alloc <- rep(0,n)
  for (i in 1:n){
    alloc[i] <- which(S[i,] == 1)
  }
  return(alloc)
}

# Initial value for the MCMC
nObs <- length(x)
S <- t(rmultinom(nObs, size = 1 , prob = rep(1/nComp,nComp))) # nObs-by-nComp matrix with component all
mu <- matrix(ncol=nComp, nrow=nIter+1)
mu[1,] <- quantile(x, probs = seq(0,1,length = nComp))
sigma2 <- matrix(ncol=nComp, nrow=nIter+1)
sigma2[1,] <- rep(var(x),nComp)
probObsInComp <- rep(NA, nComp)

# Setting up the plot
xGrid <- seq(min(x)-1*apply(x,2,sd),max(x)+1*apply(x,2,sd),length = 100)
xGridMin <- min(xGrid)
xGridMax <- max(xGrid)
mixDensMean <- rep(0,length(xGrid))
effIterCount <- 0
hist_x <- max(invisible(hist(x)$density))
ylim <- c(0,2*hist_x)


for (k in 1:nIter){
  #message(paste('Iteration number:',k))
  alloc <- S2alloc(S) # Just a function that converts between different representations of the group al
  nAlloc <- colSums(S)
  #print(nAlloc)
  # Update components probabilities
  pi <- rDirichlet(alpha + nAlloc)
```

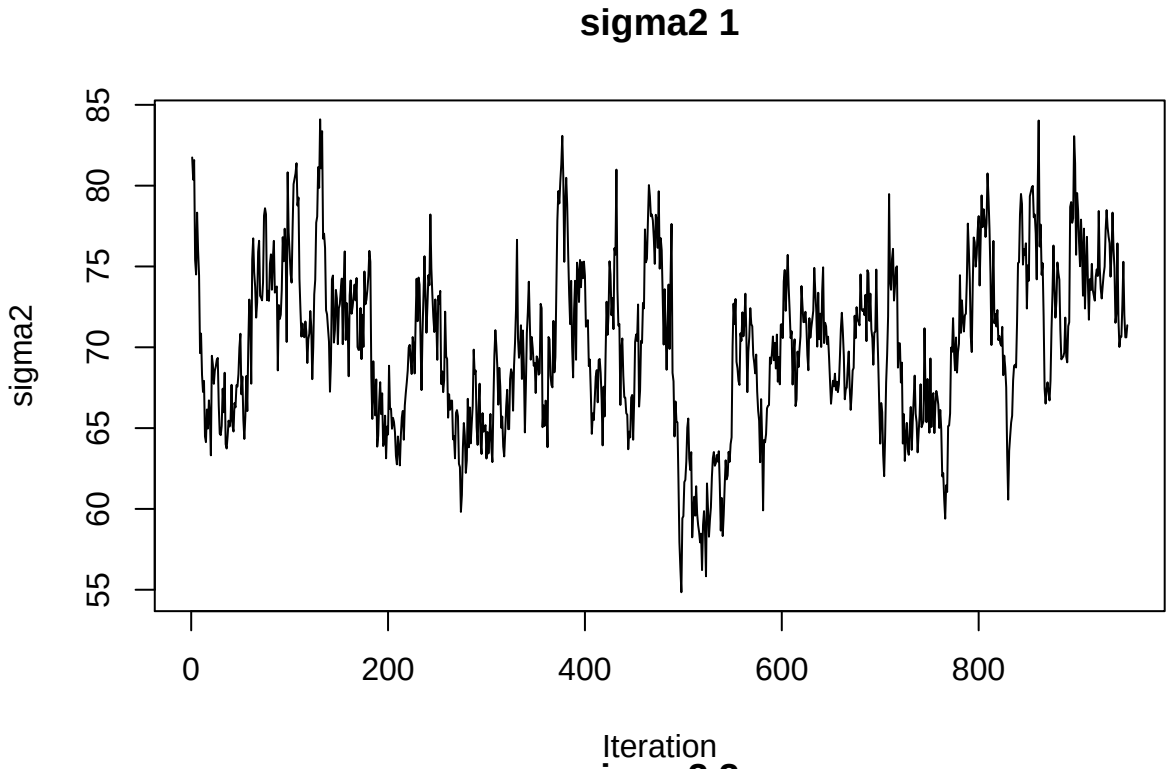
```
  # Update mu's
  for (j in 1:nComp){
    precPrior <- 1/tau2Prior[j]
    precData <- nAlloc[j]/sigma2[k,j]
    precPost <- precPrior + precData
    wPrior <- precPrior/precPost
    muPost <- wPrior*muPrior + (1-wPrior)*mean(x[alloc == j])
    tau2Post <- 1/precPost
    mu[k+1,j] <- rnorm(1, mean = muPost, sd = sqrt(tau2Post))
  }

  # Update sigma2's
  for (j in 1:nComp){
    sigma2[k+1,j] <- rScaledInvChi2(1, df = nu0[j] + nAlloc[j], scale = (nu0[j]*sigma2_0[j] + sum((x[al
  }

  # Update allocation
  for (i in 1:nObs){
    for (j in 1:nComp){
      probObsInComp[j] <- pi[j]*dnorm(x[i], mean = mu[k+1,j], sd = sqrt(sigma2[k+1,j]))
    }
    S[i,] <- t(rmultinom(1, size = 1 , prob = probObsInComp/sum(probObsInComp)))
  }

  # Printing the fitted density against data histogram
   if (plotFit && (k%%1 ==0)){
     effIterCount <- effIterCount + 1
  #   hist(x, breaks = 20, freq = FALSE, xlim = c(xGridMin,xGridMax), main = paste("Iteration number",k
     mixDens <- rep(0,length(xGrid))
  #   components <- c()
  for (j in 1:nComp){
       compDens <- dnorm(xGrid,mu[k+1,j],sd = sqrt(sigma2[k+1,j]))
       mixDens <- mixDens + pi[j]*compDens
  #     lines(xGrid, compDens, type = "l", lwd = 2, col = lineColors[j])
  #     components[j] <- paste("Component ",j)
  #   }
  mixDensMean <- ((effIterCount-1)*mixDensMean + mixDens)/effIterCount
  #
  #   lines(xGrid, mixDens, type = "l", lty = 2, lwd = 3, col = 'red')
  #   legend("topleft", box.lty = 1, legend = c("Data histogram",components, 'Mixture'),
  #          col = c("black",lineColors[1:nComp], 'red'), lwd = 2)
  #   Sys.sleep(sleepTime)
   }

   }

}
```

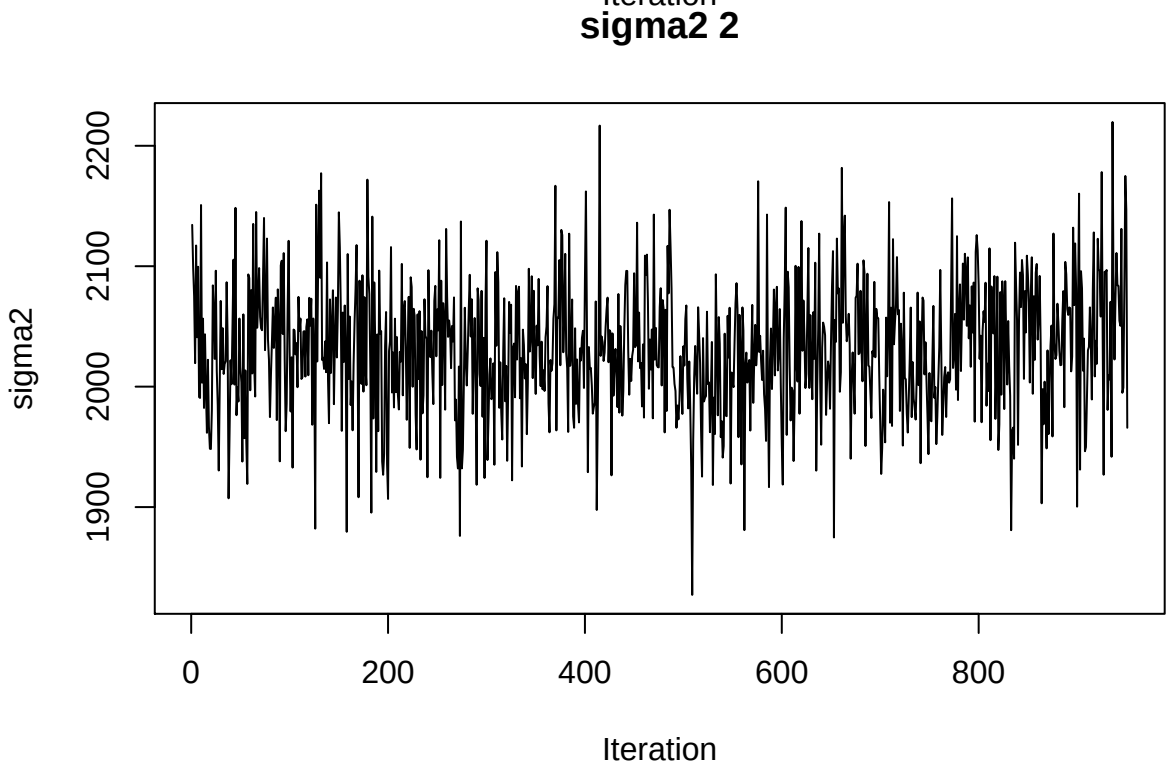
```
for(i in 1:nComp){
  plot(mu[50:nIter,i], xlab="Iteration", ylab="mu", main=paste0("mu ", i), type="l")
}
```






```
for(i in 1:nComp){
  plot(sigma2[50:nIter,i], xlab="Iteration", ylab="sigma2", main=paste0("sigma2 ", i), type="l")
}
```

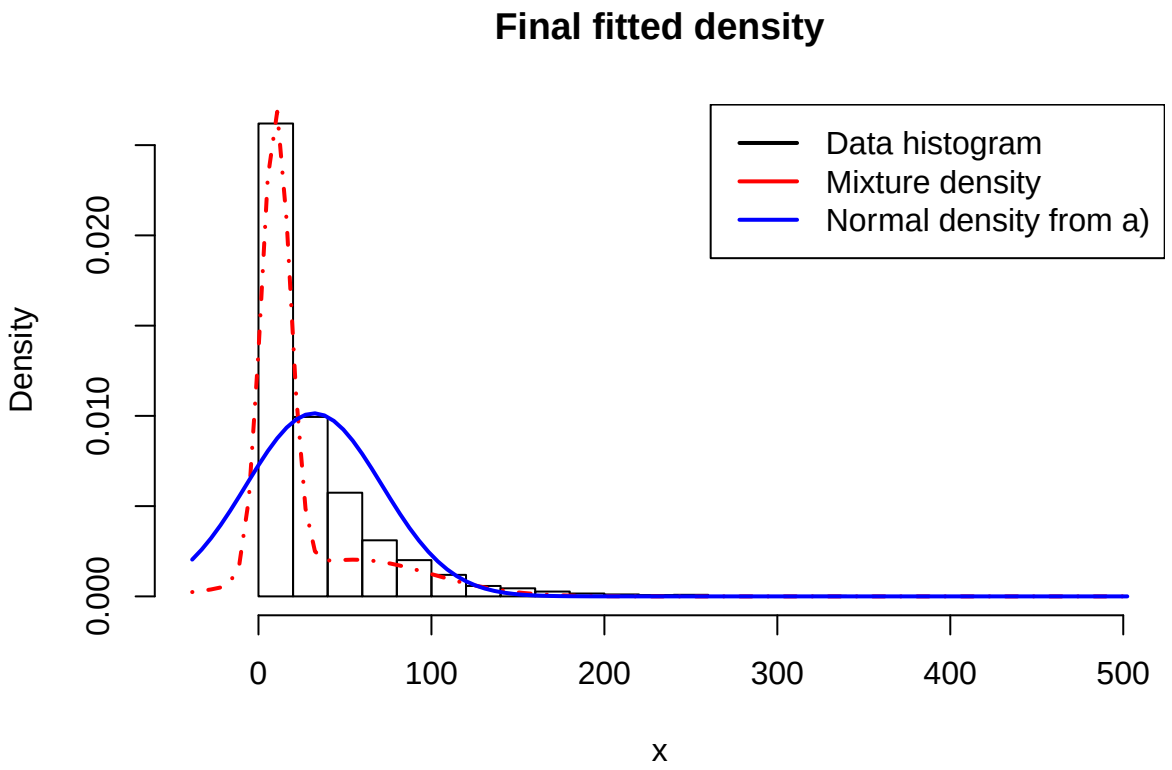




From the trajectory plots above we conclude that the μs and σ^2s of both components do not appear to converge well. The iterations seem to be correlated.

c)

```
hist(x, breaks = 20, freq = FALSE, xlim = c(xGridMin,xGridMax), main = "Final fitted density")
lines(xGrid, mixDensMean, type = "l", lwd = 2, lty = 4, col = "red")
lines(xGrid, dnorm(xGrid, mean = post_mu, sd = sqrt(post_sigma2)), type = "l", lwd = 2, col = "blue")
legend("topright", box.lty = 1, legend = c("Data histogram","Mixture density","Normal density from a)")
```



Neither the normal density from a) nor the mixture model from b) seem to capture the distribution of the data well. The mixture model captures the highest peak better, but misses the rest of the data, whereas the normal density encapsulates most of the data except the highest peak.

Question 2: Time series models in Stan

a)

We simulate data $x_{1:T}$ from the AR(1)-process: $x_t = \mu + \phi(x_{t-1} - \mu) + \epsilon_t$, where $\epsilon_t \overset{iid}{\sim} N(0, \sigma^2)$ with $\mu = 10$, $\sigma^2 = 2$, $x_1 = \mu$, $T = 200$ and $\phi \in \{-1, 1\}$.

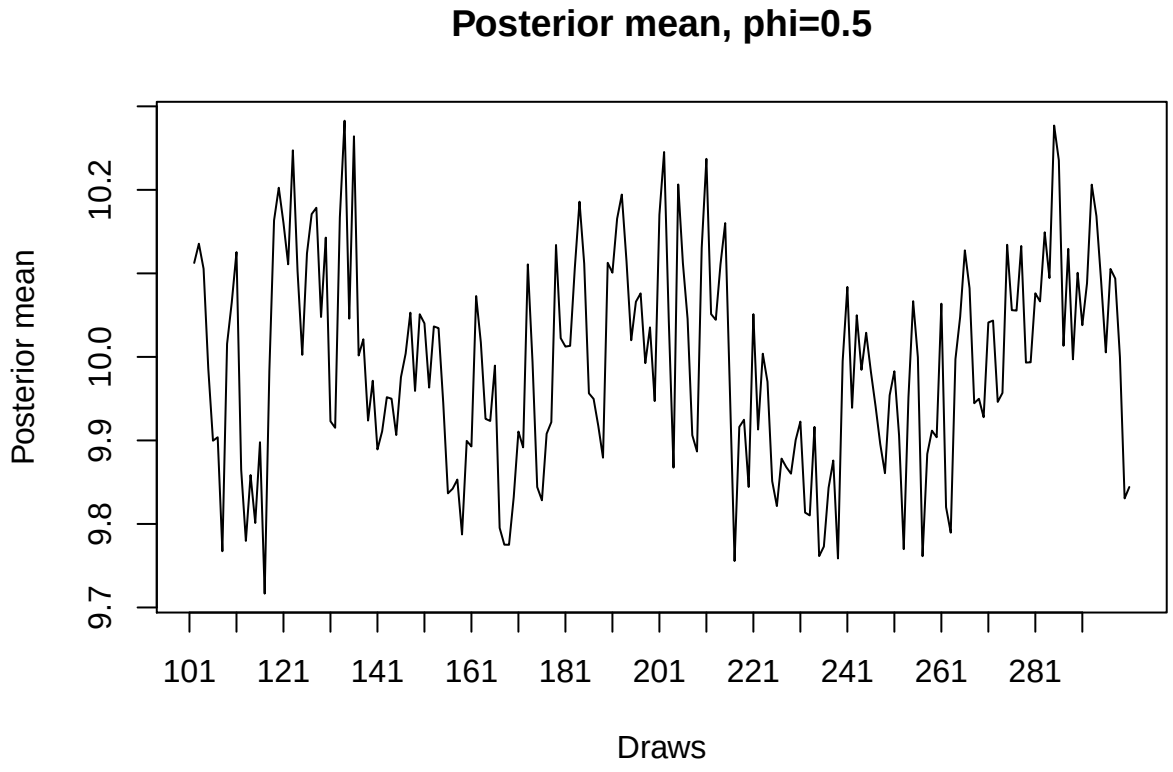
```
#Question 2

library(rstan)

#a)

ar_process = '
data {
  int<lower=0> iter;
  real mu;
```

```
  real<lower=0> sigma2;
  real<lower=-1, upper=1> phi;
}
parameters {
  real x[iter];
}
model {
  for(i in 2:iter){
    x[i] ~ normal(mu + phi*(x[i-1]-mu), sqrt(sigma2));
  }
}'

burnin <- 101
niter <- 300
mu <- 10
init_para <- list(list(x=c(mu,rep(0,niter-1))))
#needs to be a list of a list since each chain needs its own list
fit1<-stan(model_code=ar_process,
           data=list(iter=niter,sigma2=2, phi=0.5, mu=10),
           init = init_para,
           warmup=burnin,
           iter=niter,
           chains = 1,
           control = list(max_treedepth = 15))
```

```
## In file included from /home/milpo192/R/x86_64-pc-linux-gnu-library/3.4/BH/include/boost/config.hpp:3
##                  from /home/milpo192/R/x86_64-pc-linux-gnu-library/3.4/BH/include/boost/math/tools/c
##                  from /home/milpo192/R/x86_64-pc-linux-gnu-library/3.4/StanHeaders/include/stan/math
##                  from /home/milpo192/R/x86_64-pc-linux-gnu-library/3.4/StanHeaders/include/stan/math
##                  from /home/milpo192/R/x86_64-pc-linux-gnu-library/3.4/StanHeaders/include/stan/math
##                  from /home/milpo192/R/x86_64-pc-linux-gnu-library/3.4/StanHeaders/include/stan/math
##                  from /home/milpo192/R/x86_64-pc-linux-gnu-library/3.4/StanHeaders/include/stan/math
##                  from /home/milpo192/R/x86_64-pc-linux-gnu-library/3.4/StanHeaders/include/src/stan/
##                  from file4431454d8138.cpp:8:
## /home/milpo192/R/x86_64-pc-linux-gnu-library/3.4/BH/include/boost/config/compiler/gcc.hpp:186:0: war
##  #  define BOOST_NO_CXX11_RVALUE_REFERENCES
##  ^
## <command-line>:0:0: note: this is the location of the previous definition
##
## SAMPLING FOR MODEL 'e9dd3115a361dc2be0299c0f20cb0509' NOW (CHAIN 1).
##
## Gradient evaluation took 7.1e-05 seconds
## 1000 transitions using 10 leapfrog steps per transition would take 0.71 seconds.
## Adjust your expectations accordingly!
##
##
## WARNING: There aren't enough warmup iterations to fit the
##          three stages of adaptation as currently configured.
##          Reducing each adaptation stage to 15%/75%/10% of
##          the given number of warmup iterations:
##            init_buffer = 15
##            adapt_window = 76
##            term_buffer = 10
##
```

```
## Iteration:   1 / 300 [  0%]  (Warmup)
## Iteration:  30 / 300 [ 10%]  (Warmup)
## Iteration:  60 / 300 [ 20%]  (Warmup)
## Iteration:  90 / 300 [ 30%]  (Warmup)
## Iteration: 102 / 300 [ 34%]  (Sampling)
## Iteration: 131 / 300 [ 43%]  (Sampling)
## Iteration: 161 / 300 [ 53%]  (Sampling)
## Iteration: 191 / 300 [ 63%]  (Sampling)
## Iteration: 221 / 300 [ 73%]  (Sampling)
## Iteration: 251 / 300 [ 83%]  (Sampling)
## Iteration: 281 / 300 [ 93%]  (Sampling)
## Iteration: 300 / 300 [100%]  (Sampling)
##
##  Elapsed Time: 1.58905 seconds (Warm-up)
##                1.38861 seconds (Sampling)
##                2.97765 seconds (Total)
```

```
fit1_postmean <- get_posterior_mean(fit1)[burnin:niter]
plot(fit1_postmean, type="l", xaxt="n", xlab="Draws", ylab="Posterior mean", main="Posterior mean, phi=0
axis(1, at=seq(0,(niter-burnin), by=10), labels=seq(burnin,niter, by=10))
```

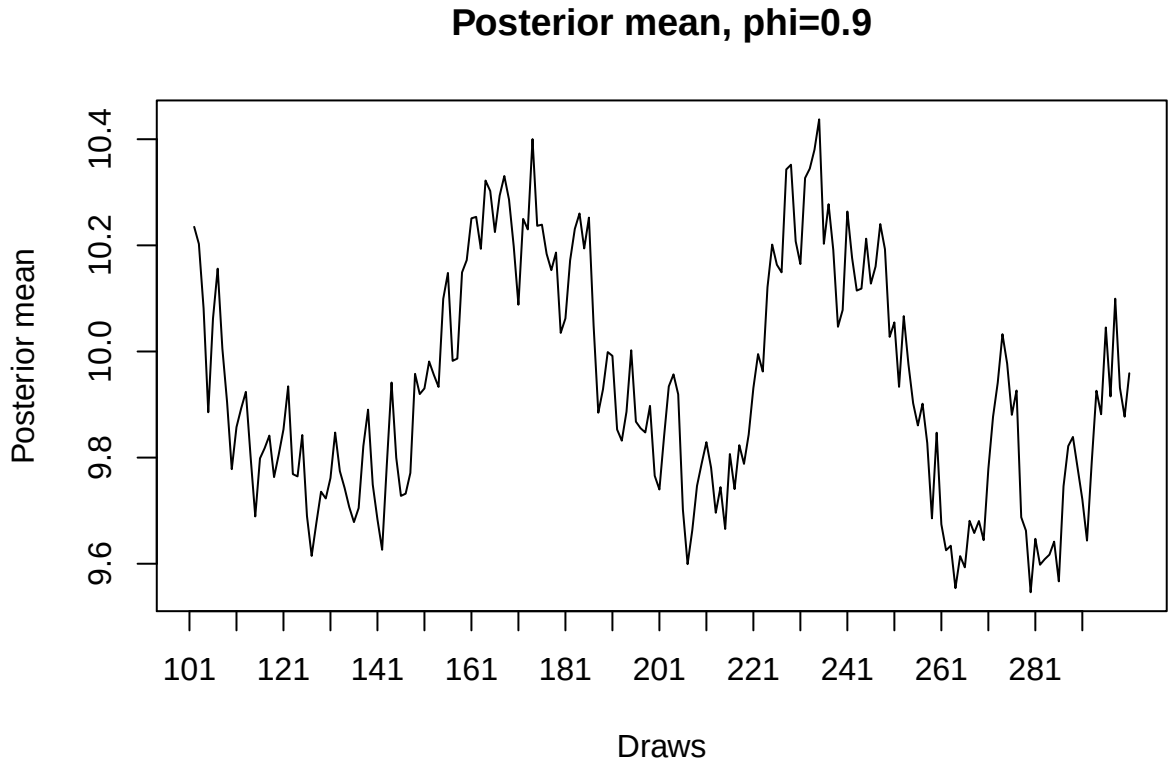



```
fit2<-stan(model_code=ar_process,
           data=list(iter=niter,sigma2=2, phi=0.9, mu=10),
           init = init_para,
           warmup=burnin,
           iter=niter,
           chains = 1,
           control = list(max_treedepth = 15))
```

```
##
## SAMPLING FOR MODEL 'e9dd3115a361dc2be0299c0f20cb0509' NOW (CHAIN 1).
```

```
##
## Gradient evaluation took 3.7e-05 seconds
## 1000 transitions using 10 leapfrog steps per transition would take 0.37 seconds.
## Adjust your expectations accordingly!
##
##
## WARNING: There aren't enough warmup iterations to fit the
##          three stages of adaptation as currently configured.
##          Reducing each adaptation stage to 15%/75%/10% of
##          the given number of warmup iterations:
##            init_buffer = 15
##            adapt_window = 76
##            term_buffer = 10
##
## Iteration:   1 / 300 [  0%]  (Warmup)
## Iteration:  30 / 300 [ 10%]  (Warmup)
## Iteration:  60 / 300 [ 20%]  (Warmup)
## Iteration:  90 / 300 [ 30%]  (Warmup)
## Iteration: 102 / 300 [ 34%]  (Sampling)
## Iteration: 131 / 300 [ 43%]  (Sampling)
## Iteration: 161 / 300 [ 53%]  (Sampling)
## Iteration: 191 / 300 [ 63%]  (Sampling)
## Iteration: 221 / 300 [ 73%]  (Sampling)
## Iteration: 251 / 300 [ 83%]  (Sampling)
## Iteration: 281 / 300 [ 93%]  (Sampling)
## Iteration: 300 / 300 [100%]  (Sampling)
##
##  Elapsed Time: 3.26657 seconds (Warm-up)
##                3.89665 seconds (Sampling)
##                7.16322 seconds (Total)
```

```
#print(fit2)
fit2_postmean <- get_posterior_mean(fit2)[burnin:niter]
plot(fit2_postmean, type="l", xaxt="n", xlab="Draws", ylab="Posterior mean", main="Posterior mean, phi=0
axis(1, at=seq(0,(niter-burnin), by=10), labels=seq(burnin,niter, by=10))
```

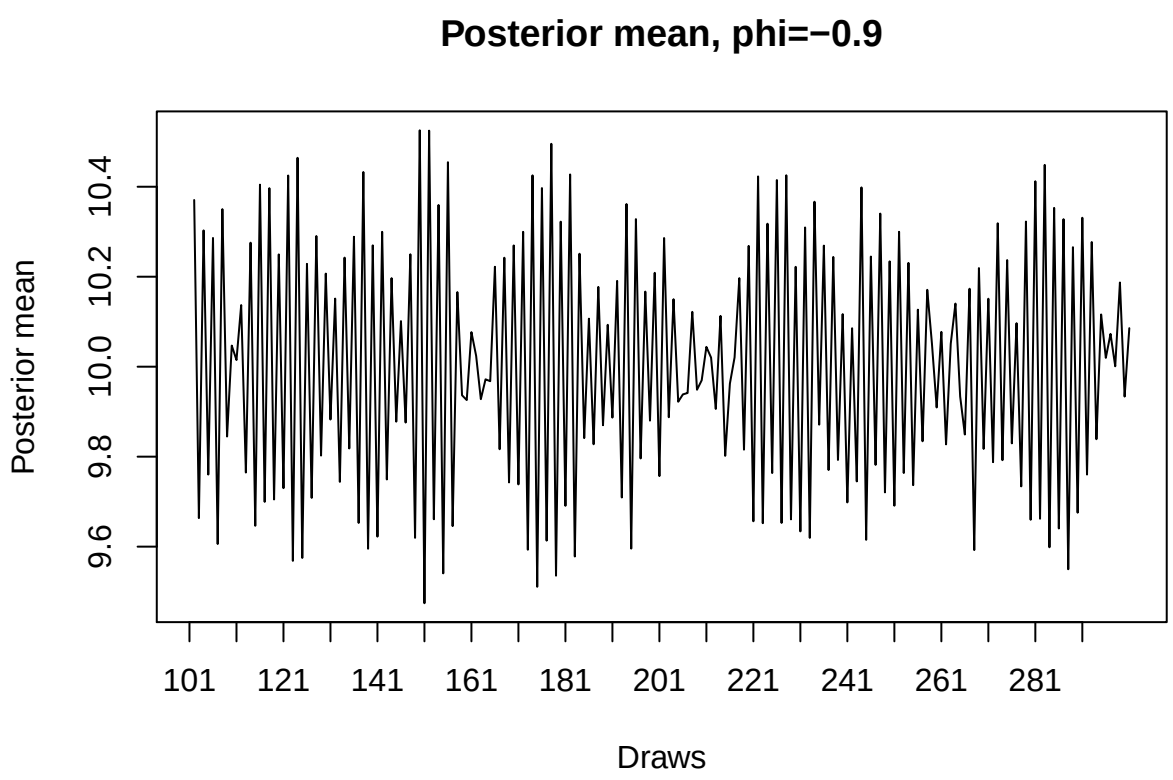



```
fit3<-stan(model_code=ar_process,
           data=list(iter=niter,sigma2=2, phi=-0.9, mu=10),
           init = init_para,
           warmup=burnin,
           iter=niter,
           chains = 1,
           control = list(max_treedepth = 15))
```

```
##
## SAMPLING FOR MODEL 'e9dd3115a361dc2be0299c0f20cb0509' NOW (CHAIN 1).
##
## Gradient evaluation took 3.3e-05 seconds
## 1000 transitions using 10 leapfrog steps per transition would take 0.33 seconds.
## Adjust your expectations accordingly!
##
##
## WARNING: There aren't enough warmup iterations to fit the
##          three stages of adaptation as currently configured.
##          Reducing each adaptation stage to 15%/75%/10% of
##          the given number of warmup iterations:
##            init_buffer = 15
##            adapt_window = 76
##            term_buffer = 10
##
## Iteration:   1 / 300 [  0%]  (Warmup)
## Iteration:  30 / 300 [ 10%]  (Warmup)
## Iteration:  60 / 300 [ 20%]  (Warmup)
## Iteration:  90 / 300 [ 30%]  (Warmup)
## Iteration: 102 / 300 [ 34%]  (Sampling)
## Iteration: 131 / 300 [ 43%]  (Sampling)
```

```
## Iteration: 161 / 300 [ 53%]  (Sampling)
## Iteration: 191 / 300 [ 63%]  (Sampling)
## Iteration: 221 / 300 [ 73%]  (Sampling)
## Iteration: 251 / 300 [ 83%]  (Sampling)
## Iteration: 281 / 300 [ 93%]  (Sampling)
## Iteration: 300 / 300 [100%]  (Sampling)
##
##  Elapsed Time: 2.6856 seconds (Warm-up)
##                5.57785 seconds (Sampling)
##                8.26346 seconds (Total)
```

```
#print(fit3)
fit3_postmean <- get_posterior_mean(fit3)[burnin:niter]
plot(fit3_postmean, type="l", xaxt="n", xlab="Draws", ylab="Posterior mean", main="Posterior mean, phi=
axis(1, at=seq(0,(niter-burnin), by=10), labels=seq(burnin,niter, by=10))
```

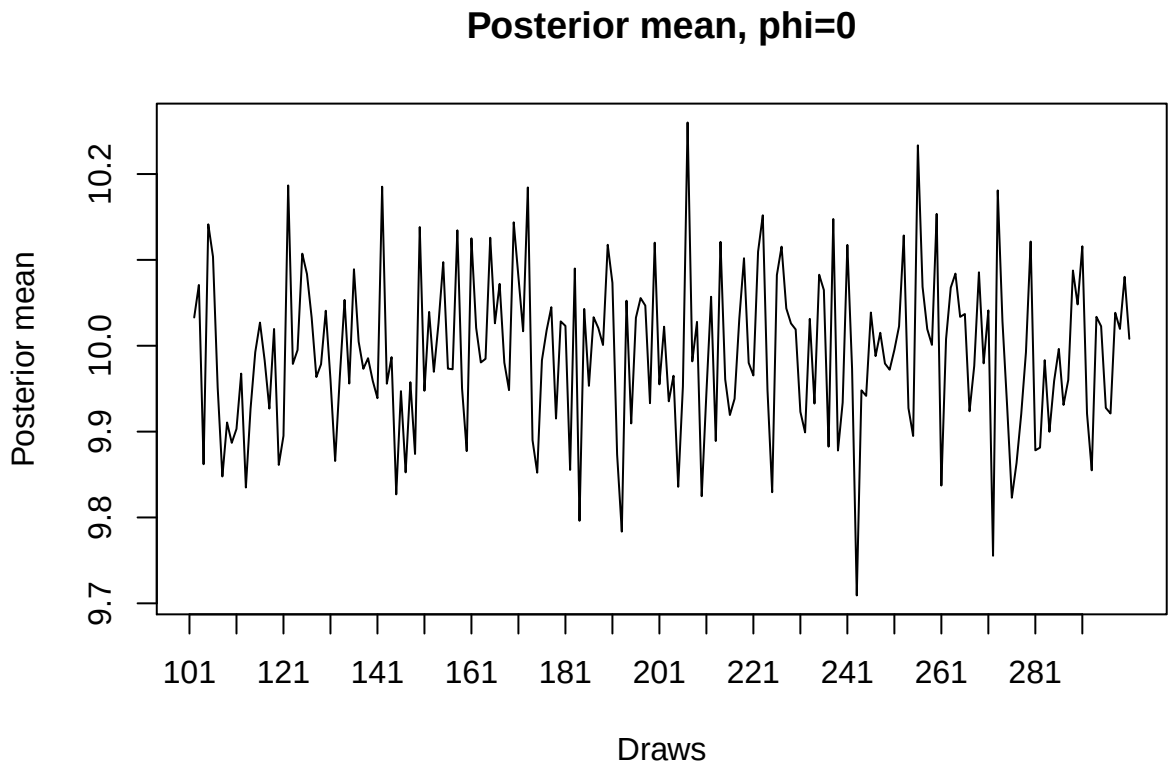



```
fit4<-stan(model_code=ar_process,
           data=list(iter=niter,sigma2=2, phi=0, mu=10),
           init = init_para,
           warmup=burnin,
           iter=niter,
           chains = 1,
           control = list(max_treedepth = 15))
```

```
##
## SAMPLING FOR MODEL 'e9dd3115a361dc2be0299c0f20cb0509' NOW (CHAIN 1).
##
## Gradient evaluation took 3.7e-05 seconds
## 1000 transitions using 10 leapfrog steps per transition would take 0.37 seconds.
## Adjust your expectations accordingly!
##
```

```
##
## WARNING: There aren't enough warmup iterations to fit the
##          three stages of adaptation as currently configured.
##          Reducing each adaptation stage to 15%/75%/10% of
##          the given number of warmup iterations:
##            init_buffer = 15
##            adapt_window = 76
##            term_buffer = 10
##
## Iteration:   1 / 300 [  0%]  (Warmup)
## Iteration:  30 / 300 [ 10%]  (Warmup)
## Iteration:  60 / 300 [ 20%]  (Warmup)
## Iteration:  90 / 300 [ 30%]  (Warmup)
## Iteration: 102 / 300 [ 34%]  (Sampling)
## Iteration: 131 / 300 [ 43%]  (Sampling)
## Iteration: 161 / 300 [ 53%]  (Sampling)
## Iteration: 191 / 300 [ 63%]  (Sampling)
## Iteration: 221 / 300 [ 73%]  (Sampling)
## Iteration: 251 / 300 [ 83%]  (Sampling)
## Iteration: 281 / 300 [ 93%]  (Sampling)
## Iteration: 300 / 300 [100%]  (Sampling)
##
##  Elapsed Time: 9.86499 seconds (Warm-up)
##                123.112 seconds (Sampling)
##                132.977 seconds (Total)
```

```
#print(fit4)
fit4_postmean <- get_posterior_mean(fit4)[burnin:niter]
plot(fit4_postmean, type="l", xaxt="n", xlab="Draws", ylab="Posterior mean", main="Posterior mean, phi=
axis(1, at=seq(0,(niter-burnin), by=10), labels=seq(burnin,niter, by=10))
```



The correlation between x_t and x_{t+h} is $\rho_h = \phi^h$, which means that larger $|\phi|$ gives larger correlation between observations h lags apart.

b)

Using the function in a) we simulate two AR(1)-processes $x_{1:T}$ with $\phi = 0.3$ and $y_{1:T}$ with $\phi = 0.95$. Then we estimate values of ϕ, μ and σ^2 for these two processes using MCMC. The non-informative priors used were:
$\mu \sim N(0, 10)$
$\sigma \sim Exp(0.1)$
$\phi \sim N(0, 1)$
The priors were chosen so they would cover a wide range of values as well as the values that were used in generating AR(1)-process $x_{1:T}$ data.

i)

```
#b)

burnin <- 101
niter <- 300
mu <- 10
init_para <- list(list(x=c(mu,rep(0,niter-1))))

xT <- stan(model_code=ar_process,
           data=list(iter=niter,sigma2=2, phi=0.3, mu=10),
           init = init_para,
           warmup=burnin,
           iter=niter,
           chains = 1,
           control = list(max_treedepth = 15))
```

```
##
## SAMPLING FOR MODEL 'e9dd3115a361dc2be0299c0f20cb0509' NOW (CHAIN 1).
##
## Gradient evaluation took 5.3e-05 seconds
## 1000 transitions using 10 leapfrog steps per transition would take 0.53 seconds.
## Adjust your expectations accordingly!
##
##
## WARNING: There aren't enough warmup iterations to fit the
##          three stages of adaptation as currently configured.
##          Reducing each adaptation stage to 15%/75%/10% of
##          the given number of warmup iterations:
##            init_buffer = 15
##            adapt_window = 76
##            term_buffer = 10
##
## Iteration:   1 / 300 [  0%]  (Warmup)
## Iteration:  30 / 300 [ 10%]  (Warmup)
## Iteration:  60 / 300 [ 20%]  (Warmup)
## Iteration:  90 / 300 [ 30%]  (Warmup)
## Iteration: 102 / 300 [ 34%]  (Sampling)
## Iteration: 131 / 300 [ 43%]  (Sampling)
```

```
## Iteration: 161 / 300 [ 53%]  (Sampling)
## Iteration: 191 / 300 [ 63%]  (Sampling)
## Iteration: 221 / 300 [ 73%]  (Sampling)
## Iteration: 251 / 300 [ 83%]  (Sampling)
## Iteration: 281 / 300 [ 93%]  (Sampling)
## Iteration: 300 / 300 [100%]  (Sampling)
##
##  Elapsed Time: 1.04522 seconds (Warm-up)
##                0.104229 seconds (Sampling)
##                1.14945 seconds (Total)
```

```
xT_postmean <- get_posterior_mean(xT)[burnin:niter]

yT <-  stan(model_code=ar_process,
            data=list(iter=niter,sigma2=2, phi=0.95, mu=10),
            init = init_para,
            warmup=burnin,
            iter=niter,
            chains = 1,
            control = list(max_treedepth = 15))
```

```
##
## SAMPLING FOR MODEL 'e9dd3115a361dc2be0299c0f20cb0509' NOW (CHAIN 1).
##
## Gradient evaluation took 3.5e-05 seconds
## 1000 transitions using 10 leapfrog steps per transition would take 0.35 seconds.
## Adjust your expectations accordingly!
##
##
## WARNING: There aren't enough warmup iterations to fit the
##          three stages of adaptation as currently configured.
##          Reducing each adaptation stage to 15%/75%/10% of
##          the given number of warmup iterations:
##            init_buffer = 15
##            adapt_window = 76
##            term_buffer = 10
##
## Iteration:   1 / 300 [  0%]  (Warmup)
## Iteration:  30 / 300 [ 10%]  (Warmup)
## Iteration:  60 / 300 [ 20%]  (Warmup)
## Iteration:  90 / 300 [ 30%]  (Warmup)
## Iteration: 102 / 300 [ 34%]  (Sampling)
## Iteration: 131 / 300 [ 43%]  (Sampling)
## Iteration: 161 / 300 [ 53%]  (Sampling)
## Iteration: 191 / 300 [ 63%]  (Sampling)
## Iteration: 221 / 300 [ 73%]  (Sampling)
## Iteration: 251 / 300 [ 83%]  (Sampling)
## Iteration: 281 / 300 [ 93%]  (Sampling)
## Iteration: 300 / 300 [100%]  (Sampling)
##
##  Elapsed Time: 9.32497 seconds (Warm-up)
##                15.7554 seconds (Sampling)
##                25.0803 seconds (Total)
```

```
yT_postmean <- get_posterior_mean(yT)[burnin:niter]

ar_mcmc = '
data {
  int<lower=0> iter;
  vector[iter] x;
}
parameters {
  real mu;
  real<lower=0> sigma;
  real<lower=-1, upper=1> phi;
}
model {
// Priors
  mu ~ normal(0,10);
  sigma ~ exponential(0.1); //very weak prior for sigma, but always postive
  phi ~ normal(0, 1);
// model
  for(i in 2:iter){
    x[i] ~ normal(mu + phi*(x[i-1]-mu), sigma);
  }
}'


burnin <- 100
niter <- 300
ndraws <- 200

xT_fit <- stan(model_code=ar_mcmc,
           data=list(iter=ndraws,x=xT_postmean),
           warmup=burnin,
           iter=niter,
           chains = 1,
           control = list(max_treedepth = 15,adapt_delta = 0.99))
```

```
## In file included from /home/milpo192/R/x86_64-pc-linux-gnu-library/3.4/BH/include/boost/config.hpp:3
##                  from /home/milpo192/R/x86_64-pc-linux-gnu-library/3.4/BH/include/boost/math/tools/c
##                  from /home/milpo192/R/x86_64-pc-linux-gnu-library/3.4/StanHeaders/include/stan/math
##                  from /home/milpo192/R/x86_64-pc-linux-gnu-library/3.4/StanHeaders/include/stan/math
##                  from /home/milpo192/R/x86_64-pc-linux-gnu-library/3.4/StanHeaders/include/stan/math
##                  from /home/milpo192/R/x86_64-pc-linux-gnu-library/3.4/StanHeaders/include/stan/math
##                  from /home/milpo192/R/x86_64-pc-linux-gnu-library/3.4/StanHeaders/include/stan/math
##                  from /home/milpo192/R/x86_64-pc-linux-gnu-library/3.4/StanHeaders/include/src/stan/
##                  from file44312f79f97b.cpp:8:
## /home/milpo192/R/x86_64-pc-linux-gnu-library/3.4/BH/include/boost/config/compiler/gcc.hpp:186:0: war
##  #  define BOOST_NO_CXX11_RVALUE_REFERENCES
##  ^
## <command-line>:0:0: note: this is the location of the previous definition
##
## SAMPLING FOR MODEL 'daecf139ddbfa84108dc172cb2a4d679' NOW (CHAIN 1).
##
## Gradient evaluation took 4.3e-05 seconds
## 1000 transitions using 10 leapfrog steps per transition would take 0.43 seconds.
## Adjust your expectations accordingly!
```

```
##
##
## WARNING: There aren't enough warmup iterations to fit the
##          three stages of adaptation as currently configured.
##          Reducing each adaptation stage to 15%/75%/10% of
##          the given number of warmup iterations:
##            init_buffer = 15
##            adapt_window = 75
##            term_buffer = 10
##
## Iteration:   1 / 300 [  0%]  (Warmup)
## Iteration:  30 / 300 [ 10%]  (Warmup)
## Iteration:  60 / 300 [ 20%]  (Warmup)
## Iteration:  90 / 300 [ 30%]  (Warmup)
## Iteration: 101 / 300 [ 33%]  (Sampling)
## Iteration: 130 / 300 [ 43%]  (Sampling)
## Iteration: 160 / 300 [ 53%]  (Sampling)
## Iteration: 190 / 300 [ 63%]  (Sampling)
## Iteration: 220 / 300 [ 73%]  (Sampling)
## Iteration: 250 / 300 [ 83%]  (Sampling)
## Iteration: 280 / 300 [ 93%]  (Sampling)
## Iteration: 300 / 300 [100%]  (Sampling)
##
##  Elapsed Time: 0.910972 seconds (Warm-up)
##                0.016608 seconds (Sampling)
##                0.92758 seconds (Total)
```

```
#pairs(xT_fit)

para_postmean <- get_posterior_mean(xT_fit)[1:3]
mu_post2 <- extract(xT_fit)$mu
phi_post2 <- extract(xT_fit)$phi
sigma2_post2 <- (extract(xT_fit)$sigma)^2

perc1x <- sapply(as.data.frame(xT_fit), FUN=quantile, probs=0.025)[1:3]
perc2x <- sapply(as.data.frame(xT_fit), FUN=quantile, probs=0.975)[1:3]
n_eff_x <- summary(xT_fit)$summary[,"n_eff"][1:3]
```

```
## Posterior means for mu, sigma2, phi:  10.00327 0.006108336 0.4085357

##
##  Upper limits for 95% credible interval:  10.026 0.08512272 0.4891142

##
##  Lower limits for 95% credible interval:  9.987065 0.07097315 0.3323227

##
##  Number of effective posterior samples:  47.99329 5.452517 5.146216
```

From the simulations above we get quite close estimates for μ. The estimate for σ^2 is much lower than the value used in part a). It seems to be highly influenced by the sample variance of the $x_{1:T}$ data, which is 0.0060313. The 95% credible interval of ϕ does not include the value of 0.3, hence the estimation of the ϕ seems to be poor as well.

```
yT_fit <- stan(model_code=ar_mcmc,
               data=list(iter=ndraws,x=yT_postmean),
               warmup=burnin,
```

```
              iter=niter,
              chains = 1)
```

```
##
## SAMPLING FOR MODEL 'daecf139ddbfa84108dc172cb2a4d679' NOW (CHAIN 1).
##
## Gradient evaluation took 4.7e-05 seconds
## 1000 transitions using 10 leapfrog steps per transition would take 0.47 seconds.
## Adjust your expectations accordingly!
##
##
## WARNING: There aren't enough warmup iterations to fit the
##          three stages of adaptation as currently configured.
##          Reducing each adaptation stage to 15%/75%/10% of
##          the given number of warmup iterations:
##            init_buffer = 15
##            adapt_window = 75
##            term_buffer = 10
##
## Iteration:   1 / 300 [  0%]  (Warmup)
## Iteration:  30 / 300 [ 10%]  (Warmup)
## Iteration:  60 / 300 [ 20%]  (Warmup)
## Iteration:  90 / 300 [ 30%]  (Warmup)
## Iteration: 101 / 300 [ 33%]  (Sampling)
## Iteration: 130 / 300 [ 43%]  (Sampling)
## Iteration: 160 / 300 [ 53%]  (Sampling)
## Iteration: 190 / 300 [ 63%]  (Sampling)
## Iteration: 220 / 300 [ 73%]  (Sampling)
## Iteration: 250 / 300 [ 83%]  (Sampling)
## Iteration: 280 / 300 [ 93%]  (Sampling)
## Iteration: 300 / 300 [100%]  (Sampling)
##
##  Elapsed Time: 0.040216 seconds (Warm-up)
##                0.206297 seconds (Sampling)
##                0.246513 seconds (Total)
```

```
Ypara_postmean <- get_posterior_mean(yT_fit)[1:3]

mu_post <- extract(yT_fit)$mu
phi_post <- extract(yT_fit)$phi
sigma2_post <- (extract(yT_fit)$sigma)^2

perc1y <- sapply(as.data.frame(yT_fit), FUN=quantile, probs=0.025)[1:3]
perc2y <- sapply(as.data.frame(yT_fit), FUN=quantile, probs=0.975)[1:3]
n_eff_y<- summary(yT_fit)$summary[,"n_eff"][1:3]
```

```
## Posterior means for mu, sigma2, phi:  9.780424 0.009043204 0.9495121
```

```
##
##  Upper limits for 95% credible interval:  10.01942 0.1032269 0.9501462
```

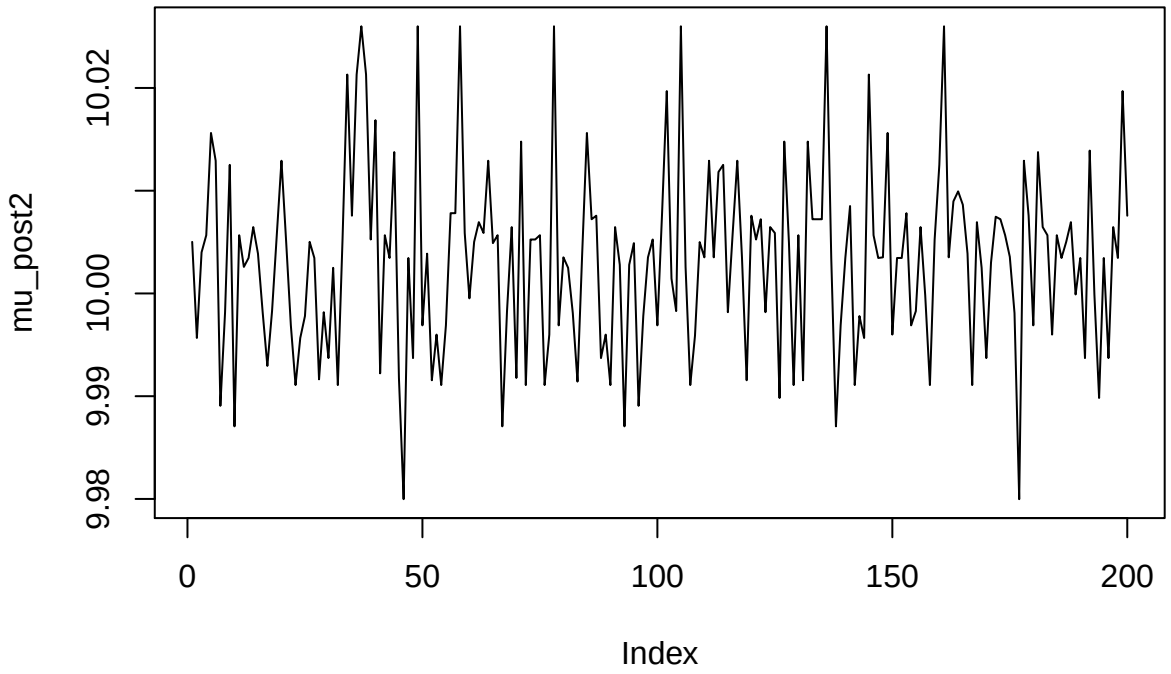
```
##
##  Lower limits for 95% credible interval:  9.544465 0.08732942 0.9489391
```

```
##
##  Number of effective posterior samples:  54.35391 106.8526 187.6466
```

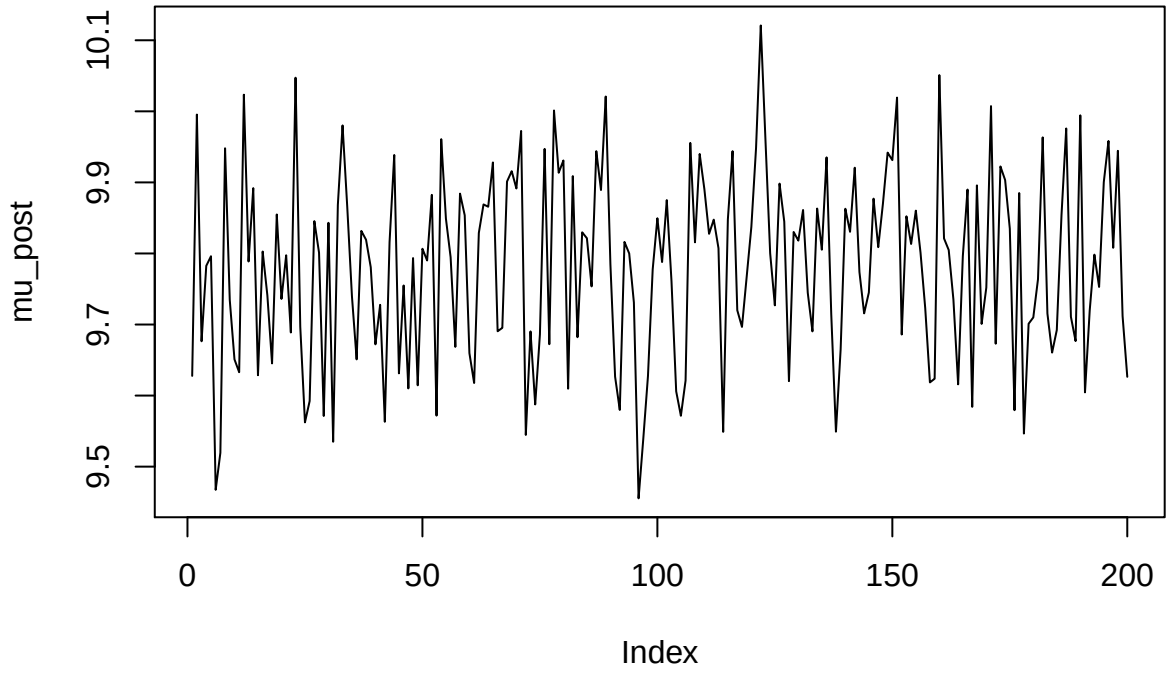
From the simulations above we get quite close estimates for μ and ϕ. However,the estimate for σ^2 is much lower than the value used in part a). It seems to be highly influenced by the sample variance of the $y_{1:T}$ data, which is 320.6930809.

ii)

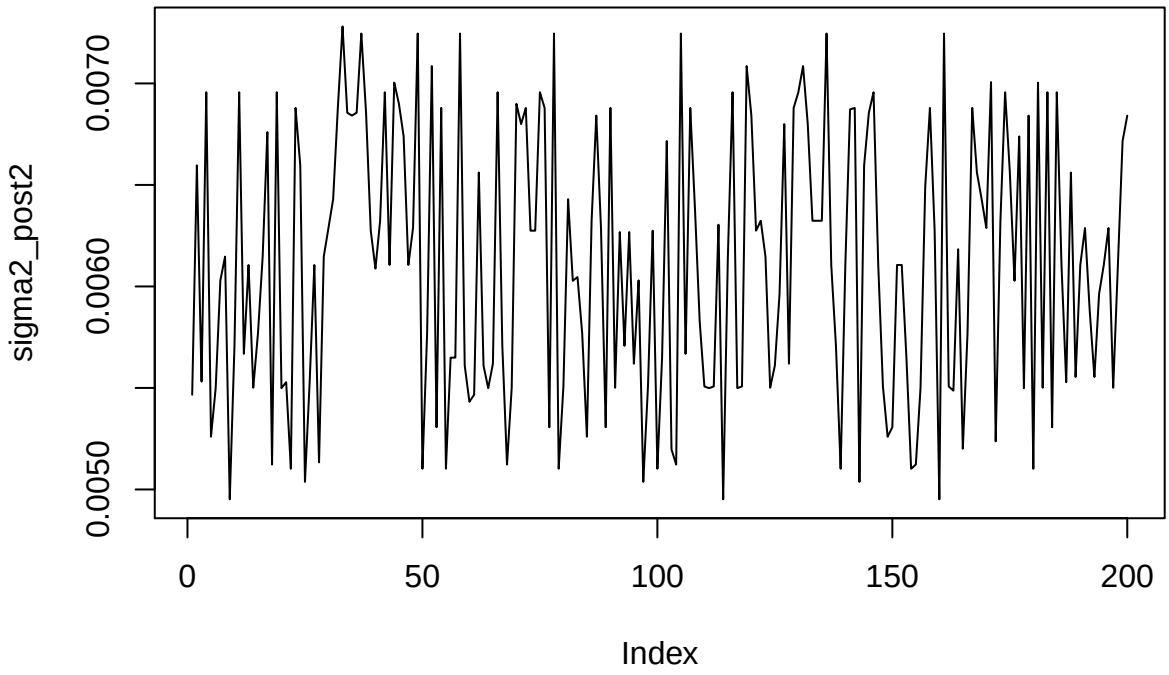
```
# Trajectory plots of the mu
plot(mu_post2, type='l')
```



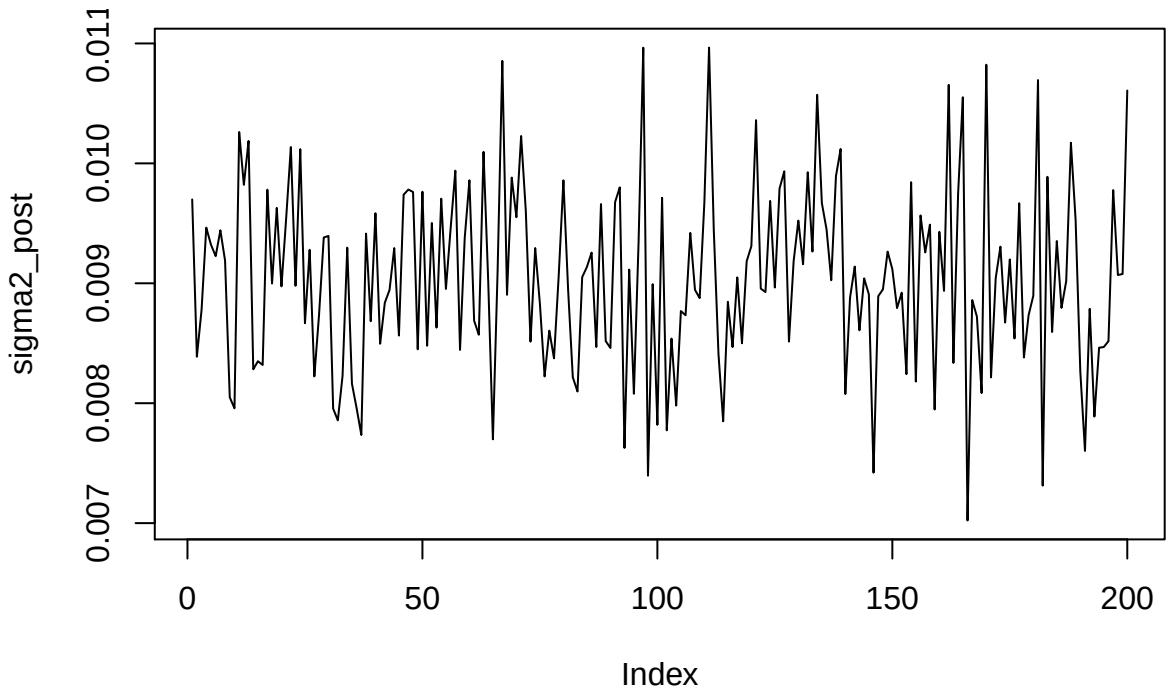

```
plot(mu_post, type='l')
```



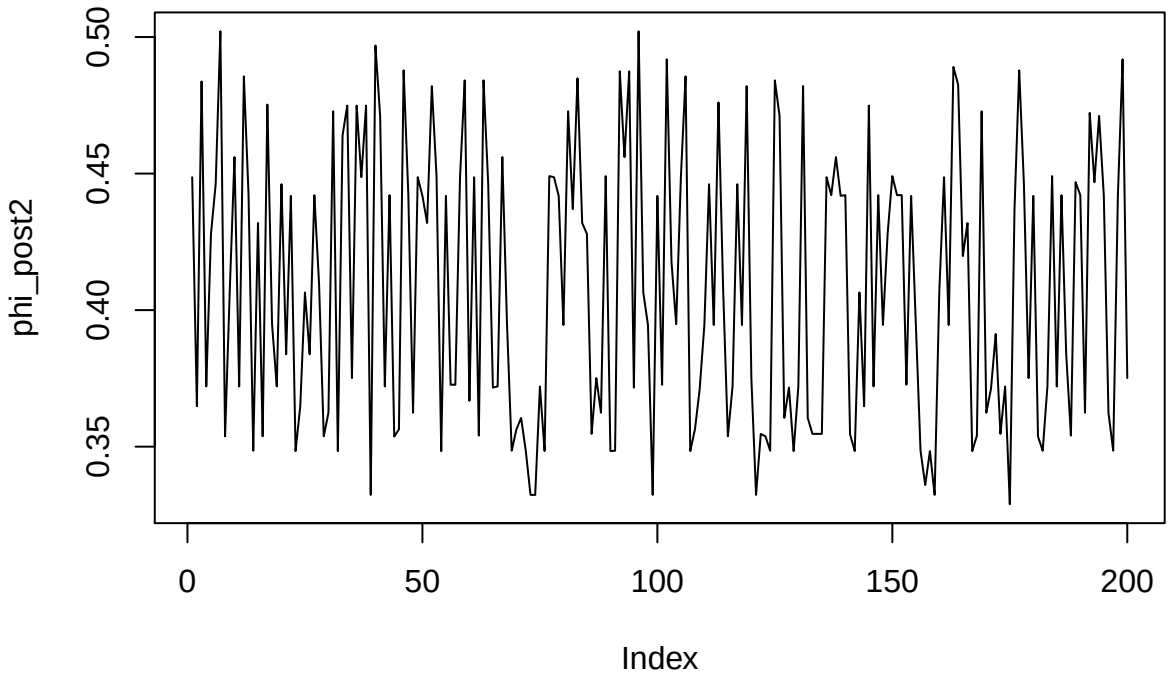
```r
# Trajectory plots of the sigma2

plot(sigma2_post2, type='l')
```



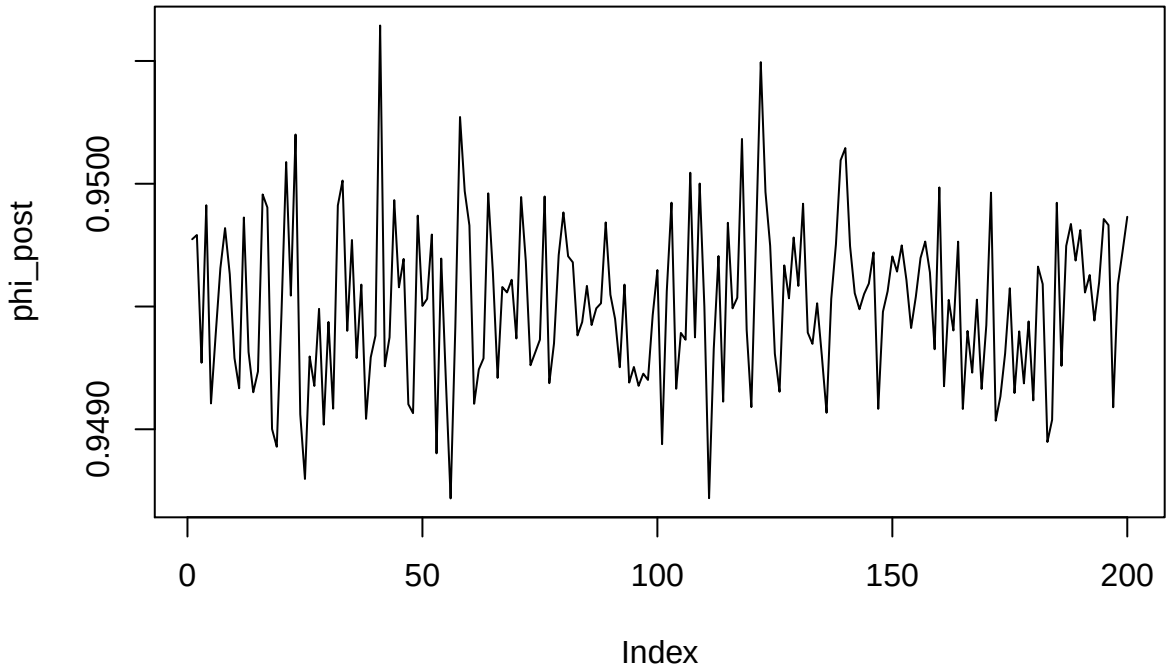

```r
plot(sigma2_post, type='l')
```




```r
# Trajectory plots of the phi

plot(phi_post2, type='l')
```

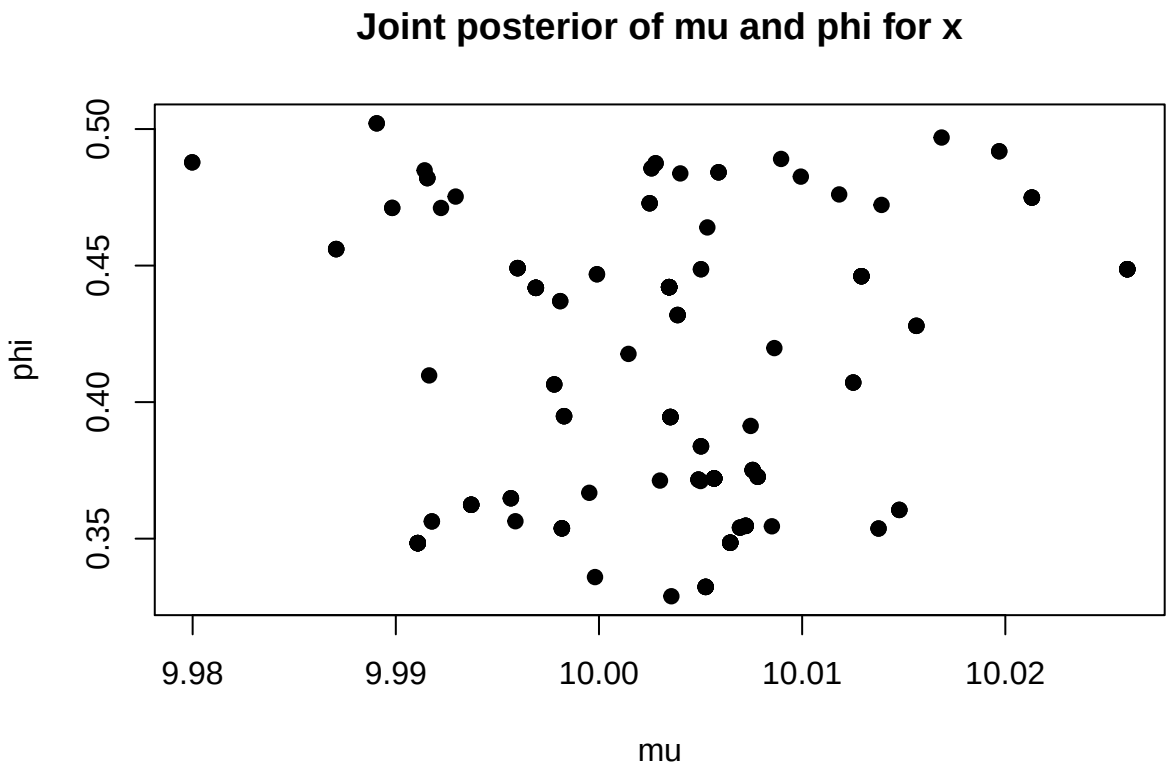



```
plot(phi_post, type='l')
```

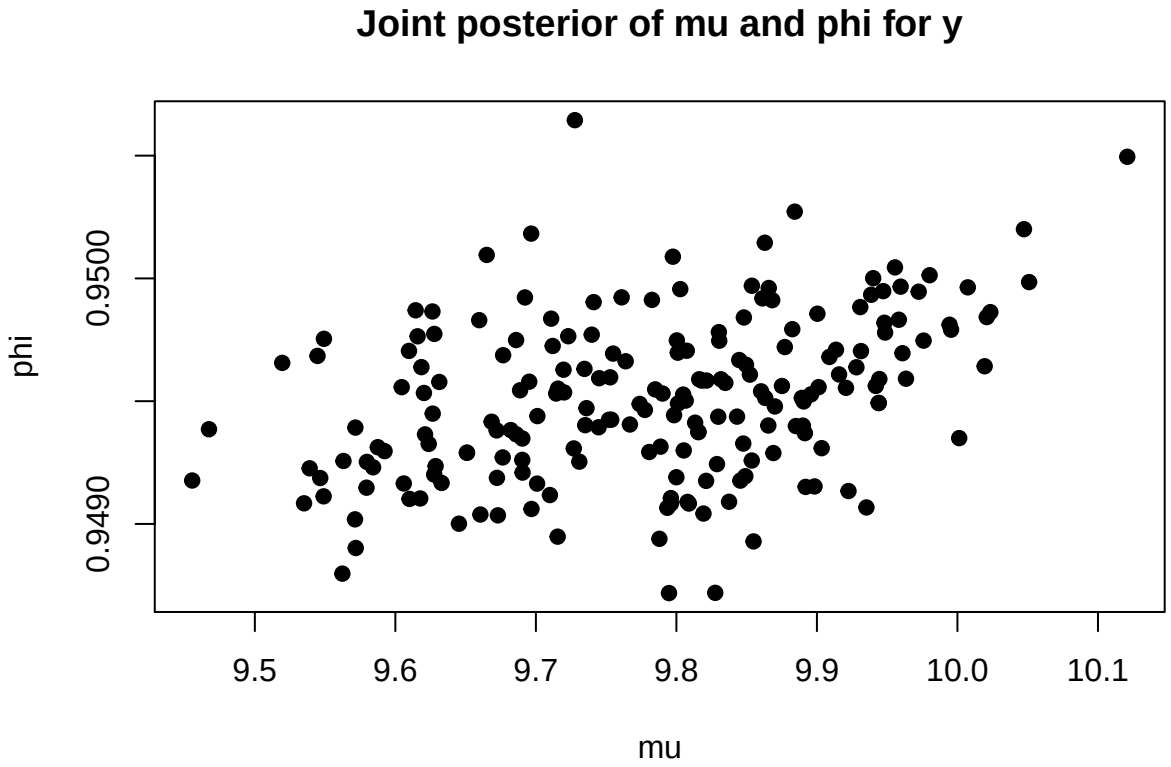


The trajectory plots of μ, σ^2 and ϕ seem to have some correlation between the iterations, however they do not have a tendency to get stuck at certain values for a number of iterations.

```
plot(mu_post2, phi_post2, pch=19, xlab="mu", ylab="phi", main="Joint posterior of mu and phi for x")
```




```
plot(mu_post, phi_post, pch=19, xlab="mu", ylab="phi", main="Joint posterior of mu and phi for y")
```



From the plots above, we can see that the joint posterior with data $x_{1:T}$ is highly correlated. The joint posterior with data $y_{1:T}$ also has some trend in it, but at a small scale.

c)

The number of infections c_t at each time point follows an independent Poisson distribution, when conditioned on a latent AR(1)-process x_t:

$$c_t|x_t \sim Poisson(\exp(x_t))$$

We are interested in estimating the latent intensity given by $\theta_t = \exp(x_t)$ over time.

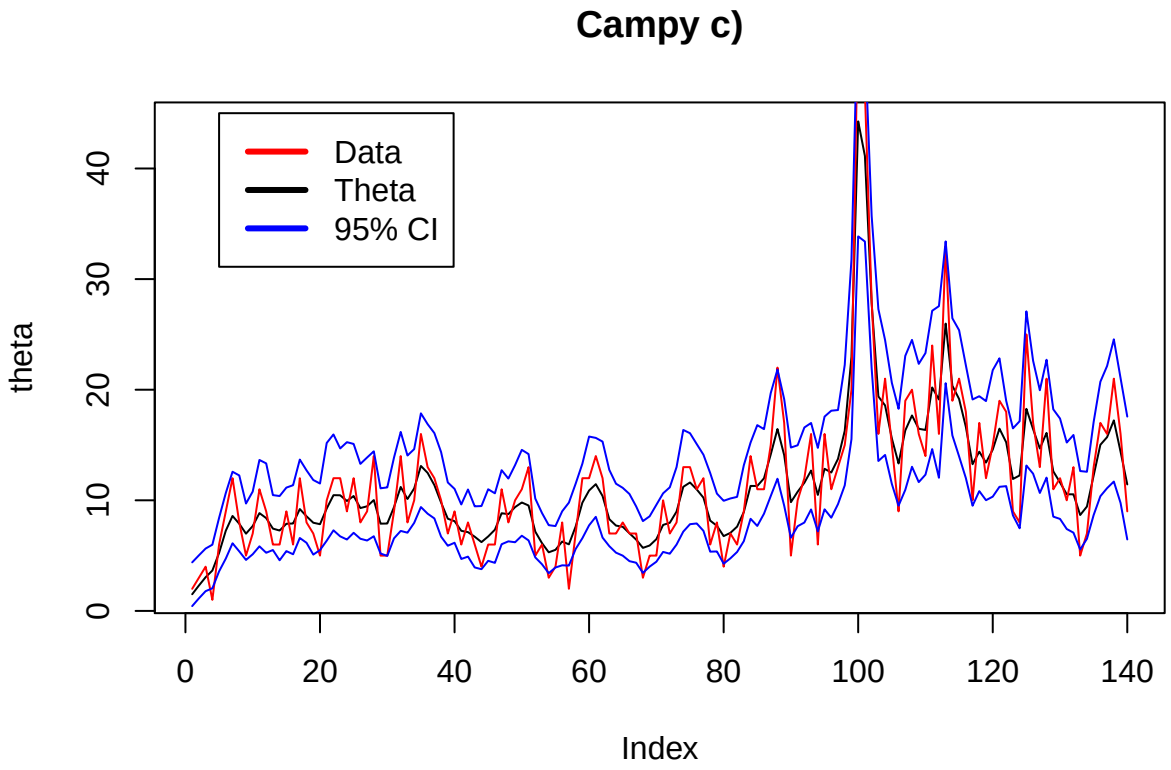
The non-informative priors used were:
$\mu \sim N(0, 10)$
$\sigma \sim Exp(1)$
$\phi \sim N(0, 0.5)$
The priors were chosen so they would cover a wide range of values as well as the values that were used in generating AR(1)-process $x_{1:T}$ data.

```
campy<-read.table("Campy.dat",header=TRUE)

campy_model = '
data {
  int<lower=0> iter;
  int c[iter];
  real<lower=1> lambda;
}
parameters {
  real mu;
  real<lower=0> sigma;
  real<lower=-1, upper=1> phi;
  real x[iter];
}

model {
  phi ~ normal(0,0.5);
  sigma ~ exponential(1);
  mu ~ normal(0,10);
  for(i in 2:iter){
    x[i] ~ normal(mu + phi*(x[i-1]-mu), sigma/lambda);
    c[i] ~ poisson(exp(x[i]));
  }

}'

c_fit <- stan(model_code=campy_model,
                     data=list(iter=140,c=campy$c, lambda=1),
                     warmup=30,
                     iter=140,
                     chains = 1)
```

```
## In file included from /home/milpo192/R/x86_64-pc-linux-gnu-library/3.4/BH/include/boost/config.hpp:3
##                  from /home/milpo192/R/x86_64-pc-linux-gnu-library/3.4/BH/include/boost/math/tools/c
##                  from /home/milpo192/R/x86_64-pc-linux-gnu-library/3.4/StanHeaders/include/stan/math
##                  from /home/milpo192/R/x86_64-pc-linux-gnu-library/3.4/StanHeaders/include/stan/math
##                  from /home/milpo192/R/x86_64-pc-linux-gnu-library/3.4/StanHeaders/include/stan/math
##                  from /home/milpo192/R/x86_64-pc-linux-gnu-library/3.4/StanHeaders/include/stan/math
##                  from /home/milpo192/R/x86_64-pc-linux-gnu-library/3.4/StanHeaders/include/stan/math
```

```
##                  from /home/milpo192/R/x86_64-pc-linux-gnu-library/3.4/StanHeaders/include/src/stan/
##                  from file44311c5c6677.cpp:8:
## /home/milpo192/R/x86_64-pc-linux-gnu-library/3.4/BH/include/boost/config/compiler/gcc.hpp:186:0: war
##  #  define BOOST_NO_CXX11_RVALUE_REFERENCES
##  ^
## <command-line>:0:0: note: this is the location of the previous definition
##
## SAMPLING FOR MODEL '1ea62bc753579321c8cc4313ccbd9bd6' NOW (CHAIN 1).
##
## Gradient evaluation took 6.2e-05 seconds
## 1000 transitions using 10 leapfrog steps per transition would take 0.62 seconds.
## Adjust your expectations accordingly!
##
##
## WARNING: There aren't enough warmup iterations to fit the
##          three stages of adaptation as currently configured.
##          Reducing each adaptation stage to 15%/75%/10% of
##          the given number of warmup iterations:
##            init_buffer = 4
##            adapt_window = 23
##            term_buffer = 3
##
## Iteration:   1 / 140 [  0%]  (Warmup)
## Iteration:  14 / 140 [ 10%]  (Warmup)
## Iteration:  28 / 140 [ 20%]  (Warmup)
## Iteration:  31 / 140 [ 22%]  (Sampling)
## Iteration:  44 / 140 [ 31%]  (Sampling)
## Iteration:  58 / 140 [ 41%]  (Sampling)
## Iteration:  72 / 140 [ 51%]  (Sampling)
## Iteration:  86 / 140 [ 61%]  (Sampling)
## Iteration: 100 / 140 [ 71%]  (Sampling)
## Iteration: 114 / 140 [ 81%]  (Sampling)
## Iteration: 128 / 140 [ 91%]  (Sampling)
## Iteration: 140 / 140 [100%]  (Sampling)
##
##  Elapsed Time: 0.02853 seconds (Warm-up)
##                0.097127 seconds (Sampling)
##                0.125657 seconds (Total)
```

```r
#print(c_fit)
theta <- exp(get_posterior_mean(c_fit))[4:143]
perc1 <- exp(sapply(as.data.frame(c_fit)[,4:143], FUN=quantile, probs=0.025))
perc2 <- exp(sapply(as.data.frame(c_fit)[,4:143], FUN=quantile, probs=0.975))
plot(theta, type="l", main="Campy c)")
lines(campy$c, col="red")
lines(perc1, col="blue")
lines(perc2, col="blue")
legend(x = 5, y=45, c("Data", "Theta", "95% CI"), col=c("red", "black", "blue"), lwd = 3)
```



The posterior mean of θ_t follows the general trend in the Campy data quite well. 95% credible interval seems to include majority of the data points.

d)

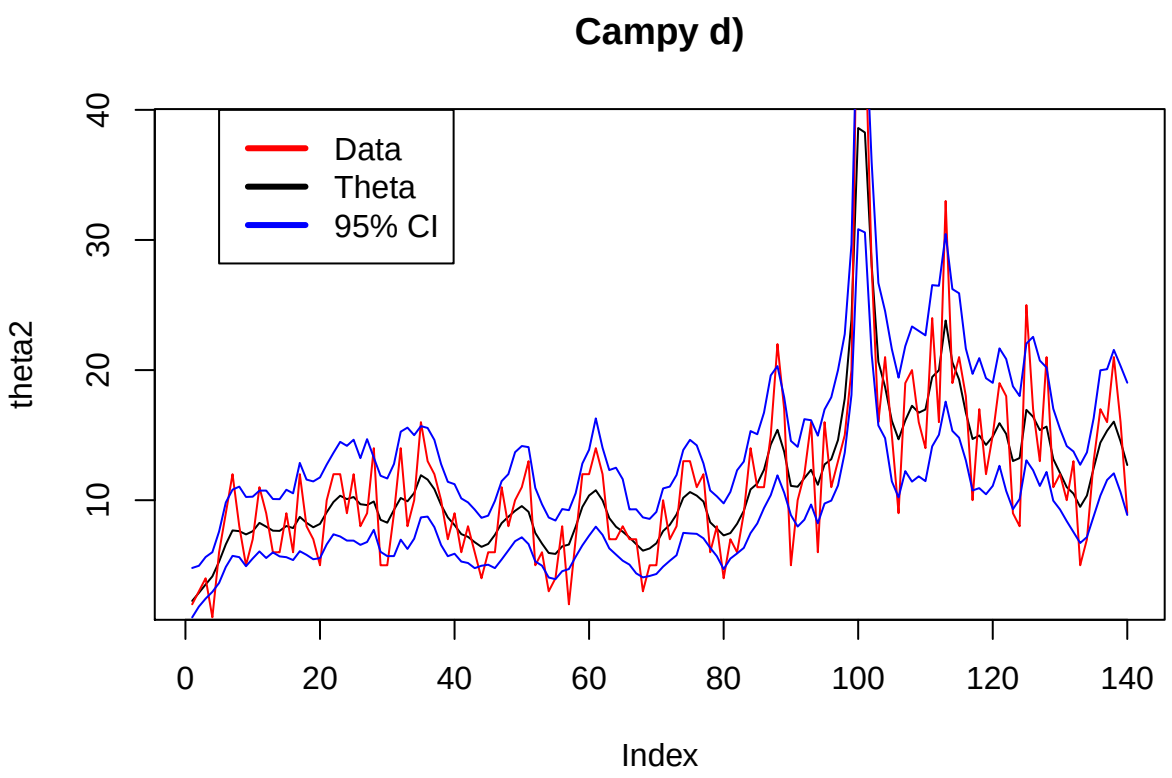
Now we assume we have a belief that the prior of σ^2 should be smoother. Therefore, we use a shrinkage constants λ to reduce ϵ_t value for each given time.

```
d_fit <- stan(model_code=campy_model,
              data=list(iter=140,c=campy$c, lambda=100),
              warmup=30,
              iter=140,
              chains = 1)
```

```
##
## SAMPLING FOR MODEL '1ea62bc753579321c8cc4313ccbd9bd6' NOW (CHAIN 1).
##
## Gradient evaluation took 6.7e-05 seconds
## 1000 transitions using 10 leapfrog steps per transition would take 0.67 seconds.
## Adjust your expectations accordingly!
##
##
## WARNING: There aren't enough warmup iterations to fit the
##          three stages of adaptation as currently configured.
##          Reducing each adaptation stage to 15%/75%/10% of
##          the given number of warmup iterations:
##            init_buffer = 4
##            adapt_window = 23
##            term_buffer = 3
##
```

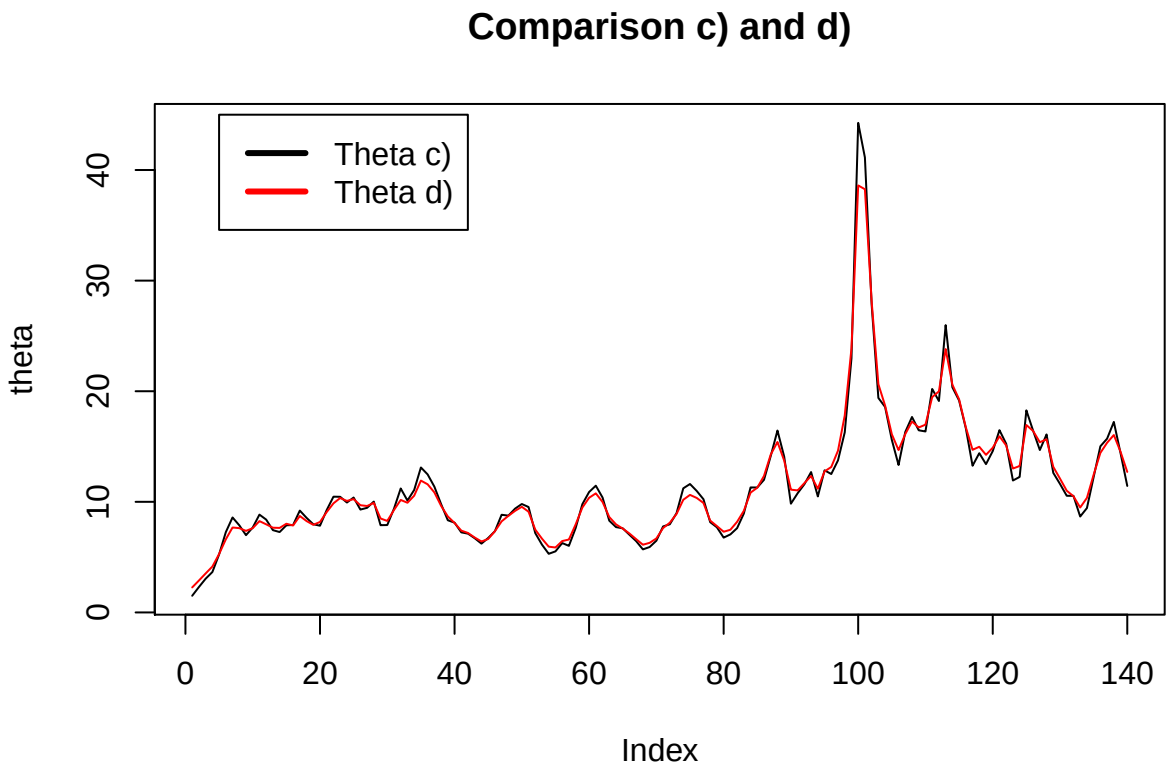
```
## Iteration:   1 / 140 [  0%]  (Warmup)
## Iteration:  14 / 140 [ 10%]  (Warmup)
## Iteration:  28 / 140 [ 20%]  (Warmup)
## Iteration:  31 / 140 [ 22%]  (Sampling)
## Iteration:  44 / 140 [ 31%]  (Sampling)
## Iteration:  58 / 140 [ 41%]  (Sampling)
## Iteration:  72 / 140 [ 51%]  (Sampling)
## Iteration:  86 / 140 [ 61%]  (Sampling)
## Iteration: 100 / 140 [ 71%]  (Sampling)
## Iteration: 114 / 140 [ 81%]  (Sampling)
## Iteration: 128 / 140 [ 91%]  (Sampling)
## Iteration: 140 / 140 [100%]  (Sampling)
##
##  Elapsed Time: 0.049873 seconds (Warm-up)
##                0.157989 seconds (Sampling)
##                0.207862 seconds (Total)
```

```r
#print(d_fit)
theta2 <- exp(get_posterior_mean(d_fit))[4:143]
perc1d <- exp(sapply(as.data.frame(d_fit)[,4:143], FUN=quantile, probs=0.025))
perc2d <- exp(sapply(as.data.frame(d_fit)[,4:143], FUN=quantile, probs=0.975))
plot(theta2, type="l", main="Campy d)")
lines(campy$c, col="red")
lines(perc1d, col="blue")
lines(perc2d, col="blue")
legend(x = 5, y=40, c("Data", "Theta", "95% CI"), col=c("red", "black", "blue"), lwd = 3)
```




```r
#Comparison:
plot(theta, type="l", main="Comparison c) and d)")
lines(theta2, col="red")
```

```
legend(x = 5, y=45, c("Theta c)", "Theta d)"), col=c("black", "red"), lwd = 3)
```



The posterior mean of θ_t in part d) does seem to be a bit smoother with shrinkage constant $\lambda = 100$, but the overall performance is a bit worse. That is, the 95% credible interval includes less values of the Campy data in comparison to part c).

Appendix

```
gibbs_sampler <- function(iter, data, mu0, tau20, nu0, sigma20){
  N <- nrow(data)
  x_bar <- mean(data[,1])
  nun <- nu0+N
  mu <- rnorm(1,mu0, sqrt(tau20))
  sigma2 <- nu0*sigma20/rchisq(1,nu0)
  results <- matrix(ncol=2,nrow=iter+1)
  results[1,1]<-mu
  results[1,2]<-sigma2
  colnames(results)<-c("mu", "sigma2")

  for(i in 1:iter){
    w <- (N/results[i,2])/((N/results[i,2])+(1/tau20))
    mun <- w*x_bar+(1-w)*mu0
    tau2n <- 1/((N/results[i,2])+(1/tau20))
    mu <- rnorm(1,mun, sqrt(tau2n))
    para_n <- (nu0*sigma20+sum((data$V1-mu)^2))/nun
    sigma2 <- nun*para_n/rchisq(1,nun)
    results[i+1,]<-c(mu, sigma2)
```

```
  }
  results
}
rain<-read.table("Rainfall.dat",header=FALSE)

# Weakly informative priors based on our guesses about the possible paramter values
mu0 <-0
tau20 <-50
nu0 <-5
sigma20 <-20

burn_in <- 100
n <- 1000
rain_gibbs <- gibbs_sampler(n, rain, mu0, tau20, nu0, sigma20)
rain_gibbs <- as.data.frame(rain_gibbs)

post_mu <- mean(rain_gibbs$mu[burn_in:n])
post_sigma2 <- mean(rain_gibbs$sigma2[burn_in:n])
plot(rain_gibbs$sigma2[burn_in:n], xaxt="n",type="l", xlab="Iteration", ylab="sigma2", main ="Gibbs samp
axis(1, at=seq(0, (n-burn_in), by=50), labels=seq(burn_in, n, by=50))
plot(rain_gibbs$mu[burn_in:n], xaxt="n", type="l", xlab="Iteration", ylab="mu", main ="Gibbs sampling o
axis(1, at=seq(0,(n-burn_in), by=50), labels=seq(burn_in,n, by=50))

rawData <- rain
x <- as.matrix(rawData)

# Model options
nComp <- 2    # Number of mixture components

# Prior options
alpha <- 10*rep(1,nComp) # Dirichlet(alpha)
muPrior <- rep(0,nComp) # Prior mean of mu
tau2Prior <- rep(10,nComp) # Prior std of mu
sigma2_0 <- rep(var(x),nComp) # s20 (best guess of sigma2)
nu0 <- rep(4,nComp) # degrees of freedom for prior on sigma2

# MCMC options
nIter <- 1000 # Number of Gibbs sampling draws

# Plotting options
plotFit <- TRUE
# lineColors <- c("blue", "green", "magenta", 'yellow')
# sleepTime <- 0.1 # Adding sleep time between iterations for plotting
################   END USER INPUT ###############

###### Defining a function that simulates from the
rScaledInvChi2 <- function(n, df, scale){
  return((df*scale)/rchisq(n,df=df))
}

####### Defining a function that simulates from a Dirichlet distribution
rDirichlet <- function(param){
  nCat <- length(param)
```

```
  piDraws <- matrix(NA,nCat,1)
  for (j in 1:nCat){
    piDraws[j] <- rgamma(1,param[j],1)
  }
  piDraws = piDraws/sum(piDraws) # Diving every column of piDraws by the sum of the elements in that co
  return(piDraws)
}

# Simple function that converts between two different representations of the mixture allocation
S2alloc <- function(S){
  n <- dim(S)[1]
  alloc <- rep(0,n)
  for (i in 1:n){
    alloc[i] <- which(S[i,] == 1)
  }
  return(alloc)
}

# Initial value for the MCMC
nObs <- length(x)
S <- t(rmultinom(nObs, size = 1 , prob = rep(1/nComp,nComp))) # nObs-by-nComp matrix with component all
mu <- matrix(ncol=nComp, nrow=nIter+1)
mu[1,] <- quantile(x, probs = seq(0,1,length = nComp))
sigma2 <- matrix(ncol=nComp, nrow=nIter+1)
sigma2[1,] <- rep(var(x),nComp)
probObsInComp <- rep(NA, nComp)

# Setting up the plot
xGrid <- seq(min(x)-1*apply(x,2,sd),max(x)+1*apply(x,2,sd),length = 100)
xGridMin <- min(xGrid)
xGridMax <- max(xGrid)
mixDensMean <- rep(0,length(xGrid))
effIterCount <- 0
hist_x <- max(invisible(hist(x)$density))
ylim <- c(0,2*hist_x)


for (k in 1:nIter){
  #message(paste('Iteration number:',k))
  alloc <- S2alloc(S) # Just a function that converts between different representations of the group al
  nAlloc <- colSums(S)
  #print(nAlloc)
  # Update components probabilities
  pi <- rDirichlet(alpha + nAlloc)

  # Update mu's
  for (j in 1:nComp){
    precPrior <- 1/tau2Prior[j]
    precData <- nAlloc[j]/sigma2[k,j]
    precPost <- precPrior + precData
    wPrior <- precPrior/precPost
    muPost <- wPrior*muPrior + (1-wPrior)*mean(x[alloc == j])
    tau2Post <- 1/precPost
```

```
    mu[k+1,j] <- rnorm(1, mean = muPost, sd = sqrt(tau2Post))
  }

  # Update sigma2's
  for (j in 1:nComp){
    sigma2[k+1,j] <- rScaledInvChi2(1, df = nu0[j] + nAlloc[j], scale = (nu0[j]*sigma2_0[j] + sum((x[al
  }

  # Update allocation
  for (i in 1:nObs){
    for (j in 1:nComp){
      probObsInComp[j] <- pi[j]*dnorm(x[i], mean = mu[k+1,j], sd = sqrt(sigma2[k+1,j]))
    }
    S[i,] <- t(rmultinom(1, size = 1 , prob = probObsInComp/sum(probObsInComp)))
  }

  # Printing the fitted density against data histogram
   if (plotFit && (k%%1 ==0)){
     effIterCount <- effIterCount + 1
  #   hist(x, breaks = 20, freq = FALSE, xlim = c(xGridMin,xGridMax), main = paste("Iteration number",k
     mixDens <- rep(0,length(xGrid))
  #   components <- c()
  for (j in 1:nComp){
       compDens <- dnorm(xGrid,mu[k+1,j],sd = sqrt(sigma2[k+1,j]))
       mixDens <- mixDens + pi[j]*compDens
  #     lines(xGrid, compDens, type = "l", lwd = 2, col = lineColors[j])
  #     components[j] <- paste("Component ",j)
  #   }
  mixDensMean <- ((effIterCount-1)*mixDensMean + mixDens)/effIterCount
  #
  #   lines(xGrid, mixDens, type = "l", lty = 2, lwd = 3, col = 'red')
  #   legend("topleft", box.lty = 1, legend = c("Data histogram",components, 'Mixture'),
  #          col = c("black",lineColors[1:nComp], 'red'), lwd = 2)
  #   Sys.sleep(sleepTime)
   }

   }

}
for(i in 1:nComp){
  plot(mu[50:nIter,i], xlab="Iteration", ylab="mu", main=paste0("mu ", i), type="l")
}

for(i in 1:nComp){
  plot(sigma2[50:nIter,i], xlab="Iteration", ylab="sigma2", main=paste0("sigma2 ", i), type="l")
}
hist(x, breaks = 20, freq = FALSE, xlim = c(xGridMin,xGridMax), main = "Final fitted density")
lines(xGrid, mixDensMean, type = "l", lwd = 2, lty = 4, col = "red")
lines(xGrid, dnorm(xGrid, mean = post_mu, sd = sqrt(post_sigma2)), type = "l", lwd = 2, col = "blue")
legend("topright", box.lty = 1, legend = c("Data histogram","Mixture density","Normal density from a)")

#Question 2
```

```
library(rstan)

#a)

ar_process = '
data {
  int<lower=0> iter;
  real mu;
  real<lower=0> sigma2;
  real<lower=-1, upper=1> phi;
}
parameters {
  real x[iter];
}
model {
  for(i in 2:iter){
    x[i] ~ normal(mu + phi*(x[i-1]-mu), sqrt(sigma2));
  }
}'

burnin <- 101
niter <- 300
mu <- 10
init_para <- list(list(x=c(mu,rep(0,niter-1))))
#needs to be a list of a list since each chain needs its own list
fit1<-stan(model_code=ar_process,
           data=list(iter=niter,sigma2=2, phi=0.5, mu=10),
           init = init_para,
           warmup=burnin,
           iter=niter,
           chains = 1,
           control = list(max_treedepth = 15))
fit1_postmean <- get_posterior_mean(fit1)[burnin:niter]
plot(fit1_postmean, type="l", xaxt="n", xlab="Draws", ylab="Posterior mean", main="Posterior mean, phi=
axis(1, at=seq(0,(niter-burnin), by=10), labels=seq(burnin,niter, by=10))

fit2<-stan(model_code=ar_process,
           data=list(iter=niter,sigma2=2, phi=0.9, mu=10),
           init = init_para,
           warmup=burnin,
           iter=niter,
           chains = 1,
           control = list(max_treedepth = 15))
#print(fit2)
fit2_postmean <- get_posterior_mean(fit2)[burnin:niter]
plot(fit2_postmean, type="l", xaxt="n", xlab="Draws", ylab="Posterior mean", main="Posterior mean, phi=
axis(1, at=seq(0,(niter-burnin), by=10), labels=seq(burnin,niter, by=10))

fit3<-stan(model_code=ar_process,
           data=list(iter=niter,sigma2=2, phi=-0.9, mu=10),
           init = init_para,
           warmup=burnin,
           iter=niter,
```

```
            chains = 1,
            control = list(max_treedepth = 15))

#print(fit3)
fit3_postmean <- get_posterior_mean(fit3)[burnin:niter]
plot(fit3_postmean, type="l", xaxt="n", xlab="Draws", ylab="Posterior mean", main="Posterior mean, phi=
axis(1, at=seq(0,(niter-burnin), by=10), labels=seq(burnin,niter, by=10))

fit4<-stan(model_code=ar_process,
            data=list(iter=niter,sigma2=2, phi=0, mu=10),
            init = init_para,
            warmup=burnin,
            iter=niter,
            chains = 1,
            control = list(max_treedepth = 15))
#print(fit4)
fit4_postmean <- get_posterior_mean(fit4)[burnin:niter]
plot(fit4_postmean, type="l", xaxt="n", xlab="Draws", ylab="Posterior mean", main="Posterior mean, phi=
axis(1, at=seq(0,(niter-burnin), by=10), labels=seq(burnin,niter, by=10))
#b)

burnin <- 101
niter <- 300
mu <- 10
init_para <- list(list(x=c(mu,rep(0,niter-1))))

xT <- stan(model_code=ar_process,
            data=list(iter=niter,sigma2=2, phi=0.3, mu=10),
            init = init_para,
            warmup=burnin,
            iter=niter,
            chains = 1,
            control = list(max_treedepth = 15))
xT_postmean <- get_posterior_mean(xT)[burnin:niter]

yT <-  stan(model_code=ar_process,
             data=list(iter=niter,sigma2=2, phi=0.95, mu=10),
             init = init_para,
             warmup=burnin,
             iter=niter,
             chains = 1,
             control = list(max_treedepth = 15))
yT_postmean <- get_posterior_mean(yT)[burnin:niter]

ar_mcmc = '
data {
  int<lower=0> iter;
  vector[iter] x;
}
parameters {
  real mu;
  real<lower=0> sigma;
  real<lower=-1, upper=1> phi;
```

```
}
model {
// Priors
  mu ~ normal(0,10);
  sigma ~ exponential(0.1); //very weak prior for sigma, but always postive
  phi ~ normal(0, 1);
// model
  for(i in 2:iter){
    x[i] ~ normal(mu + phi*(x[i-1]-mu), sigma);
  }
}'


burnin <- 100
niter <- 300
ndraws <- 200

xT_fit <- stan(model_code=ar_mcmc,
           data=list(iter=ndraws,x=xT_postmean),
           warmup=burnin,
           iter=niter,
           chains = 1,
           control = list(max_treedepth = 15,adapt_delta = 0.99))

#pairs(xT_fit)

para_postmean <- get_posterior_mean(xT_fit)[1:3]
mu_post2 <- extract(xT_fit)$mu
phi_post2 <- extract(xT_fit)$phi
sigma2_post2 <- (extract(xT_fit)$sigma)^2

perc1x <- sapply(as.data.frame(xT_fit), FUN=quantile, probs=0.025)[1:3]
perc2x <- sapply(as.data.frame(xT_fit), FUN=quantile, probs=0.975)[1:3]
n_eff_x <- summary(xT_fit)$summary[,"n_eff"][1:3]
cat("Posterior means for mu, sigma2, phi: ", c(mean(mu_post2), mean(sigma2_post2), mean(phi_post2)))
cat("\n Upper limits for 95% credible interval: ", perc2x)
cat("\n Lower limits for 95% credible interval: ", perc1x)
cat("\n Number of effective posterior samples: ", n_eff_x)

yT_fit <- stan(model_code=ar_mcmc,
               data=list(iter=ndraws,x=yT_postmean),
               warmup=burnin,
               iter=niter,
               chains = 1)

Ypara_postmean <- get_posterior_mean(yT_fit)[1:3]

mu_post <- extract(yT_fit)$mu
phi_post <- extract(yT_fit)$phi
sigma2_post <- (extract(yT_fit)$sigma)^2

perc1y <- sapply(as.data.frame(yT_fit), FUN=quantile, probs=0.025)[1:3]
perc2y <- sapply(as.data.frame(yT_fit), FUN=quantile, probs=0.975)[1:3]
```

```
n_eff_y<- summary(yT_fit)$summary[,"n_eff"][1:3]
cat("Posterior means for mu, sigma2, phi: ", c(mean(mu_post), mean(sigma2_post), mean(phi_post)))
cat("\n Upper limits for 95% credible interval: ", perc2y)
cat("\n Lower limits for 95% credible interval: ", perc1y)
cat("\n Number of effective posterior samples: ", n_eff_y)
# Trajectory plots of the mu
plot(mu_post2, type='l')

plot(mu_post, type='l')

# Trajectory plots of the sigma2

plot(sigma2_post2, type='l')

plot(sigma2_post, type='l')

# Trajectory plots of the phi

plot(phi_post2, type='l')

plot(phi_post, type='l')
plot(mu_post2, phi_post2, pch=19, xlab="mu", ylab="phi", main="Joint posterior of mu and phi for x")

plot(mu_post, phi_post, pch=19, xlab="mu", ylab="phi", main="Joint posterior of mu and phi for y")
campy<-read.table("Campy.dat",header=TRUE)

campy_model = '
data {
  int<lower=0> iter;
  int c[iter];
  real<lower=1> lambda;
}
parameters {
  real mu;
  real<lower=0> sigma;
  real<lower=-1, upper=1> phi;
  real x[iter];
}

model {
  phi ~ normal(0,0.5);
  sigma ~ exponential(1);
  mu ~ normal(0,10);
  for(i in 2:iter){
    x[i] ~ normal(mu + phi*(x[i-1]-mu), sigma/lambda);
    c[i] ~ poisson(exp(x[i]));
  }

}'

c_fit <- stan(model_code=campy_model,
                     data=list(iter=140,c=campy$c, lambda=1),
                     warmup=30,
```

```
                    iter=140,
                    chains = 1)

#print(c_fit)
theta <- exp(get_posterior_mean(c_fit))[4:143]
perc1 <- exp(sapply(as.data.frame(c_fit)[,4:143], FUN=quantile, probs=0.025))
perc2 <- exp(sapply(as.data.frame(c_fit)[,4:143], FUN=quantile, probs=0.975))
plot(theta, type="l", main="Campy c)")
lines(campy$c, col="red")
lines(perc1, col="blue")
lines(perc2, col="blue")
legend(x = 5, y=45, c("Data", "Theta", "95% CI"), col=c("red", "black", "blue"), lwd = 3)

d_fit <- stan(model_code=campy_model,
              data=list(iter=140,c=campy$c, lambda=100),
              warmup=30,
              iter=140,
              chains = 1)

#print(d_fit)
theta2 <- exp(get_posterior_mean(d_fit))[4:143]
perc1d <- exp(sapply(as.data.frame(d_fit)[,4:143], FUN=quantile, probs=0.025))
perc2d <- exp(sapply(as.data.frame(d_fit)[,4:143], FUN=quantile, probs=0.975))
plot(theta2, type="l", main="Campy d)")
lines(campy$c, col="red")
lines(perc1d, col="blue")
lines(perc2d, col="blue")
legend(x = 5, y=40, c("Data", "Theta", "95% CI"), col=c("red", "black", "blue"), lwd = 3)

#Comparison:
plot(theta, type="l", main="Comparison c) and d)")
lines(theta2, col="red")
legend(x = 5, y=45, c("Theta c)", "Theta d)"), col=c("black", "red"), lwd = 3)
```