Group tasks

Exercise 2 (Gibbs sampler - 12 points)

(a) Complete the derivations from the lecture:

$$\begin{aligned} p(x) &\propto \exp(-0.5(ax^2 - 2xb)) \\ &\propto \exp(-0.5a(x^2 - 2xb/a)) \\ &\propto \exp(-0.5a((x - b/a)^2 - (b/a)^2) \\ &\propto \exp(-0.5a(x - b/a)^2) \exp(a/2b/a)^2) \\ &\propto \exp(-0.5a(x - b/a)^2)). \end{aligned}$$

This corresponds to a Gaussian kernel with expectation b/a and variance $1/a$.
(b) Complete the following tasks:

1. In case you are using your own Gibbs sampler and your results under conditions 2 and 3 below do not match the exact results provided by R-INLA (https://www.r-inla.org/), please improve your own code.

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

```r
set.seed(44566)
n <- 30
mu <- 4
sigma2 <- 16 #\sigma^2
y <- rnorm(n=n, mean=mu, sd=sqrt(sigma2))
```

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


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

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

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))

####################
## Gibbs sampler (1 chain)
####################
# initialization
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")
  # }

}

# 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"]
```

```r
par(mfrow = c(3,1))
for (i in 1:3) {
  plot(gibbs_samples[,i], type = "l", xlab = "Iteration",
       ylab = "Value")
  title(main = paste("Traceplot for", colnames(gibbs_samples)[i]))
}
```

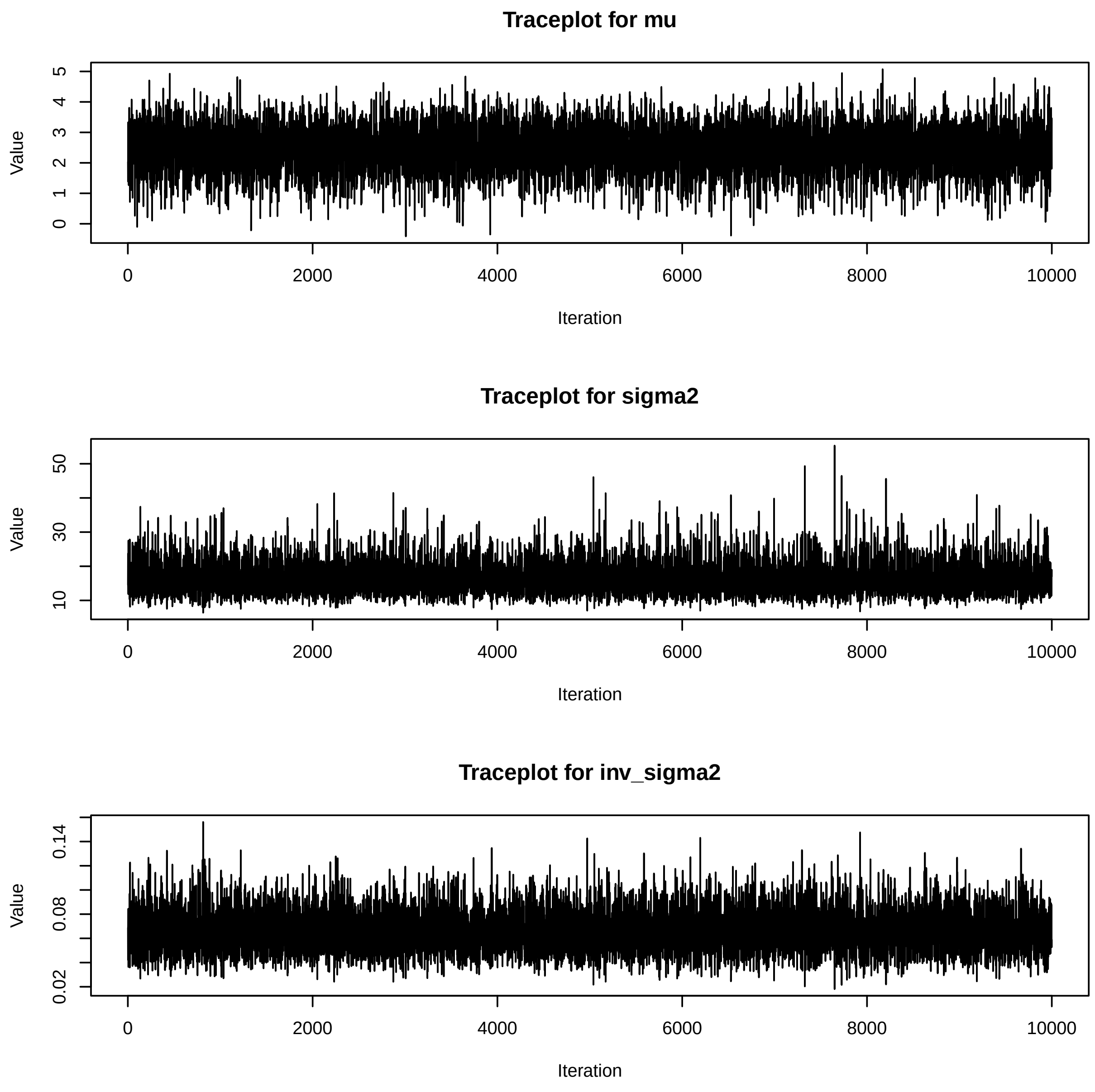

Figure 1: Traceplots for μ, σ^2 and $1/\sigma^2$ in Exercise 2 (Gibbs sampler).

```
summary(gibbs_samples)

##        mu               sigma2         inv_sigma2
##  Min.   :-0.4126   Min.   : 6.403   Min.   :0.01808
##  1st Qu.: 1.9865   1st Qu.:13.145   1st Qu.:0.05335
##  Median : 2.4692   Median :15.567   Median :0.06424
##  Mean   : 2.4479   Mean   :16.333   Mean   :0.06562
##  3rd Qu.: 2.9279   3rd Qu.:18.743   3rd Qu.:0.07608
##  Max.   : 5.0731   Max.   :55.318   Max.   :0.15619

for(ii in 1:3) {
  print(colnames(gibbs_samples)[ii])
  print(quantile(gibbs_samples[, ii], probs = c(0.025, 0.5, 0.975)))
}

## [1] "mu"
##      2.5%       50%     97.5%
## 0.9436683 2.4692436 3.8080215
## [1] "sigma2"
##      2.5%       50%     97.5%
##  9.733305 15.566910 27.221752
## [1] "inv_sigma2"
##       2.5%        50%      97.5%
## 0.03673533 0.06423882 0.10274003
```

4. Explanation of the code: First of all we set the number of thinning, the number of burn-in iterations and the length of the MCMC chain. `n.tot` is the number of total iterations the Gibbs sampler must be run. Next, we define a matrix with the number of rows equal to the number of iterations and the number of columns equal to the number of parameters that need to be estimated by MCMC algorithm. Gibbs sampler starts by initializing the MCMC chains. The `for` loop goes through `n.tot` iterations and computes parameters according to the full conditional posterior distributions for μ and σ^2 (see lecture script, page 46) at each iteration. Moreover, the code generates the samples for the precision $1/\sigma^2$ as the reciprocal of σ^2. The code also cuts the first `n.burnin` many iterations and saves every `n.thin`-th iterations. Moreover, the code prints in which iteration the chain is after each 1000 iteration.

```
library(MASS)
par(mfrow=c(1,3), mgp=c(1, 0.5, 0))
# plot for mean
rg <- range(inla.output$marginals.fixed$"(Intercept)"[,2])
truehist(mu_gibbs_samples, prob=TRUE, col="yellow",
```

```r
         xlab=expression(mu), mar = c(1,1,0,1))
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), mar = c(1,1,0,1))
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), mar = c(1,1,0,1))
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")
```

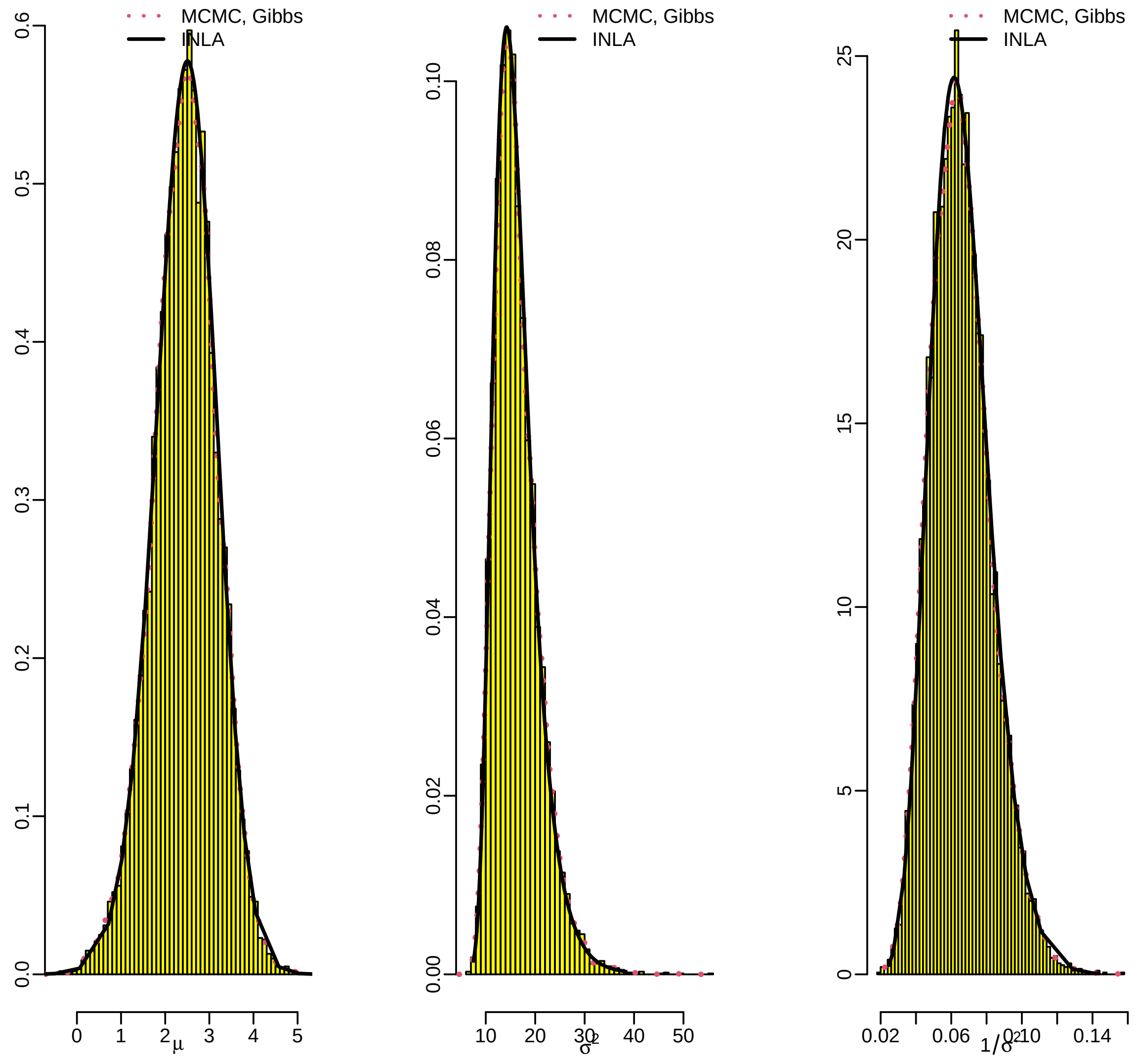

Figure 2: The overlap of MCMC Gibbs sampler and INLA.

It is interesting to see whether the marginal posterior distributions obtained from INLA and by Gibbs sampler overlap. Indeed, Figure 2 shows that the posterior densities overlap.

Exercise 3 (Metropolis-Hastings sampler for a logistic regression - 18 points)

```r
## Metropolis-Hastings for logistic model
## Two independent normal proposals

rm(list=ls())

## Data from Collett, D. (2003) Modelling Binary Data 2nd Edition
## Table 1.6 p. 7 Number of deaths from pneumonia amongst batches of 40 mice
## exposed to different doses of an anti-pneumococcus serum (grouped)
## Table 3.1 p. 71 (original) Number of deaths from pneumonia in mice exposed
## to various doses of an anti-pneumococcus serum

# the covariate values (dose)
# x_original <- c(0.0028, 0.0056, 0.0112, 0.0225, 0.0450)
x_original <- c(0.0028, 0.0028, 0.0056, 0.0112, 0.0225, 0.0450)
# the centered covariate values
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)

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



####################
## Bayesian analysis
####################

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

```r
mhSampler <- function(s_alpha = 1, s_beta = 60, n.thin = 1) {
  #Generates MCMC samples through MH Sampler
  ##s_alpha, s_beta: standard deviations for the normal proposal
  ##n_thin: number of thinning

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

set.seed(44566)

# 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)


######################################
## univariate random walk proposals ##
######################################


alpha_samples <- c()
beta_samples <- c()
# number of accepted proposals
alpha_yes <- 0
beta_yes <- 0

# starting values
alpha <- 0
beta <- 0


# 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 accetpance rate
logacc <- enum - denom

if(log(runif(1)) <= logacc){
  # accept the proposed value
  beta <- beta_star
  beta_yes <- beta_yes + 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(i%%1000 ==0){
#   # print the acceptance rates on the fly
#   cat(c(i, alpha_yes/count, beta_yes/count), "\n")
# }
```

```
}
  return(list(
    samples = data.frame(alpha = alpha_samples,
                         beta = beta_samples),
    alpha_accepted = alpha_yes,
    beta_accepted = beta_yes,
    iterations = count))
}

mhSampler_low <- mhSampler(s_alpha = 0.01, s_beta = 1)
mhSampler_middle <- mhSampler(s_alpha = 1, s_beta = 100)
mhSampler_high <- mhSampler(s_alpha = 50, s_beta = 5000)
```

1. What the code in `04MHSample.R` is doing?

   At each iteration, we draw a sample from one of the proposal distributions and compute the acceptance rate by dividing the posterior with the new parameter by the posterior with the old parameter. To ensure numerical stability, the code uses a log transformation. Samples are accepted only if the log acceptance rate is larger or equal to the log of a random uniform value. Otherwise, the sample is rejected and the previous value is assigned.

2. Comment on the differences in plots depending on the tuning parameters' choice. What are the reasons for observed differences?

```
mh_plot <- function(list) {
  with(list, {
    par(mar = c(2,4,3,2) + 0.1)
    layout(matrix(c(1,2,5,3,4,5), nrow = 3, ncol = 2))
    for (i in 1:2) {
      plot(samples[,i], type = "l", xlab = "Iteration",
           ylab = "Value")
      abline(h = quantile(samples[,i], 0.025), lty = 2, col = "red")
      abline(h = quantile(samples[,i], 0.975), lty = 2, col = "red")
      title(main = paste("Traceplot for", colnames(samples)[i]))
      acf(samples[,i], main = " ", ci = 0)
      title(main = paste("Autocorrelation for", colnames(samples[i])))
    }
    ccf(samples[,1], samples[,i], type = "correlation",
        main = " ", ci = 0)
    title(main = "Crosscorrelation for alpha and beta")
  })
}
```

```
mh_plot(mhSampler_low)
```

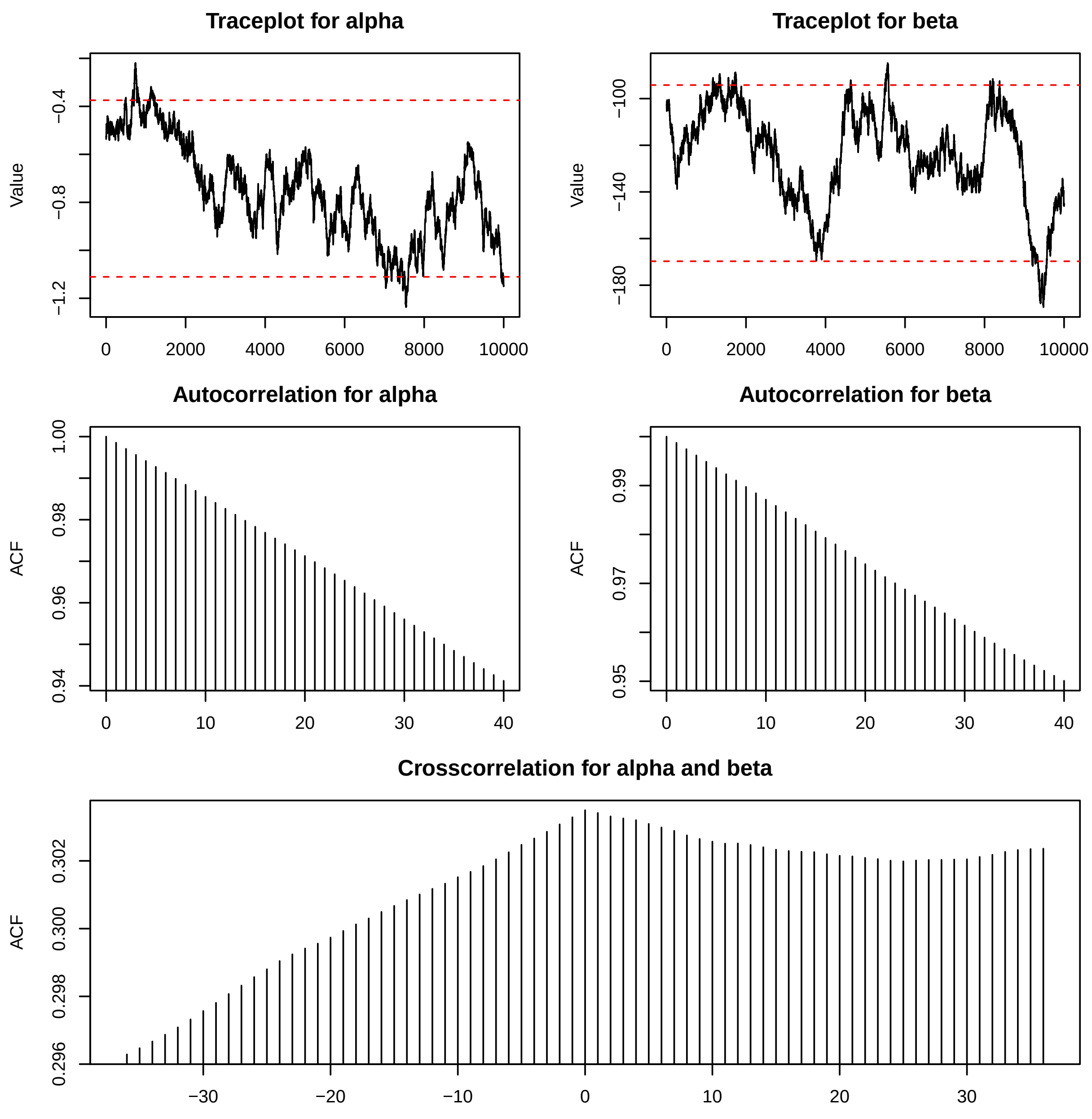

Figure 3: The traceplots and the auto-correlation plots for α and β and the corss-correlation plot for α and β in low tuning parameters case. Horizontal lines are 2.5% and 97.5% percentiles.

Low tuning parameters

When using the lower tuning settings, the MH sampler is heavily auto- and cross-correlated even for larger lags. This leads to highly dependent chains, and a failure to fully explore the parameter space. The auto-correlation and cross-corellation plots in Figure 3 show that in case of low tuning parameters the samples are heavily correlated because the algorithm takes only small steps when exploring the parameter space and, therefore, the draws are very close to each other. In this case the acceptance rate is high because the proposed values come from a very small region around the previously accepted value. The traceplot shows that the sampler stays at the same value for quite long.

Middle tuning parameters

With middle tuning parameters, the MH sampler appears much more well-behaved. The traceplots in Figure 4 show that a large proportion of the parameter space is explored. Furthermore, we observe a distinct reduction in auto- and cross-correlation for larger lags. Thinning of the chain could reduce the correlations even further.

High tuning parameters

The steps of the random walk are too large, the acceptance rate is close to 0 and most of the proposed values get rejected. This means that the algorithm accepts the proposed values not often enough so that the chain remains at the same value for a large number of iterations. Therefore, the auto-correlations and cross-correlations of the resulting sample are high. The traceplot shows steps which means that the sampler remained in the same value for too long.

Interesting to note, that the auto-correlation and cross-correlation plots in cases of low and high tuning parameters show that the samples are heavily correlated but because of different reasons. Furthermore, the traceplots in the low and high tuning parameters case show that the algorithm stays at the same pont for a long time, but because of different reasons.

As noted previously too low or too high tuning parameters are problematic. Proposal distributions with too low spread are very uninformative, leading to a log acceptance rates that are very high, which in turn means that the algorithm will accept samples too often, even if the change is rather small.

On the other hand, proposal distributions with too large spread lead to samples close to the previous value. The algorithm will only accept new samples whenever the uniform random sample happens to be extremely small. With this the sample moves too slow and doesn't mix well.

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

```r
acc_rates_low <- with(mhSampler_low, {
  c(alpha_accepted/iterations,
  beta_accepted/iterations)})
```

```r
mh_plot(mhSampler_middle)
```

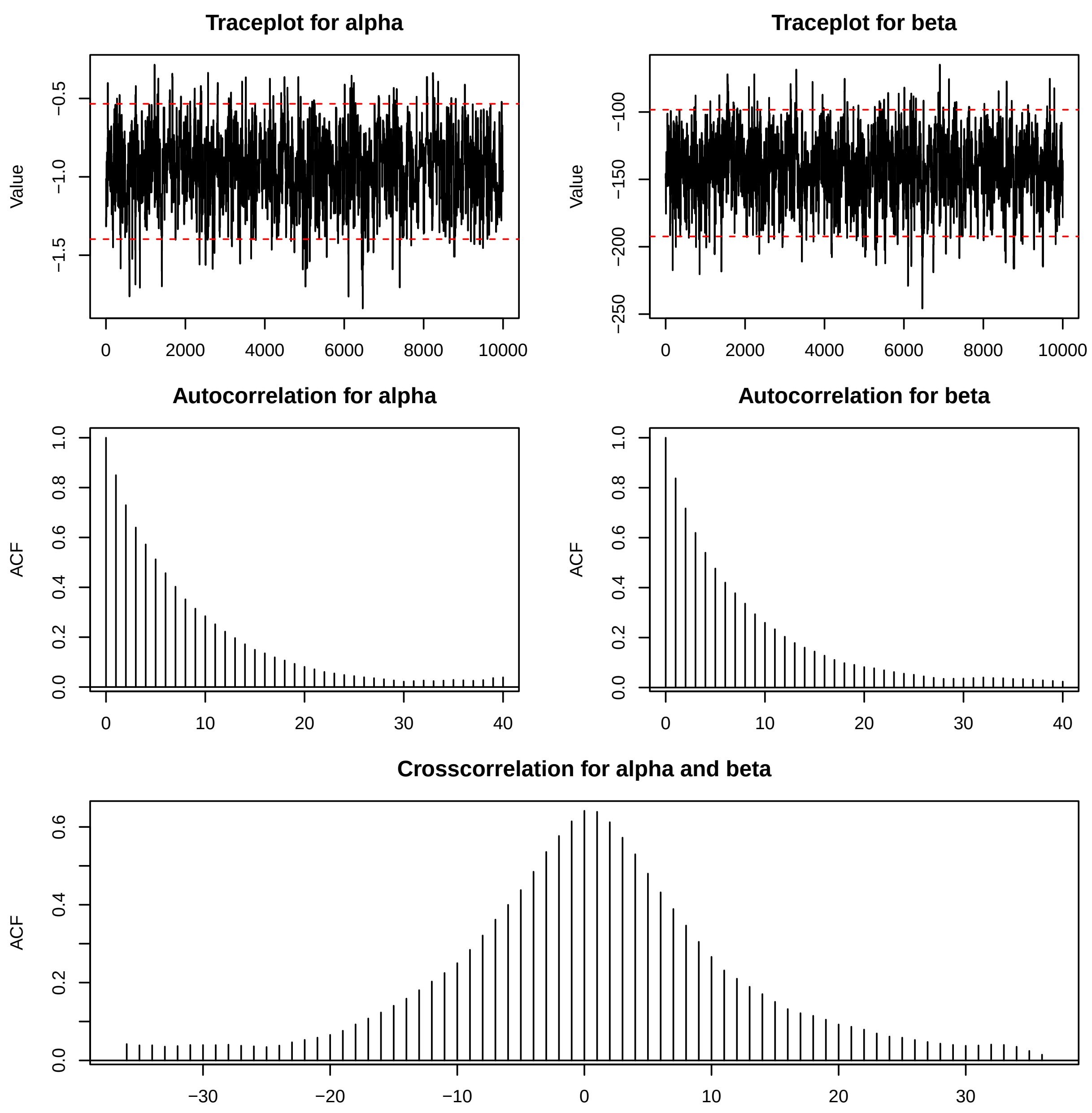

Figure 4: The traceplots and the auto-correlation plots for α and β and the corss-correlation plot for α and β in middle tuning parameters case. Horizontal lines are 2.5% and 97.5% percentiles.

```
mh_plot(mhSampler_high)
```

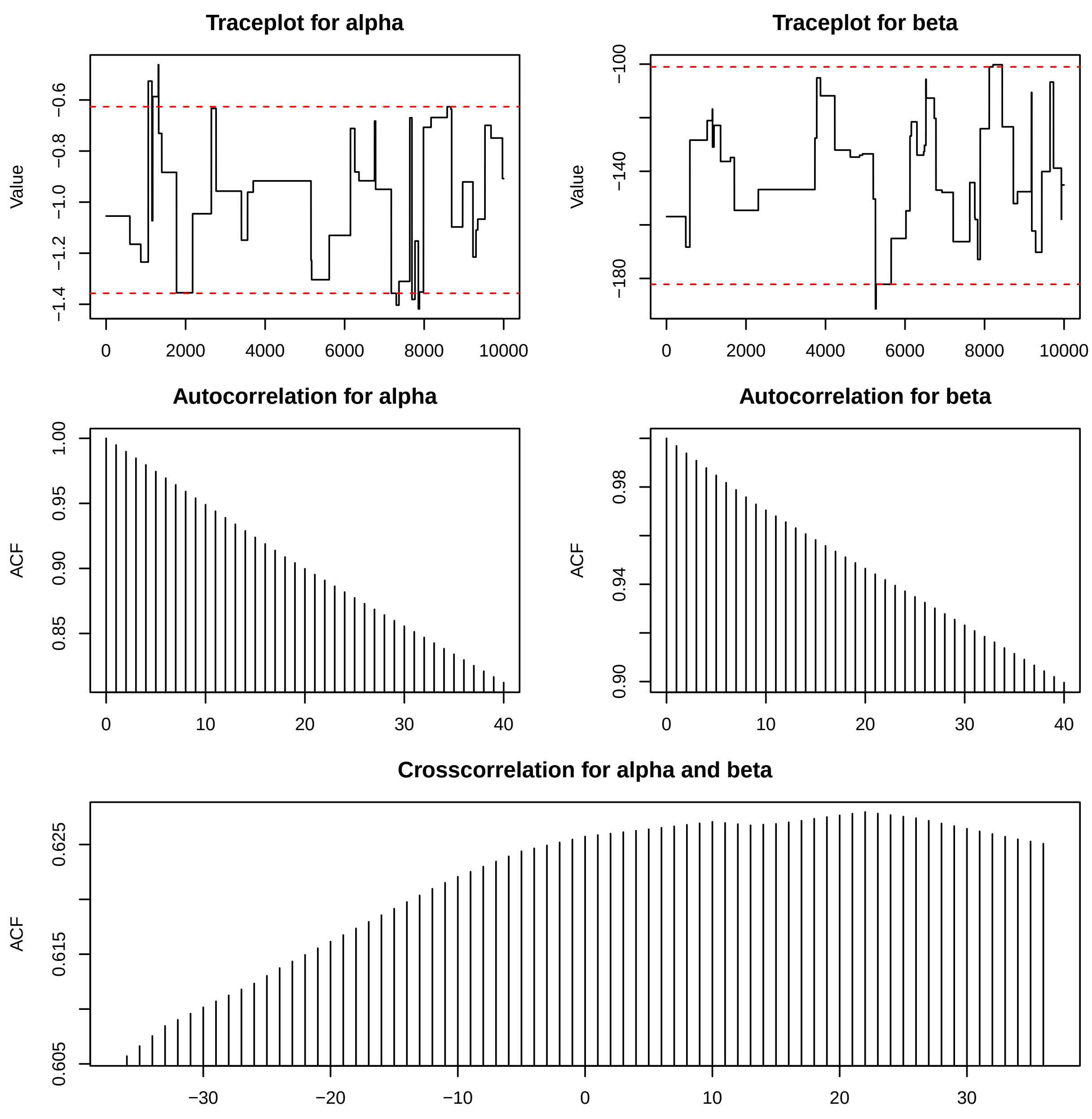

Figure 5: The traceplots and the auto-correlation plots for α and β and the corss-correlation plot for α and β in high tuning parameters case. Horizontal lines are 2.5% and 97.5% percentiles.

```r
acc_rates_middle <- with(mhSampler_middle, {
  c(alpha_accepted/iterations,
  beta_accepted/iterations)})

acc_rates_high <- with(mhSampler_high, {
  c(alpha_accepted/iterations,
  beta_accepted/iterations)})

acc_rates <- data.frame(cbind(c(0.01, 1, 50), c(1, 100, 5000),
                              rbind(acc_rates_low,
                                    acc_rates_middle,
                                    acc_rates_high)))
colnames(acc_rates) <- c("s_alpha", "s_beta", "acc_alpha", "acc_beta")
rownames(acc_rates) <- c("Low", "Middle", "High")
knitr::kable(acc_rates, digits = 3)
```

	s_alpha	s_beta	acc_alpha	acc_beta
Low	0.01	1	0.976	0.974
Middle	1.00	100	0.217	0.231
High	50.00	5000	0.005	0.005

The rule of thumb of an acceptance rate of 0.2 - 0.4 is achieved using the middle tuning parameters, yielding an acceptance rate for alpha of 21.7 % and an acceptance rate for beta of 23.1 %.

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.

```r
summary(mhSampler_middle$samples)

##      alpha                 beta        
##  Min.   :-1.8400   Min.   :-245.82  
##  1st Qu.:-1.0969   1st Qu.:-156.74  
##  Median :-0.9353   Median :-139.36  
##  Mean   :-0.9499   Mean   :-141.18  
##  3rd Qu.:-0.7971   3rd Qu.:-124.76  
##  Max.   :-0.2842   Max.   : -64.85  

for(ii in 1:2) {
  print(colnames(mhSampler_middle$samples)[ii])
  print(quantile(mhSampler_middle$samples[, ii],
                 probs = c(0.025, 0.5, 0.975)))
}
```

```
## [1] "alpha"
##       2.5%        50%      97.5%
## -1.3975759 -0.9352910 -0.5340364
## [1] "beta"
##       2.5%        50%      97.5%
## -192.37525 -139.35901  -98.32787
```

5. Figure 6 shows the logistic curve for median posterior values of α and β together with data and interpret the result. We see that the relation between the dosage and response takes an S - shape (left panel) but on the logit scale it is linear (right panel). The plot shows that median posteriors fit quite well to the data. Moreover, Figure 6 shows that the probability of death decreases with increasing dose of the anti-pneumocossus serum with posterior median OR exp(-139.359) with 95 % CrI (OR): (exp(-192.3753), exp(-98.3279)).

```
# plot the dose-response curve
x_range <- seq(from=range(x)[1], to=range(x)[2], by=0.001)
logits <- quantile(mhSampler_middle$samples[, "alpha"], probs = 0.5) +
  quantile(mhSampler_middle$samples[, "beta"],
           probs = 0.5) * x_range
probs <- exp(logits) / (1+exp(logits))

par(mar = c(2,4,3,2) + 0.1, mfrow = c(1,2), pty = "s")
plot(x_range, probs, ylim=c(0,1), type="l",
     xlim = c(range(x)[1], range(x)[2]),
     lwd=3, lty=2,  col="blue",  xlab="Centered dosage",
     ylab="Proportion deaths")
points(x, y/n)

# plot the dose-response curve on the logit scale
plot(x_range, logits, xlab="Centered dosage", ylim = c(-6,2),
     ylab="logit(Proportion deaths)",  col=" green",
     type="l", xlim = c(range(x)[1], range(x)[2]))
points(x, log((y/n)/(1-y/n)))
```

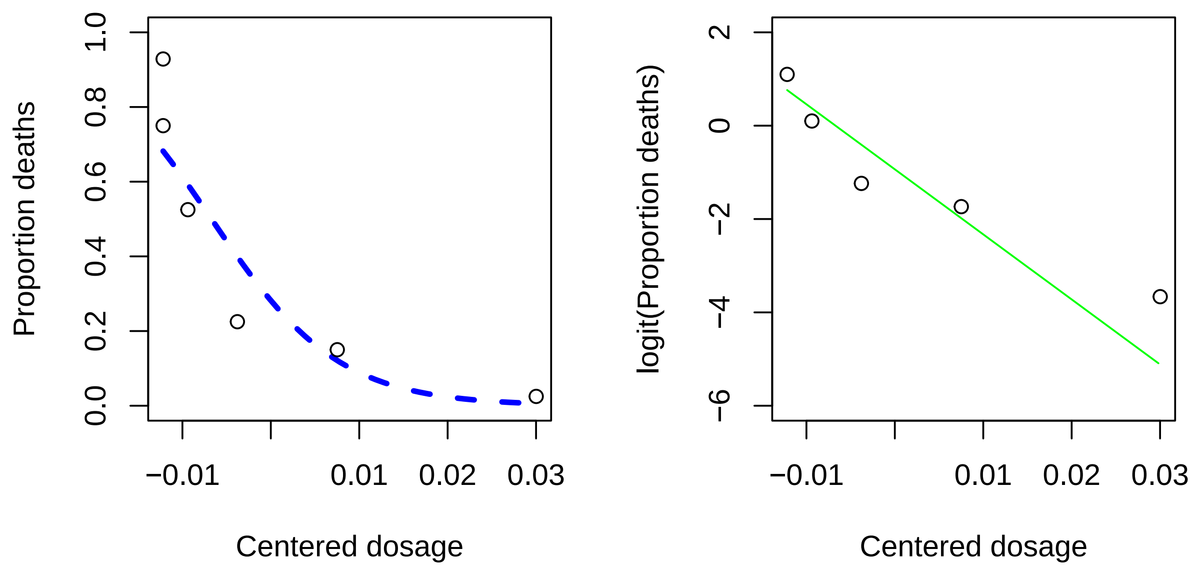

Figure 6: The dosage-response curve (left) and dosage-response curve on the logit scale (right) for the Mice data in Exercise 3.