STA421 Foundations of Bayesian Methodology4

Group Worksheet 04

Jerome Sepin

Exercise 2: (Gibbs sampler)

Use the file 04GibbsSampler.R provided on OLAT. Let $y_1, ...y_n \sim \mathcal{N}(\mu, \sigma^2)$, where $\mu \sim \mathcal{N}(\mu_0, \sigma_0^2)$ and $1/\sigma^2 \sim G(a_0, b_0)$

(a) Complete the derivations from the lecture by proving that if $p(x) = exp(-0.5(ax^2 - 2bx))$ then $X \sim \mathcal{N}(\lfloor/\dashv, \infty/\dashv)$

$$
\begin{aligned}
p(x) &= exp(-0.5(ax^2 - 2bx)) \\
&= exp\left(-\frac{1}{2}\left(\frac{x^2 - b/a}{1/a}\right)^2\right) \cdot exp\left(\frac{1}{2} \cdot \frac{b^2/a^2}{1/a}\right) \\
&= exp\left(-\frac{1}{2}\left(\frac{x^2 - b/a}{1/a}\right)^2\right) \cdot exp\left(\frac{1}{2} \cdot \frac{b^2}{a}\right)
\end{aligned}
$$

Normalizing this function should lead to a density function:

$$
\begin{aligned}
p'(x) &= \frac{p(x)}{\int_\infty^\infty p(x)dx} \\
&= \frac{exp\left(-\frac{1}{2}\left(\frac{x^2 - b/a}{1/a}\right)^2\right) \cdot exp\left(\frac{1}{2} \cdot \frac{b^2}{a}\right)}{exp\left(\frac{1}{2} \cdot \frac{b^2}{a}\right) \int_\infty^\infty exp\left(-\frac{1}{2}\left(\frac{x^2 - b/a}{1/a}\right)^2\right) dx} \\
&= \frac{\frac{1}{1/a\sqrt{2\pi}} exp\left(-\frac{1}{2}\left(\frac{x^2 - b/a}{1/a}\right)^2\right)}{\int_\infty^\infty \frac{1}{1/a\sqrt{2\pi}} exp\left(-\frac{1}{2}\left(\frac{x^2 - b/a}{1/a}\right)^2\right) dx} \\
&= \frac{1}{1/a\sqrt{2\pi}} exp\left(-\frac{1}{2}\left(\frac{x^2 - b/a}{1/a}\right)^2\right)
\end{aligned}
$$

This means:

$X \sim \mathcal{N}(\lfloor/\dashv, \infty/\dashv)$

(b) Complete the following tasks:

1 Generate data: a random normal sample by assuming set.seed(44566),$n = 30, \mu = 4$ and $\sigma^2 = 16$.

```r
# generate data
set.seed(44566)
mu <- 4
sigma2 <- 16
n <- 30
y <- rnorm(n=n, mean=mu, sd=sqrt(sigma2))
```

2 Set the parameters of the prior distributions to $\mu_0 = -3, \sigma_0^2 = 4, a_0 = 1.6$ (shape) and $b_0 = 0.4$ (rate), take a burn-in of 4000 simulations and run your code for a total of 10000 simulations when assuming your own initial values. Provide traceplots and summaries (mean, sd, 0.025, 0.5, 0.975 quantiles) of the marginal posteriors for μ, σ^2 and $1/\sigma^2$.

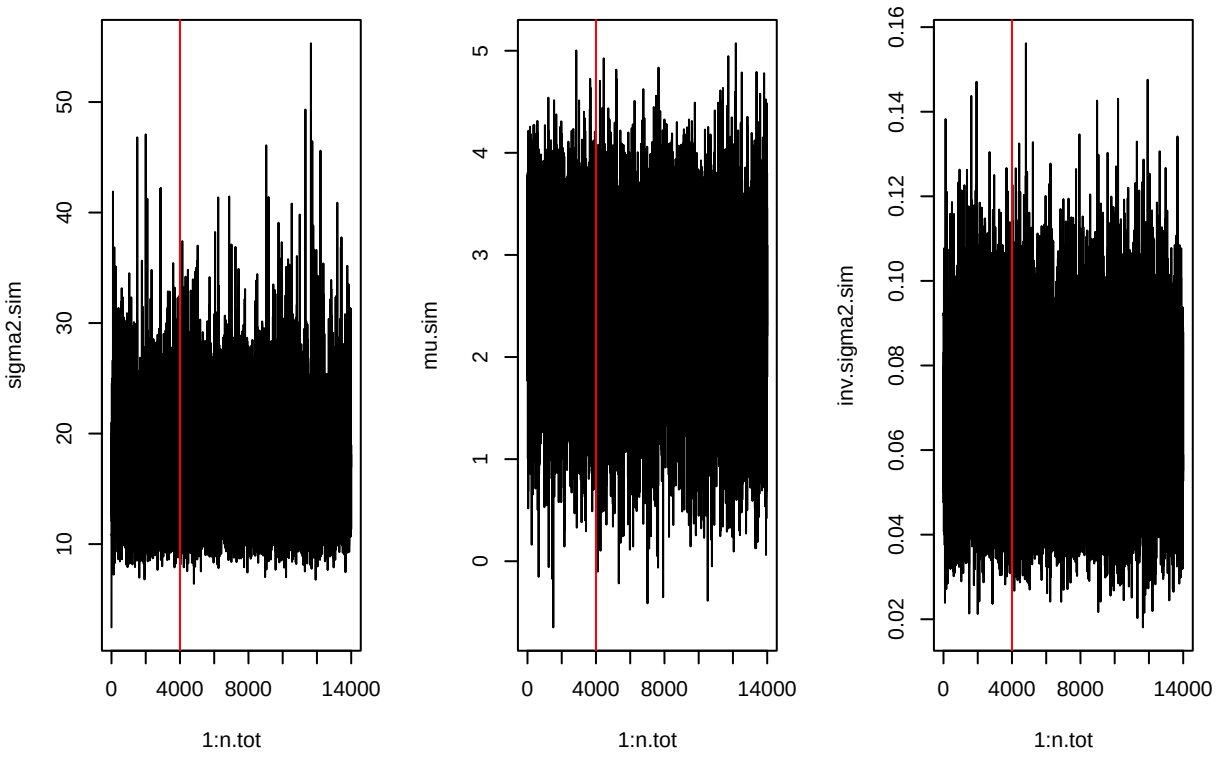
```r
# Define the parameters of the prior distributions
mu0 <- -3
sigma2_0 <- 4
a0 <- 1.6
b0 <- 0.4
```

```r
####################
## Gibbs sampler (1 chain)
####################
# initialisation
set.seed(44566)

n.iter <- 10000
n.burnin <- 4000
n.thin <- 1
#n.thin <- floor((n.iter-n.burnin)/500)
n.chains <- 1
parameters <- c("mu", "sigma2", "inv_sigma2")
n.parameters <- length(parameters)

n.tot <- n.burnin + n.iter*n.thin

gibbs_samples <- matrix(NA, nrow = n.iter, ncol = n.parameters)
colnames(gibbs_samples) <- parameters

mu.sim <- rep(NA, length = n.tot)
sigma2.sim <- rep(NA, length = n.tot)
inv.sigma2.sim <- rep(NA, length = n.tot)

#Set the initial value
sigma2.sim[1] <- 1/runif(n.chains)

# set the counter
k <- 1

#Run the for loop (only one chain)
for(i in 2:(n.burnin+n.iter*n.thin)){

  mu.sim[i] <- rnorm(1,
                     mean = (sum(y)/sigma2.sim[i-1] + mu0/sigma2_0) /
                       (n/sigma2.sim[i-1] + 1/sigma2_0),
```

```
                    sd = sqrt(1/(n/sigma2.sim[i-1] + 1/sigma2_0)))

  sigma2.sim[i] <- 1/rgamma(1, shape = n/2 + a0,
                            scale = 1 / (sum((y-mu.sim[i])^2)/2 + b0))

  inv.sigma2.sim[i] <- 1/sigma2.sim[i]

  # after the burnin save every n.thin'th sample
  if((i > n.burnin) && (i%%n.thin == 0)){
    gibbs_samples[k,] <- c(mu.sim[i], sigma2.sim[i], inv.sigma2.sim[i])
    k <- k + 1
  }

  if(i%%1000 == 0){
    # report on the fly in which iteration the chain is
    cat(i, "\n")
  }

}
```

```
## 1000
## 2000
## 3000
## 4000
## 5000
## 6000
## 7000
## 8000
## 9000
## 10000
## 11000
## 12000
## 13000
## 14000
```

```
# Traceplots
par(mfrow = c(1,3))
plot(y=sigma2.sim,x=1:n.tot,type = "l")
abline(v=n.burnin,col ="red")
plot(y=mu.sim,x=1:n.tot,type = "l")
abline(v=n.burnin,col ="red")
plot(y=inv.sigma2.sim,x=1:n.tot,type = "l")
abline(v=n.burnin,col ="red")
```




```
# n.iter samples after n.burnin taking every n.thin'th sample
dim(gibbs_samples)
```

```
## [1] 10000     3
```

```
mu_gibbs_samples <- gibbs_samples[,"mu"]
sigma2_gibbs_samples <- gibbs_samples[,"sigma2"]
inv_sigma2_gibbs_samples <- gibbs_samples[,"inv_sigma2"]

rbind(
"mean"=apply(X=gibbs_samples,2, mean),
"sd"=apply(X=gibbs_samples,2, sd),
apply(X=gibbs_samples,2, function(x) {quantile(x, prob = c(0.025,0.5,0.975))})
)
```

```
##              mu    sigma2 inv_sigma2
## mean  2.4479404 16.333156 0.06561631
## sd    0.7206035  4.548334 0.01695249
## 2.5%  0.9436683  9.733305 0.03673533
## 50%   2.4692436 15.566910 0.06423882
## 97.5% 3.8080215 27.221752 0.10274003
```

```
# plots (compare with INLA)
# INLA is exact
```

```
# Question: How long should we run the MCMC Gibbs chain to get close to the exact INLA approximation?
# Run for 10 min, for 30 min, for 1h...
```

3 For this model, INLA (https://www.r-inla.org/) provides exact results. Compare results provided by the Gibbs sampler with those provided by INLA.

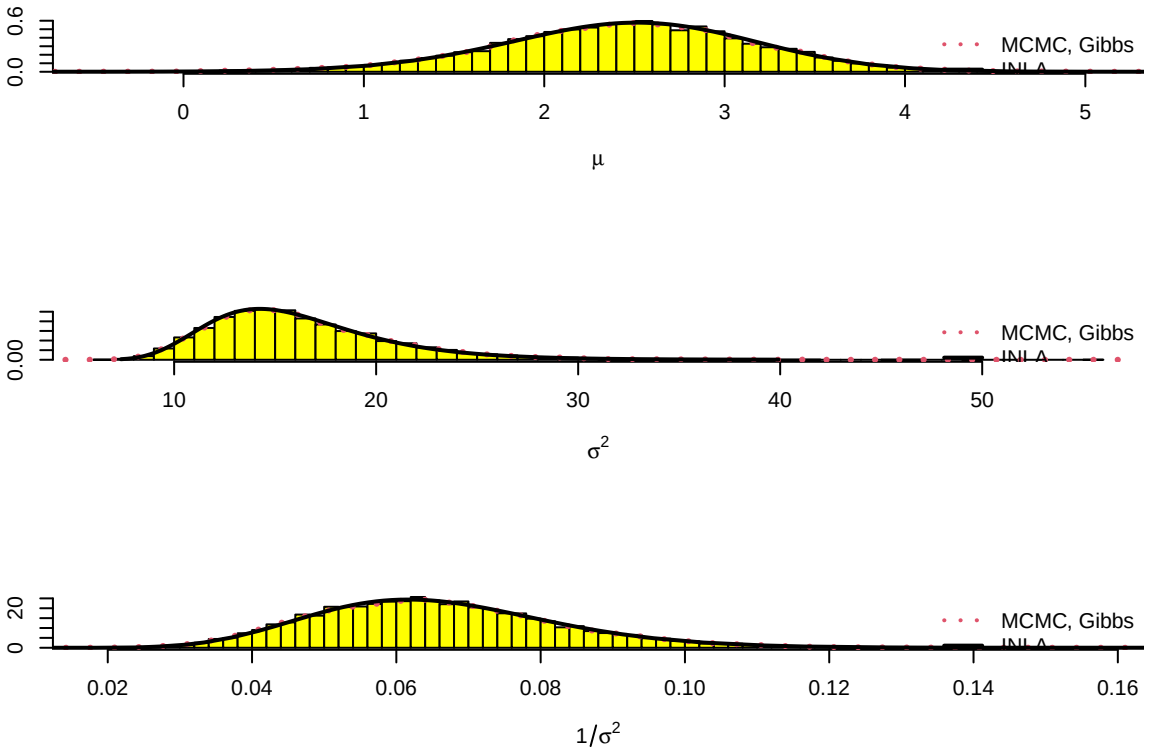
```
################################
# INLA exact result (motivation)
################################

# https://www.r-inla.org/
library(INLA)
```

```
## Loading required package: Matrix

## Loading required package: foreach

## Warning: package 'foreach' was built under R version 4.1.3

## Loading required package: parallel

## Loading required package: sp

## Warning: package 'sp' was built under R version 4.1.3

## This is INLA_22.03.16 built 2022-03-16 13:18:13 UTC.
##  - See www.r-inla.org/contact-us for how to get help.
```

```
formula <- y ~ 1
inla.output <- inla(formula,data=data.frame(y=y),
                    control.family = list(hyper =
                                            list(prec = list(prior="loggamma",param=c(a0,b0)))),
                    control.fixed = list(mean.intercept=mu0, prec.intercept=1/sigma2_0))
```

```
library(MASS)

par(mfrow=c(3,1))
# plot for mean
rg <- range(inla.output$marginals.fixed$"(Intercept)"[,2])
truehist(mu_gibbs_samples, prob=TRUE, col="yellow", xlab=expression(mu))
lines(density(mu_gibbs_samples),lty=3,lwd=3, col=2)
lines(inla.output$marginals.fixed$"(Intercept)",lwd=2)
legend("topright",c("MCMC, Gibbs","INLA"),lty=c(3,1),lwd=c(2,2),col=c(2,1),cex=1.0,bty="n")

# plot for variance
m_var <-inla.tmarginal(function(x) 1/x, inla.output$marginals.hyperpar[[1]])
truehist(sigma2_gibbs_samples, prob=TRUE, col="yellow", xlab=expression(sigma^2))
lines(density(sigma2_gibbs_samples),lty=3,lwd=3, col=2)
lines(m_var,lwd=2)
```

```
legend("topright",c("MCMC, Gibbs","INLA"),lty=c(3,1),lwd=c(2,2),col=c(2,1),cex=1.0,bty="n")

# plot for precision
truehist(inv_sigma2_gibbs_samples, prob=TRUE, col="yellow", xlab=expression(1/sigma^2))
lines(density(inv_sigma2_gibbs_samples),lty=3,lwd=3, col=2)
lines(inla.output$marginals.hyperpar[[1]],lwd=2)
legend("topright",c("MCMC, Gibbs","INLA"),lty=c(3,1),lwd=c(2,2),col=c(2,1),cex=1.0,bty="n")
```



4 Explain step by step in your own words what the code in 04GibbsSampler.R is doing.

One has information about the data in form of a likelihood, the distribution that describes the data (here normal) and an idea about the parameters of it (mu & sigma^2) in form of priors (prior for mu is again a normal, and of sigma^2 is an inverse gamma). One has then a kernel of a joint posterior distribution for the parameters but can get the kernel of the conditional distribution of only one parameter by simply treating the other one as a fixed parameter.

What we do now is simply initialzing the parameters randomly and then stepwise "update" the parameters until the parameters do not change too much anymore (more or less converge). Of course at the beginning the "convergence" is step because one comes from purely at random to a state that acutally describes the data. This phase until one gets a reasonable state is called burn in and the parameters within this phase are not taken for the posterior distribution for the parameters of interest. But after this phase the parameters that we get in each step are taken and form the posterior distribution.

Exercise 3: (Metropolis-Hastings sampler for logistic regression)

Use the file 04MHSampler.R provided on OLAT. The code from the lecture contains a random walk Metropolis(-Hastings) sampler in R with one univariate normal proposal for the intercept and an independent univariate normal proposal for the slope in the logistic regression for binomial mice data (Collett, 2003, p.71). Data in Table 1 show the number of deaths from pneumonia y in n mice exposed to various doses x of an anti-pneumococcus serum. For the Bayesian analysis two independent normal priors for intercept and slope with mean = 0 and variance = 10000 are assumed.

Task: Make sure that n.thin = 1. Tune acceptance rates by changing the spread (tuning parameters s_alpha and s_beta) of the proposal distributions. Set the tuning parameters of the proposals to low (0.01, 1), middle (1, 100) and high (50, 5000) values to obtain different acceptance rates. Generate MCMC samples under each of the above conditions. Investigate and interpret the traceplots, auto-correlations and cross-correlations.

```
# the covariate values (dose)
x_original <- c(0.0028, 0.0028, 0.0056, 0.0112, 0.0225, 0.0450)
# the centered covariate values (centered dose)
x <- x_original - mean(x_original)
# number of mice deaths
# y <- c(35, 21, 9, 6, 1)
y <- c(26, 9, 21, 9, 6, 1)
# total number of mice
# n <- c(40, 40, 40, 40, 40)
n <- c(28, 12, 40, 40, 40, 40)
print(data.frame("x"=x,"y"=y,"n"=n))
```

```
##              x  y  n
## 1 -0.012183333 26 28
## 2 -0.012183333  9 12
## 3 -0.009383333 21 40
## 4 -0.003783333  9 40
## 5  0.007516667  6 40
## 6  0.030016667  1 40
```

1 Explain step by step in your own words what the code in 04MHSampler.R is doing.

The aim of the Metropolis-Hastings algorithm is to generate a collection of states (here parameters α, β) that describe our target distribution, which is the logistic regression $f(\alpha, \beta|\boldsymbol{y}, \boldsymbol{n}, \boldsymbol{x})$. Via a Markov process, the state (the parameters) should converge to a stationary state which is the desired target distribution.

The algorithm starts with the so called condition of detailed balance and means simply that each transition, for example $\alpha \to \alpha'$, is reversible and happens with the same probability. More specifically this means the probability to be in state α and transitioning to state α' is equal to the probability to be in state α' and transitioning to state α.

$$\underbrace{P(\alpha'|\alpha)}_{\text{Transitioning to state } \alpha' \text{ given one is in } \alpha} \cdot \underbrace{f(\alpha, \beta|\boldsymbol{y}, \boldsymbol{n}, \boldsymbol{x})}_{\text{P to be in state } \alpha} = P(\alpha|\alpha') f(\alpha', \beta|\boldsymbol{y}, \boldsymbol{n}, \boldsymbol{x})$$

Which can be rewritten as:

$$\frac{P(\alpha'|\alpha)}{P(\alpha|\alpha')} = \frac{f(\alpha', \beta|\boldsymbol{y}, \boldsymbol{n}, \boldsymbol{x})}{f(\alpha, \beta|\boldsymbol{y}, \boldsymbol{n}, \boldsymbol{x})}$$

For simplicity we focus her on one transition, namely $P(\alpha'|\alpha)$ but the same procedures also hold for the back-transition $P(\alpha|\alpha')$. Now, the transition $P(\alpha'|\alpha)$ has to be split into two steps. The first one is to

"propose" the next value (α') and the second one is to accept this proposal or not. The proposal distribution $q(\alpha'|\alpha)$ is in our case here a normal distribution, so $\alpha' \sim \mathcal{N}(\alpha, \sigma_\alpha^2)$ and this means that the proposed value α' is depending on the current value α but the spread is defined by a tuning parameter σ_α^2 which kind of defines the update step. The acceptance distribution $A(\alpha', \alpha)$ does not have to be defined here as we will see. We can now rewrite:

$$\begin{aligned}\frac{A(\alpha', \alpha) \cdot q(\alpha'|\alpha)}{A(\alpha, \alpha') \cdot q(\alpha|\alpha')} &= \frac{f(\alpha', \beta|\boldsymbol{y}, \boldsymbol{n}, \boldsymbol{x})}{f(\alpha, \beta|\boldsymbol{y}, \boldsymbol{n}, \boldsymbol{x})} \\ \frac{A(\alpha', \alpha)}{A(\alpha, \alpha')} &= \frac{f(\alpha', \beta|\boldsymbol{y}, \boldsymbol{n}, \boldsymbol{x}) \cdot q(\alpha|\alpha')}{f(\alpha, \beta|\boldsymbol{y}, \boldsymbol{n}, \boldsymbol{x}) \cdot q(\alpha'|\alpha)}\end{aligned}$$

In case of the Metropolis-Hastings algorithm the acceptance ratio $\frac{A(\alpha', \alpha)}{A(\alpha, \alpha')}$ can be written as A^α and it is forced to be lower or equal to 1 (why????):

$$A^\alpha = \min\left(1, \frac{f(\alpha', \beta|\boldsymbol{y}, \boldsymbol{n}, \boldsymbol{x}) \cdot q(\alpha|\alpha')}{f(\alpha, \beta|\boldsymbol{y}, \boldsymbol{n}, \boldsymbol{x}) \cdot q(\alpha'|\alpha)}\right)$$

Now, if we get a proposed value α' that is more probable then the current value α this means that the acceptance ratio A^α is higher than 1 and we should always accept this value. On the other hand, if we get a proposed value that is less likely we should sometimes accept it. This is achieved by adding a stochastic component in form of a random sample from the uniform distribution. If the acceptance ratio is higher then the random sample we still accept this point. If it is lower we reject and stick with the current value α.

```
U <- runif(1)
ifelse(U <= A_alpha, alpha <- alpha_star, alpha <- alpha)
```

```
# Assumption
# variance of normal priors
sigma2 <- 10^(4)

#####################################
## Step 1: R: (univariate proposal) Metropolis MCMC settings
#####################################

# number of MCMC iterations
n.iter <- 10000
# burnin length
n.burnin <- 4000
# thinning parameter
n.thin <- 1
#n.thin <- floor((n.iter-n.burnin)/500)

# standard deviations for the
# normal proposal
s_alpha <- 1
s_beta <- 60

MH <- function(s_alpha, s_beta, sigma2 , n.thin = 1,
               n.burnin = 4000, n.iter=10000,show = F){
  alpha_samples <- c()
  beta_samples <- c()
  alpha_yes_history <- rep(0,n.burnin + n.iter*n.thin)
```

```r
beta_yes_history  <- rep(0,n.burnin + n.iter*n.thin)
# number of accepted proposals
alpha_yes <- 0
beta_yes <- 0

# starting values
alpha <- 0
beta <- 0

# inverse logit: logit^(-1)(alpha + beta*x)
mypi <- function(alpha, beta, x){
  tmp <- exp(alpha + beta*x)
  pi <- tmp/(1+tmp)
  return(pi)
}

# counter
count <- 0
# start the MCMC algorithm (the first iteration after the burn-in is 1)
for(i in -n.burnin:(n.iter*n.thin)){
  count <- count +1

  ## update alpha
  # generate a new proposal for alpha
  alpha_star <- rnorm(1, alpha, sd=s_alpha)

  # NOTE: it is more stable to calculate everything on the log scale
  enum  <- sum(dbinom(y, size=n, prob=mypi(alpha_star, beta, x), log=TRUE)) +
    dnorm(alpha_star, mean=0, sd=sqrt(sigma2), log=TRUE)
  denom <- sum(dbinom(y, size=n, prob=mypi(alpha     , beta, x), log=TRUE))  +
    dnorm(alpha     , mean=0, sd=sqrt(sigma2), log=TRUE)

  # log acceptance rate (since we use a random walk proposal there is no
  #   proposal ratio in the acceptance probability)
  logacc <- enum - denom
  if(log(runif(1)) <= logacc){
    # accept the proposed value
    alpha <- alpha_star
    alpha_yes <- alpha_yes + 1
  }

  ## update beta
  # generate a new proposal for beta
  beta_star <- rnorm(1, beta, sd=s_beta)

  enum <- sum(dbinom(y, size=n, prob=mypi(alpha, beta_star, x), log=TRUE)) +
    dnorm(beta_star, mean=0, sd=sqrt(sigma2), log=TRUE)
  denom<- sum(dbinom(y, size=n, prob=mypi(alpha, beta, x), log=TRUE)) +
    dnorm(beta, mean=0, sd=sqrt(sigma2), log=TRUE)
  # log acceptance rate
  logacc <- enum - denom

  if(log(runif(1)) <= logacc){
```

```
      # accept the proposed value
      beta <- beta_star
      beta_yes <- beta_yes + 1
      alpha_yes_history[count] <- 1
      beta_yes_history[count]  <- 1
    }

    # after the burnin save every kth sample
    if((i > 0) && (i%%n.thin == 0)){
      alpha_samples <- c(alpha_samples, alpha)
      beta_samples <- c(beta_samples, beta)
    }
    if(show){
      if(i%%1000 ==0){
        # print the acceptance rates on the fly
        print(cbind("i"=as.integer(i), "acc.rate alpha"=alpha_yes/count,
                    "acc.rate beta"=beta_yes/count))
        #cat(c(i, alpha_yes/count, beta_yes/count), "\n")
      }
    }
  }
  #output
  output <- list("alpha_samples"=alpha_samples, "beta_samples"=beta_samples,
                 "alpha_yes"=alpha_yes, "beta_yes"=beta_yes,
                 "alpha_yes_history"=alpha_yes_history,
                 "beta_yes_history"=beta_yes_history
                 )
  return(output)
}
```

```
set.seed(44566)
tuning_low    <- MH(s_alpha=0.01, s_beta=1, sigma2 , n.thin = 1,
                    n.burnin = 4000, n.iter=10000, show = T)
```

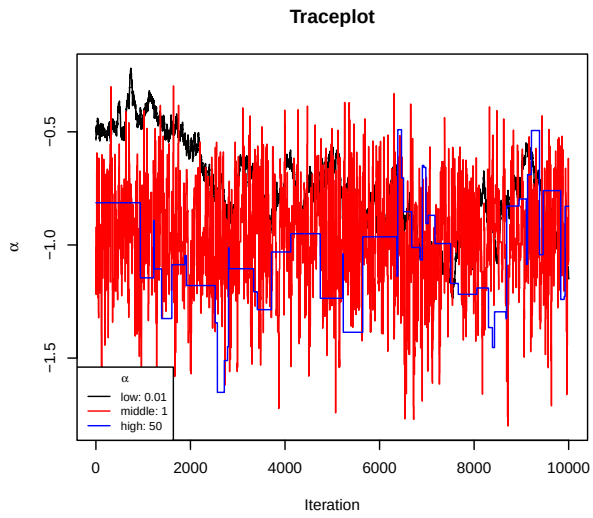
```
##          i acc.rate alpha acc.rate beta
## [1,] -4000              1             0
##          i acc.rate alpha acc.rate beta
## [1,] -3000      0.9440559     0.9090909
##          i acc.rate alpha acc.rate beta
## [1,] -2000       0.966017     0.9355322
##          i acc.rate alpha acc.rate beta
## [1,] -1000      0.9723426     0.9510163
##      i acc.rate alpha acc.rate beta
## [1,] 0      0.9740065     0.9562609
##         i acc.rate alpha acc.rate beta
## [1,] 1000      0.9696061     0.9638072
##         i acc.rate alpha acc.rate beta
## [1,] 2000      0.9691718     0.9671721
##         i acc.rate alpha acc.rate beta
## [1,] 3000      0.9714327     0.9705756
##         i acc.rate alpha acc.rate beta
## [1,] 4000      0.9715036     0.9703787
##         i acc.rate alpha acc.rate beta
```

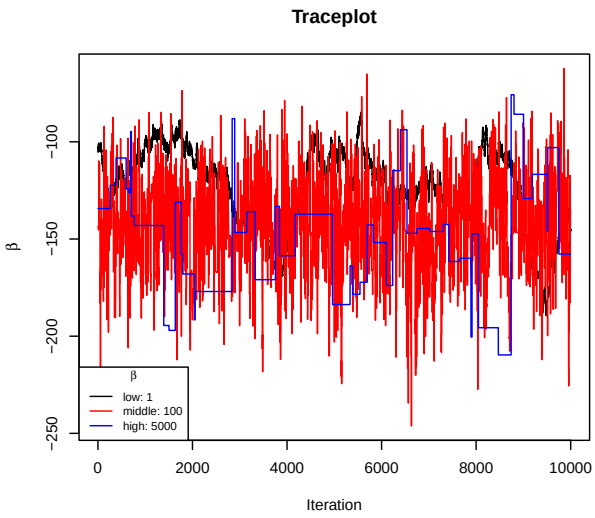
```
## [1,] 5000      0.9726697     0.9706699
##         i acc.rate alpha acc.rate beta
## [1,] 6000      0.9740026     0.9720028
##         i acc.rate alpha acc.rate beta
## [1,] 7000       0.975275     0.9740933
##         i acc.rate alpha acc.rate beta
## [1,] 8000      0.9753354     0.9750021
##         i acc.rate alpha acc.rate beta
## [1,] 9000      0.9760018     0.9746173
##          i acc.rate alpha acc.rate beta
## [1,] 10000       0.976216      0.974359
```

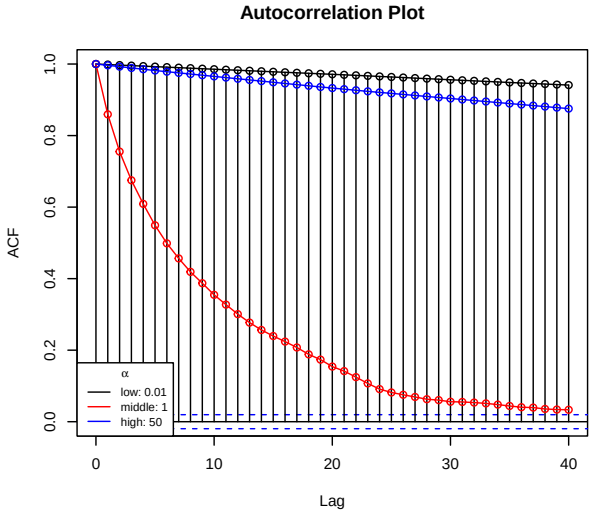
```
tuning_middle <- MH(s_alpha=1, s_beta=100, sigma2 , n.thin = 1,
                    n.burnin = 4000, n.iter=10000, show = F)
tuning_high   <- MH(s_alpha=50, s_beta=5000, sigma2 , n.thin = 1,
                    n.burnin = 4000, n.iter=10000, show = F)
```

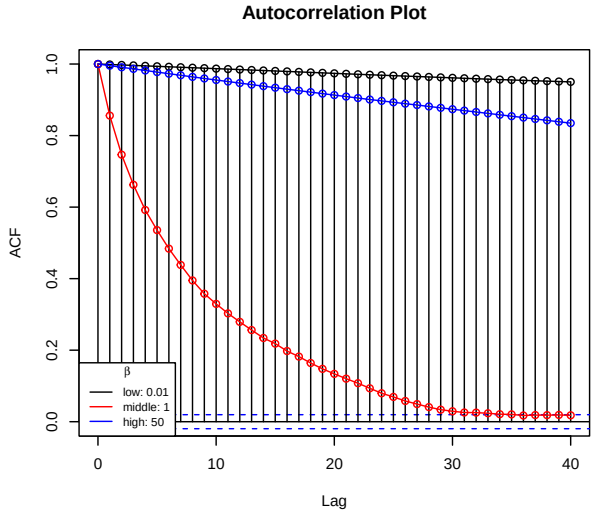
```
par(mfrow = c(3,2))
#alpha traceplot
plot(x= 1:10000, tuning_low$alpha_samples,type = "l",ylab = expression(alpha),
     xlab = "Iteration",main = "Traceplot",
     ylim = range(tuning_low$alpha_samples, tuning_middle$alpha_samples,
                  tuning_high$alpha_samples))
lines(x= 1:10000, tuning_middle$alpha_samples,col = "red")
lines(x= 1:10000, tuning_high$alpha_samples,col = "blue")
legend("bottomleft", col = c("black","red","blue"),lty = 1,
       legend = c("low: 0.01","middle: 1","high: 50"),title = expression(alpha),
       cex = 0.8,bg="white")

#beta traceplot
plot(x= 1:10000, tuning_low$beta_samples,type = "l",ylab = expression(beta),
     xlab = "Iteration",main = "Traceplot",
     ylim = range(tuning_low$beta_samples, tuning_middle$beta_samples,
                  tuning_high$beta_samples))
lines(x= 1:10000, tuning_middle$beta_samples,col = "red")
lines(x= 1:10000, tuning_high$beta_samples,col = "blue")
legend("bottomleft", col = c("black","red","blue"),lty = 1,
       legend = c("low: 1","middle: 100","high: 5000"),title = expression(beta),
       cex = 0.8,bg="white")

#alpha ACF
plot(y=acf(tuning_low$alpha_samples,plot=F)$acf,x=0:40,type = "o",col = "black",
     ylim = c(0,1),ylab = "ACF",xlab = "Lag",main = "Autocorrelation Plot")
abline(h=c(-1,1)*qnorm((1 + .95)/2)/sqrt(acf(tuning_low$alpha_samples,plot=F)$n.used),
       col = "blue",lty = 2)
abline(h=0)
lines(y=acf(tuning_low$alpha_samples,plot=F)$acf,x=0:40,type = "h",col = "black")
lines(y=acf(tuning_middle$alpha_samples,plot=F)$acf,x=0:40,type = "o",col = "red")
lines(y=acf(tuning_high$alpha_samples,plot=F)$acf,x=0:40,type = "o",col = "blue")
legend("bottomleft", col = c("black","red","blue"),lty = 1,
       legend = c("low: 0.01","middle: 1","high: 50"),title = expression(alpha),
       cex = 0.8,bg="white")

#beta ACF
```

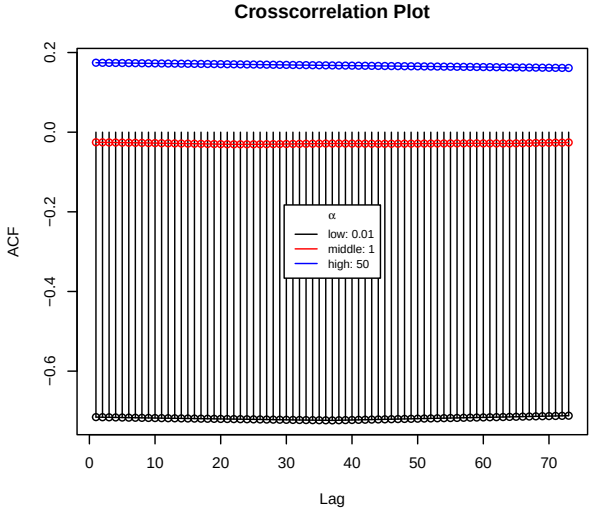
```
plot(y=acf(tuning_low$beta_samples,plot=F)$acf,x=0:40,type = "o",col = "black",
     ylim = c(0,1),ylab = "ACF",xlab = "Lag",main = "Autocorrelation Plot")
abline(h=c(-1,1)*qnorm((1 + .95)/2)/sqrt(acf(tuning_low$beta_samples,plot=F)$n.used),
       col = "blue",lty = 2)
abline(h=0)
lines(y=acf(tuning_low$beta_samples,plot=F)$acf,x=0:40,type = "h",col = "black")
lines(y=acf(tuning_middle$beta_samples,plot=F)$acf,x=0:40,type = "o",col = "red")
lines(y=acf(tuning_high$beta_samples,plot=F)$acf,x=0:40,type = "o",col = "blue")
legend("bottomleft", col = c("black","red","blue"),lty = 1,
       legend = c("low: 0.01","middle: 1","high: 50"),title = expression(beta),
       cex = 0.8,bg="white")

#alpha CCF
plot(ccf(y=tuning_low$alpha_samples,x=1:10000,plot=F)$acf,type = "o",col = "black",
     ylab = "ACF",xlab = "Lag",main = "Crosscorrelation Plot",
     ylim = range(ccf(y=tuning_low$alpha_samples,plot=F,x=1:10000)$acf,
                  ccf(y=tuning_middle$alpha_samples,plot=F,x=1:10000)$acf,
                  ccf(y=tuning_high$alpha_samples,plot=F,x=1:10000)$acf))
lines(ccf(y=tuning_low$alpha_samples,plot=F,x=1:10000)$acf,type = "h",col = "black")
lines(ccf(y=tuning_middle$alpha_samples,plot=F,x=1:10000)$acf,type = "o",col = "red")
lines(ccf(y=tuning_high$alpha_samples,plot=F,x=1:10000)$acf,type = "o",col = "blue")
legend("center", col = c("black","red","blue"),lty = 1,
       legend = c("low: 0.01","middle: 1","high: 50"),title = expression(alpha),
       cex = 0.8,bg="white")


#beta CCF
plot(ccf(y=tuning_low$beta_samples,x=1:10000,plot=F)$acf,type = "o",col = "black",
     ylab = "ACF",xlab = "Lag",main = "Crosscorrelation Plot",
     ylim = range(ccf(y=tuning_low$beta_samples,plot=F,x=1:10000)$acf,
                  ccf(y=tuning_middle$beta_samples,plot=F,x=1:10000)$acf,
                  ccf(y=tuning_high$beta_samples,plot=F,x=1:10000)$acf))
lines(ccf(y=tuning_low$beta_samples,plot=F,x=1:10000)$acf,type = "h",col = "black")
lines(ccf(y=tuning_middle$beta_samples,plot=F,x=1:10000)$acf,type = "o",col = "red")
lines(ccf(y=tuning_high$beta_samples,plot=F,x=1:10000)$acf,type = "o",col = "blue")
legend("center", col = c("black","red","blue"),lty = 1,
       legend = c("low: 0.01","middle: 1","high: 50"),title = expression(beta),
       cex = 0.8,bg="white")
```

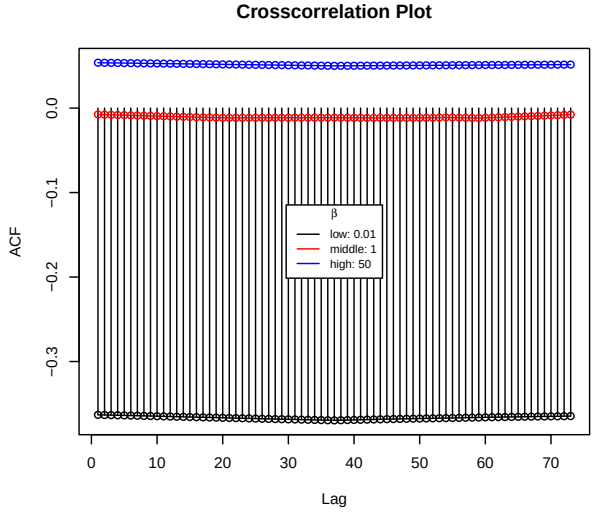












2 Comment on the differences in traceplots, auto-correlations (acf()) and cross-correlations depending

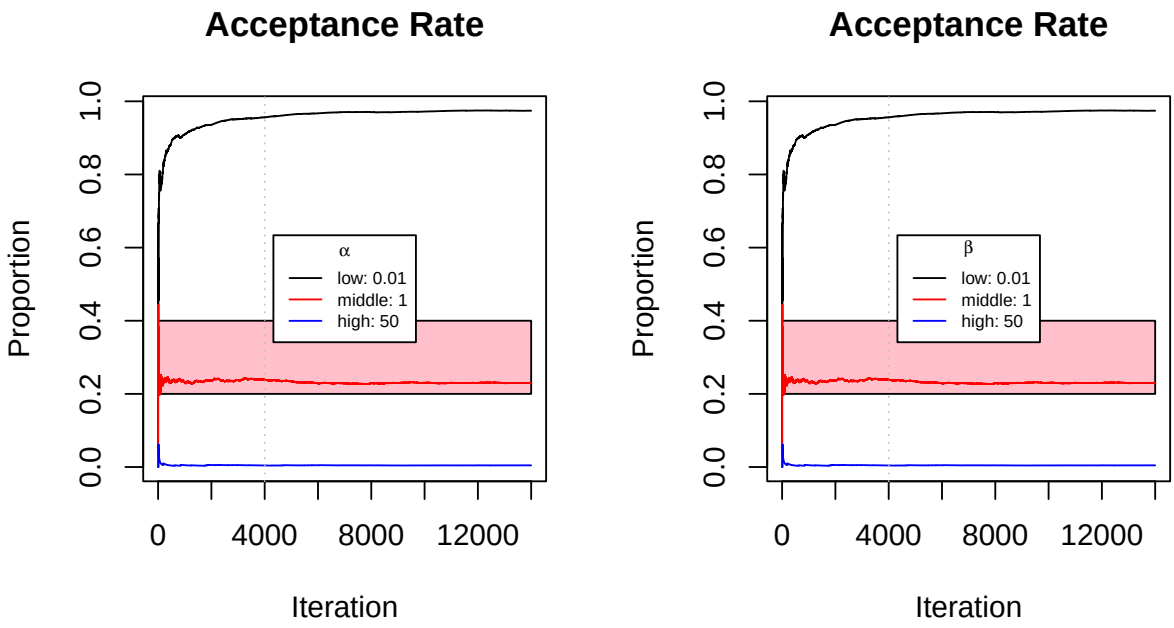
on the tuning parameters choice. What are the reasons for observed differences?

With high tuning parameters s_alpha and s_beta which are in form of the standard deviations of the proposal distribution $\theta' \sim \mathcal{N}(\theta_i, \sigma_\theta^2)$ the proposed value gets easier or harder accepted. If the standard devation is high, the proposed value is not as easily accepted (low acceptance rate) and this gets visual in the traceplot by having horizontal lines because this means that the algorithm "is stuck" with the same value. This gets also visible in the ACF and CCF plots by showing high correlation because from step to step the value is most likely still the same and therefore a large correlation.

If the standard deviation is too low, the acceptance rate is high but each updating step only converges rather slowly which means the updated value is very very similar to the value before. This also means for the ACF and CCF plots that the correlation is rather high because the iterative values, altough they are not the same, are still very similar leading to a high correlation.

3 Under which condition the optimal acceptance rate of about 0.2-0.4 ("rule of thumb") is attained?

```
par(mfrow = c(1,2))
plot(cumsum(tuning_low$alpha_yes_history)/
       c(1:length(cumsum(tuning_low$alpha_yes_history))) ,
     type = "l",main = "Acceptance Rate",ylab = "Proportion",xlab = "Iteration")
polygon( x = c(0,0,14000,14000), y=c( 0.2,0.4,0.4,0.2) ,col="pink")

lines(cumsum(tuning_middle$alpha_yes_history)/
        c(1:length(cumsum(tuning_middle$alpha_yes_history))),col = "red")
lines(cumsum(tuning_high$alpha_yes_history)/
        c(1:length(cumsum(tuning_high$alpha_yes_history))),col = "blue")
abline(v = 4000,lty = 3, col ="gray")
legend("center", col = c("black","red","blue"),lty = 1,
       legend = c("low: 0.01","middle: 1","high: 50"),title = expression(alpha),
       cex = 0.6,bg="white")
plot(cumsum(tuning_low$beta_yes_history)/
       c(1:length(cumsum(tuning_low$beta_yes_history))) ,
     type = "l",main = "Acceptance Rate",ylab = "Proportion",xlab = "Iteration")
polygon( x = c(0,0,14000,14000), y=c( 0.2,0.4,0.4,0.2) ,col="pink")
lines(cumsum(tuning_middle$beta_yes_history)/
        c(1:length(cumsum(tuning_middle$beta_yes_history))),col = "red")
lines(cumsum(tuning_high$beta_yes_history)/
        c(1:length(cumsum(tuning_high$beta_yes_history))),col = "blue")
abline(v = 4000,lty = 3, col ="gray")
legend("center", col = c("black","red","blue"),lty = 1,
       legend = c("low: 0.01","middle: 1","high: 50"),title = expression(beta),
       cex = 0.6,bg="white")
```



According to the "Rule of thumb" the optimal tuning parametes of the proposal distributions are when the α proposal distribution has a standard deviation of 1 and the β proposal distribution has a standard deviation of 100.

4 Provide summaries (mean, sd, 0.025, 0.5, 0.975 quantiles) of the marginal posteriors for α and β for the middle choice of tuning parameters.

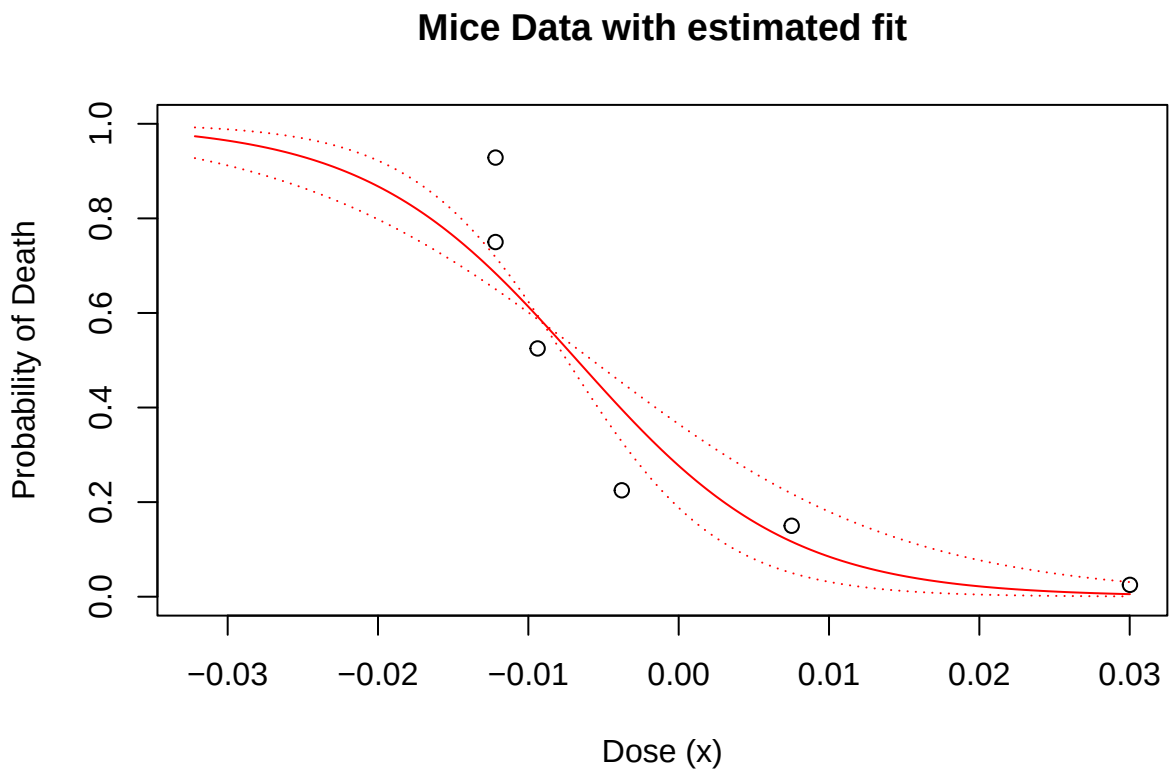
```
results <- rbind(
  "alpha"=cbind(t(quantile(tuning_middle$alpha_samples,probs = c(0.025,0.5,0.975))),
                "mean"=mean(tuning_middle$alpha_samples),
                "sd"=sd(tuning_middle$alpha_samples)),
  "beta"=cbind(t(quantile(tuning_middle$beta_samples,probs = c(0.025,0.5,0.975))),
                "mean"=mean(tuning_middle$beta_samples),
                "sd"=sd(tuning_middle$beta_samples))
)
rownames(results) <-  c("alpha","beta")
results
```

```
##              2.5%          50%       97.5%         mean         sd
## alpha   -1.461708   -0.9592253  -0.5537754   -0.9673889  0.2352573
## beta  -196.708001 -142.0147341 -96.3710245 -143.0714300 25.3517136
```

5 Plot the logistic curve for median posterior values of α and β together with data and interpret the result.

```
  mypi <- function(alpha, beta, x){
    tmp <- exp(alpha + beta*x)
    pi <- tmp/(1+tmp)
    return(pi)
```

```
  }

x_grid <- seq(from = min(x)-0.02,to = max(x),length.out = 1000)
plot(x=x_grid, y= mypi(alpha = results[1,2], beta  = results[2,2],x =x_grid ),
     col = "red",type = "l",ylab = "Probability of Death",xlab = "Dose (x)",
     ylim = c(0,1),
     main = "Mice Data with estimated fit")
lines(x=x_grid, y= mypi(alpha = results[1,1], beta  = results[2,1],x =x_grid ),
      col = "red",lty = 3)
lines(x=x_grid, y= mypi(alpha = results[1,3], beta  = results[2,3],x =x_grid ),
      col = "red",lty = 3)
lines(x,y/n,type = "p")
```



We see that the estimated parameters by the Metropolis-Hastings algorithm nicely fit the data.