Homework 3S

Olivia Hackworth

4/8/2019

```
library(tidyverse)
#library(nimble)
library(MASS)
library(invgamma)
library(knitr)
library(coda)
```

Let the data generating model be

Likelihood:
$y_i \sim N(\mu, \sigma^2), i = 1, 2, ..., n$

Priors:
$\mu \sim N(\mu_0, \tau_0^2)$
$\sigma^2 \sim Inv - Gamma(\alpha, \beta)$
$\mu_0 = 2, \tau_0 = 6, \alpha = 5, \beta = 5$

1. Simulate n = 20 observations according to $y_i \sim N(3, 1), i = 1, ..., n$. Use this data for subsequent problems.
2. Write down the joint posterior distribution $p(\mu, \sigma^2 | y_{1:n})$. Run a metropolis algorithm for this posterior.
3. Let $\lambda = log(\sigma^2)$, write down the joint posterior $p(\mu, \lambda | y_{1:n})$. Run a metropolis algortithm for this posterior.
4. Derive the conditional posterior distributions $p(\mu | \sigma^2, y_{1:n})$.,$p(\sigma^2 | \mu, y_{1:n})$.
5. Run a Gibbs sampler for $p(\mu, \sigma^2 | y_{1:n})$.
6. If the quantity of interest is σ^2, compare the results from 2,3, and 5 (in terms of computational efficiency and statistical efficiency).
7. If the quantity of interest is μ/σ, compare the results from 2,3, and 5 (in terms of computational efficiency and statistical efficiency).

1

```
n = 20
mu_0 = 2
tau_0 = 6
alpha = 5
beta = 5

obs = rnorm(20,3,1)
obs
```

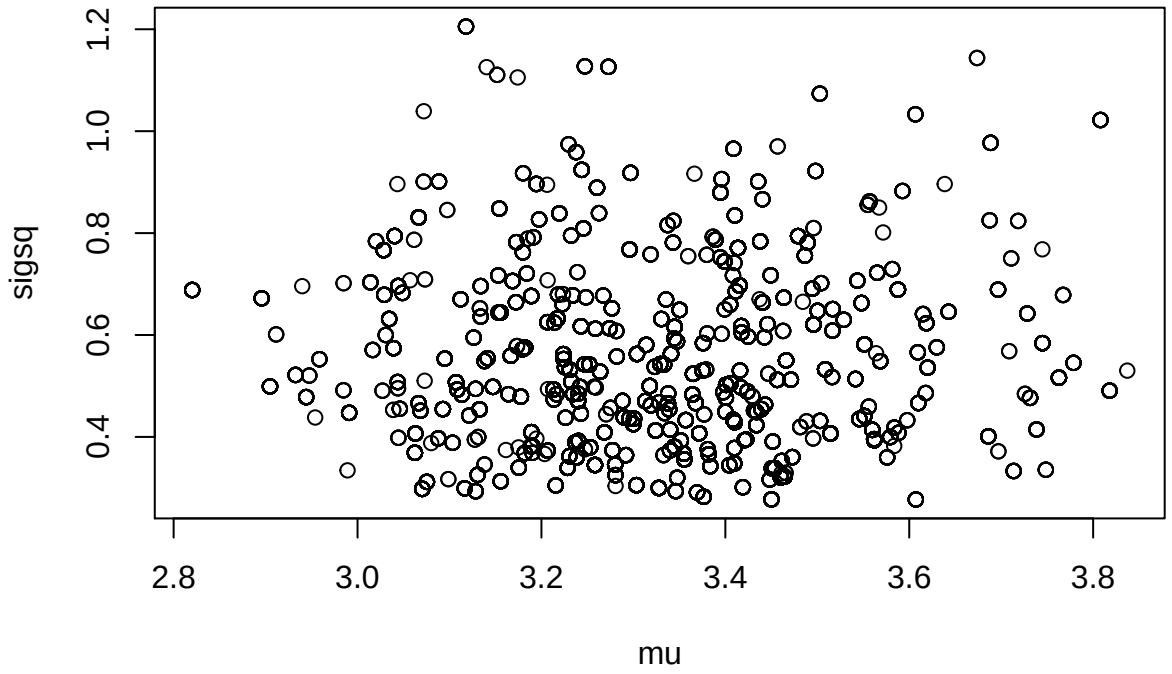
```
##  [1] 2.596302 2.082011 5.606375 3.587024 1.685190 2.726580 3.071134
##  [8] 2.739590 3.856902 2.986692 2.994492 3.458866 2.674611 3.563816
## [15] 3.869340 3.584412 3.649777 4.026180 3.567889 4.344585
```

2

Joint posterior distribution:
$p(\mu, \sigma^2|y_{1:n}) = p(\mu) * p(\sigma^2) * p(y|\mu, \sigma^2)$
$= [(\frac{1}{\sqrt{2\pi\tau_0^2}}) * e^{-(\mu-\mu_0)^2/2\tau_0^2}] * [\frac{\beta^\alpha}{\Gamma(\alpha)}(\sigma^2)^{\alpha-1}e^{-\beta/\sigma^2}] * [\prod_{i=1}^{n}(\frac{1}{\sqrt{2\pi\sigma^2}}) * e^{-(y_i-\mu_0)^2/2\sigma^2}]$

```
jointpost = function(mu, sigsq){
  #prior mu
  pmu = dnorm(mu,mu_0,tau_0,log = T)
  #prior sigsq
  psigsq = dinvgamma(sigsq,alpha,scale = beta,log = T)
  #likelihood
  py = sum(dnorm(obs,mu,sd = sqrt(sigsq),log = T))
  #joint posterior - log
  p = pmu+psigsq+py

  #p = exp(p)

  return(p)
}

one_iteration <- function(start){
  #jumping rule = multivariate normal
  #proposal
  sig = matrix(c(1,0,0,1),2,2)
  proposal = mvrnorm(n = 1, mu = c(start[1],start[2]), Sigma = sig)
  if(proposal[2] < 0){
    proposal[2] <- 0.0001   #safety net -sigsq can't be neg
    }

  #calculate r
  num = jointpost(proposal[1],proposal[2])
  denom = jointpost(start[1],start[2])
  r = exp(num-denom)

  #accept or reject
  accept_rate <- min(1, r)
  new <- start
  if(runif(1) < accept_rate){
    new <- proposal
    }
  return(new)
  }

MH_post <- function(Niter = 1000, burnin = 500){
  #blank matrix to store samples
  samples_mat <- matrix(NA, nrow = Niter, ncol = 2) #cols = mu and sigsq
  #random starting point
  samples_mat[1,] = c(1,1)
  #run through algorithm
  for(i in 2:Niter){
      samples_mat[i,] <- one_iteration(samples_mat[i - 1,])
      #print(i)
  }
```

```
  return(samples_mat[burnin:Niter,])
}

MH_post_data = MH_post(10000,500)

plot(MH_post_data[,1],MH_post_data[,2], xlab = "mu", ylab = "sigsq")
```



Note: In the one_iteration function, there is an if statement that reassigns negative sigma squared values to a value close to 0. In this scenario, this action doesn't have a big impact, but should be updated for future iterations. A better option would be to reject all proposals that include a negative sigma squared values.

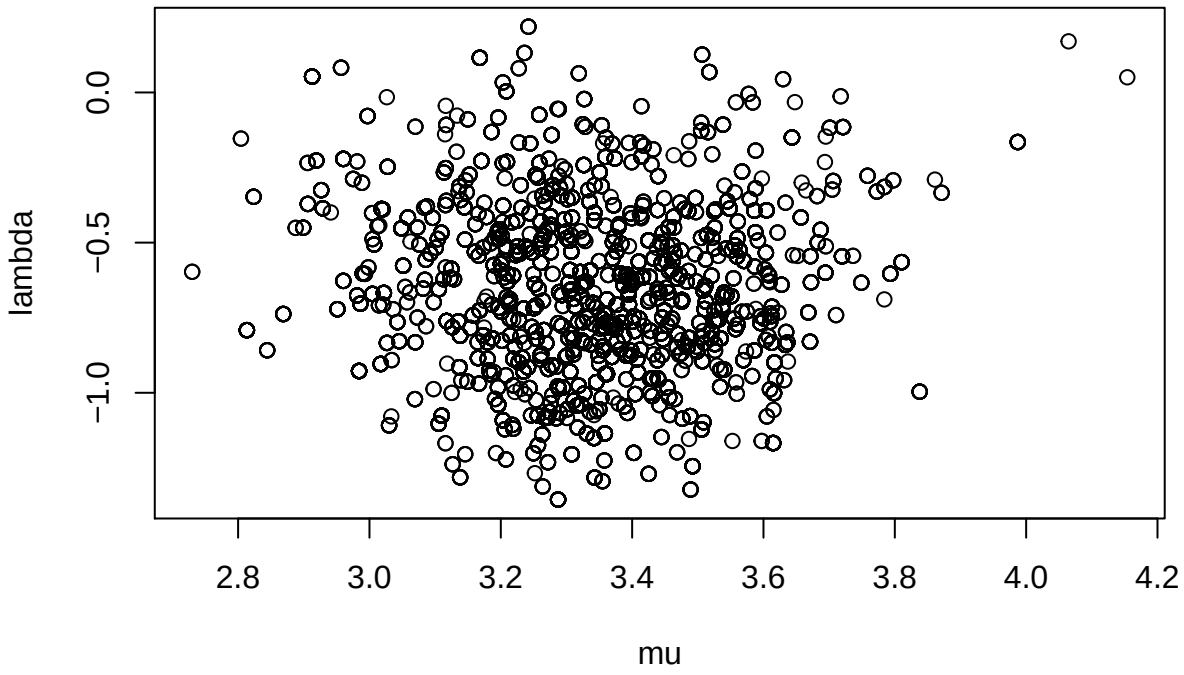
3

Joint posterior with lambda:
$\lambda = log(\sigma^2)$ so $\sigma^2 = e^\lambda$
$J = e^\lambda$

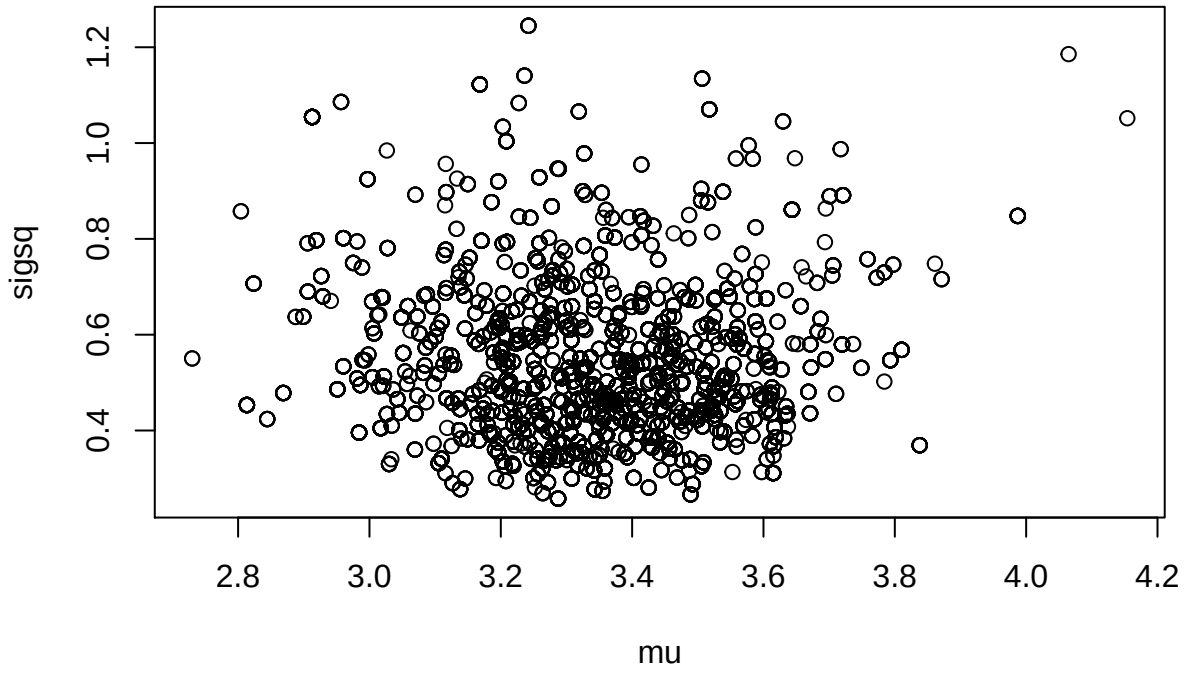
$p(\mu, \lambda|y_{1:n}) = [(\frac{1}{\sqrt{2\pi\tau_0^2}}) * e^{-(\mu-\mu_0)^2/2\tau_0^2}] * [\frac{\beta^\alpha}{\Gamma(\alpha)}(e^\lambda)^{\alpha-1}e^{-\beta/e^\lambda}] * e^\lambda * [\prod_{i=1}^n(\frac{1}{\sqrt{2\pi e^\lambda}}) * e^{-(y_i-\mu_0)^2/2e^\lambda}]$

```
jointpost_lambda = function(mu, lambda){
  #prior mu
  pmu = dnorm(mu,mu_0,tau_0,log = T)
  #prior lambda
  plambda = dinvgamma(exp(lambda),alpha,scale = beta,log = T)+lambda
  #likelihood
  py = sum(dnorm(obs,mu,sd = sqrt(exp(lambda)),log = T))
  #joint posterior - log
  p = pmu+plambda+py

  #p = exp(p)

  return(p)
```

```
}

one_iteration2 <- function(start){
  #jumping rule = multivariate normal
  #proposal
  sig = matrix(c(1,0,0,1),2,2)
  proposal = mvrnorm(n = 1, mu = c(start[1],start[2]), Sigma = sig)

  #calculate r
  num = jointpost_lambda(proposal[1],proposal[2])
  denom = jointpost_lambda(start[1],start[2])
  r = exp(num-denom)

  #accept or reject
  accept_rate <- min(1, r)
  new <- start
  if(runif(1) < accept_rate){
    new <- proposal
    }
  return(new)
  }

MH_post_lambda <- function(Niter = 1000, burnin = 500){
  #blank matrix to store samples
  samples_mat <- matrix(NA, nrow = Niter, ncol = 2) #cols = mu and lambda
  #random starting point
  samples_mat[1,] = c(1,1)
  #run through algorithm
  for(i in 2:Niter){
      samples_mat[i,] <- one_iteration2(samples_mat[i - 1,])
      #print(i)
  }
  return(samples_mat[burnin:Niter,])
}

MH_post_data_lambda = MH_post_lambda(10000,500)

plot(MH_post_data_lambda[,1],MH_post_data_lambda[,2], xlab = "mu", ylab = "lambda")
```




```
MH_post_data_sigsq = cbind(MH_post_data_lambda[,1],exp(MH_post_data_lambda[,2]))
plot(MH_post_data_sigsq[,1],MH_post_data_sigsq[,2], xlab = "mu", ylab = "sigsq")
```



4

When we condition on one parameter, we fix it which makes our model a single parameter model. This means we can use the single parameter results as the conditionals.

On page 42:

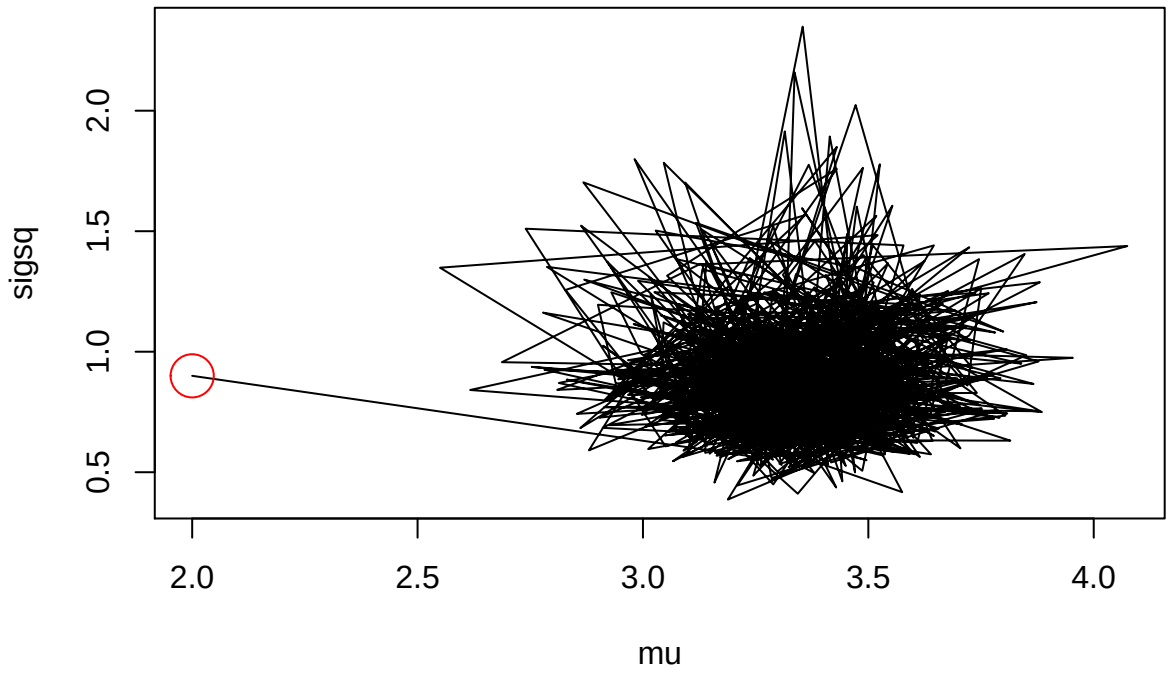
$p(\mu|\sigma^2, \mathbf{y}_{1:n}) = N(\mu_n, \tau_n)$ where $\mu_n = \frac{\frac{1}{\tau_0^2}\mu_0 + \frac{\sum y_i}{\sigma^2}}{\frac{1}{\tau_0^2}+\frac{n}{\sigma^2}}$ and $\frac{1}{\tau_n^2} = \frac{1}{\tau_0^2} + \frac{n}{\sigma^2}$

Using the formula on page 43 and then converting from a scaled inverse chi square to an inverse gamma: $p(\sigma^2|\mu, \mathbf{y}_{1:n}) = Inv - Gamma(2\alpha + n, (2\alpha + n)/2 * (\frac{2\beta + n*\nu}{2\alpha+n}))$ where $\nu = \frac{1}{n}\sum(y_i - \mu)^2$ and α and β are the original parameters given in the set up of the assignment.

Note: The inverse Gamma shape parameter is incorrect. It should be alpha +n/2.

5

```
mu_conditional = function(sigsq_star){
  mu_n = ((1/tau_0^2)*mu_0+(sum(obs)/sigsq_star))/((1/tau_0^2)+(n/sigsq_star))
  tau_n_sq = 1/((1/tau_0^2)+(n/sigsq_star))

  mu_new = rnorm(1,mu_n,sd = sqrt(tau_n_sq))
  return(mu_new)
}

sigsq_conditional = function(mu_star){
  #scaled inverse chi square parameters
  a = 2*alpha+n
  nu = (1/n)*sum((obs-mu_star)^2)
  b = (2*beta+n*nu)/(2*alpha+n)

  #switch to inverse gamma parameters
  param_1 = a/2
  param_2 = (a/2)*b

  sigsq_new = rinvgamma(1,shape = param_1,rate = param_2)
  return(sigsq_new)
}

Niter = 1000
params_mat = array(0, c(Niter, 2))
params_mat[1,] = c(2,0.9) # starting point
for(k in 2:Niter){
  params_mat[k, 1] = mu_conditional(params_mat[k-1,2])
  params_mat[k, 2] = sigsq_conditional(params_mat[k,1])
}

plot(params_mat, type = 'l', xlab = "mu", ylab = "sigsq")
points(params_mat[1, 1], params_mat[1, 2], col = "red", cex = 3)
```



6

store runtimes

```
runtimes = matrix(0,nrow = 3, ncol = 2) #col1 = starttime, col2 = endtime
```

Run #2 as MCMC (4 chains)

```
MCMC_2 = function(Niter,Burnin,Nchains){
  chain_list = list()
  for(i in 1:Nchains){
    one_chain = MH_post(Niter = Niter,burnin = Burnin)
    chain_list[[i]] = mcmc(one_chain)
  }
  return(chain_list)
}

runtimes[1,1] = Sys.time()
chain_list_2 = MCMC_2(Niter = 10000,Burnin=5000,Nchains = 4)
runtimes[1,2] = Sys.time()
```

Run #3 as MCMC (4 chains)

```
MCMC_3 = function(Niter,Burnin,Nchains){
  chain_list = list()
  for(i in 1:Nchains){
    one_chain = MH_post_lambda(Niter = Niter,burnin = Burnin)
    one_chain = cbind(one_chain[,1],exp(one_chain[,2])) #transform back to sigsqs
    chain_list[[i]] = mcmc(one_chain)
  }
  return(chain_list)
}
```

```
runtimes[2,1] = Sys.time()
chain_list_3 = MCMC_3(Niter = 10000,Burnin=5000,Nchains = 4)
runtimes[2,2] = Sys.time()
```

Run #5 as MCMC (4 chains)

```
MCMC_5 = function(Niter,Burnin,Nchains){
  chain_list = list()
  for(i in 1:Nchains){
    params_mat = array(0, c(Niter, 2))
    params_mat[1,] = c(2,0.9) # starting point
    for(k in 2:Niter){
      params_mat[k, 1] = mu_conditional(params_mat[k-1,2])
      params_mat[k, 2] = sigsq_conditional(params_mat[k,1])
      }
    chain_list[[i]] = mcmc(params_mat[Burnin:Niter,])
  }
  return(chain_list)
}

runtimes[3,1] = Sys.time()
chain_list_5 = MCMC_5(Niter = 10000,Burnin=5000,Nchains = 4)
runtimes[3,2] = Sys.time()
```

Comparing runtimes for the different algorithms

```
colnames(runtimes) = c("Start","End")
Total_time = runtimes[,2]-runtimes[,1]
runtimes = cbind(runtimes,Total_time)

kable(runtimes)
```

Start	End	Total_time
1557428648	1557428651	3.666442
1557428651	1557428655	3.510538
1557428655	1557428655	0.479871

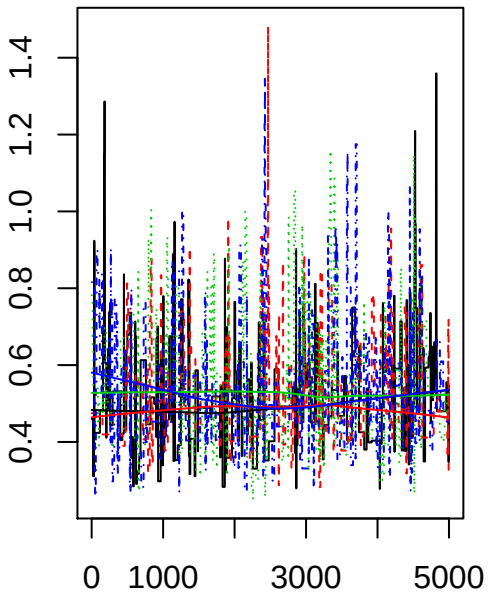
The gibbs MCMC (row 3) is much faster computationally than either of the metropolis MCMCs.

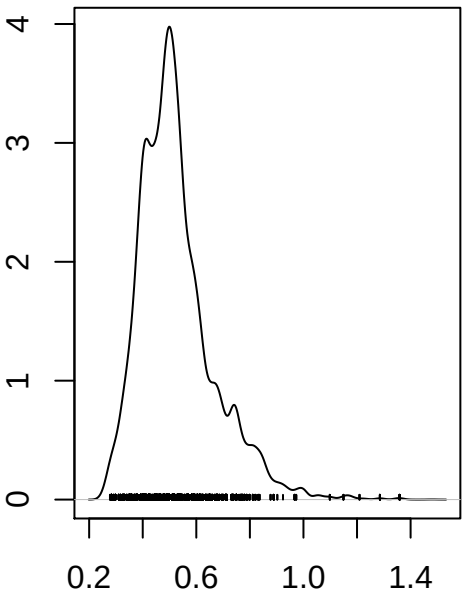
Comparing the estimation of σ^2 for the different algorithms

```
mcmc_list_2 = mcmc.list(chain_list_2[[1]][,2],chain_list_2[[2]][,2],chain_list_2[[3]][,2],chain_list_2[
mcmc_list_3 = mcmc.list(chain_list_3[[1]][,2],chain_list_3[[2]][,2],chain_list_3[[3]][,2],chain_list_3[
mcmc_list_5 = mcmc.list(chain_list_5[[1]][,2],chain_list_5[[2]][,2],chain_list_5[[3]][,2],chain_list_5[
```

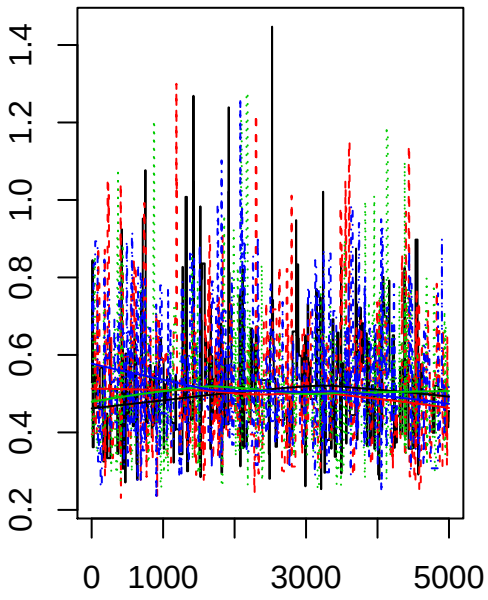
Trace plots

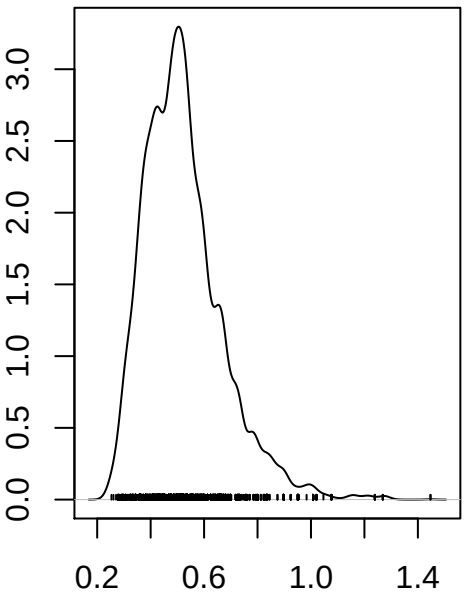
```
plot(mcmc_list_2)
```



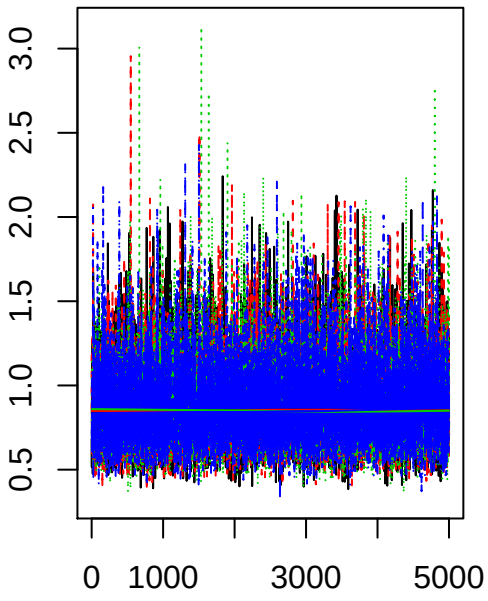


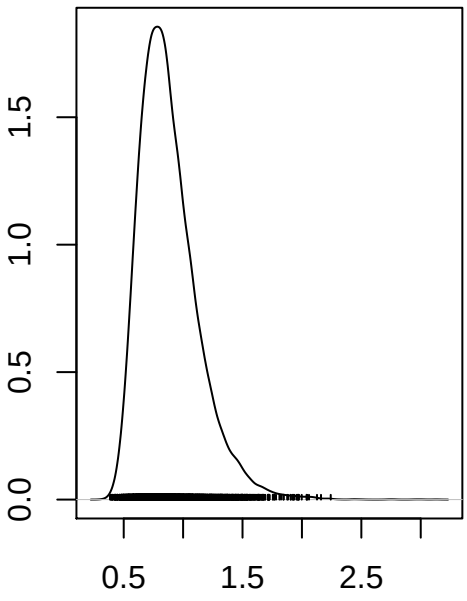

```
plot(mcmc_list_3)
```






```
plot(mcmc_list_5)
```





The gibbs algorithm has the smoothest curve, and the most overlab between the chains in the trace plots.

Gelman and Rubin's convergence diagnostic

```
gelman.diag(mcmc_list_2)
```

```
## Potential scale reduction factors:
##
##      Point est. Upper C.I.
## [1,]       1.01       1.01
```

```
gelman.diag(mcmc_list_3)
```

```
## Potential scale reduction factors:
##
##      Point est. Upper C.I.
## [1,]          1       1.01
```

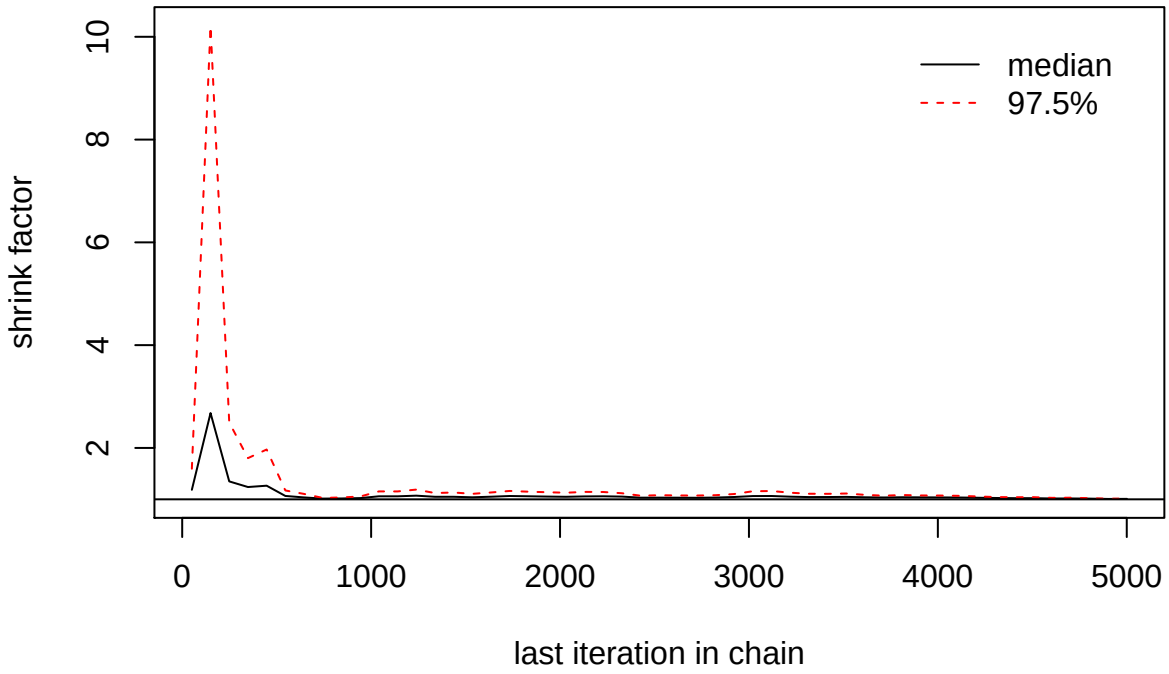
```
gelman.diag(mcmc_list_5)
```

```
## Potential scale reduction factors:
##
##      Point est. Upper C.I.
## [1,]          1          1
```

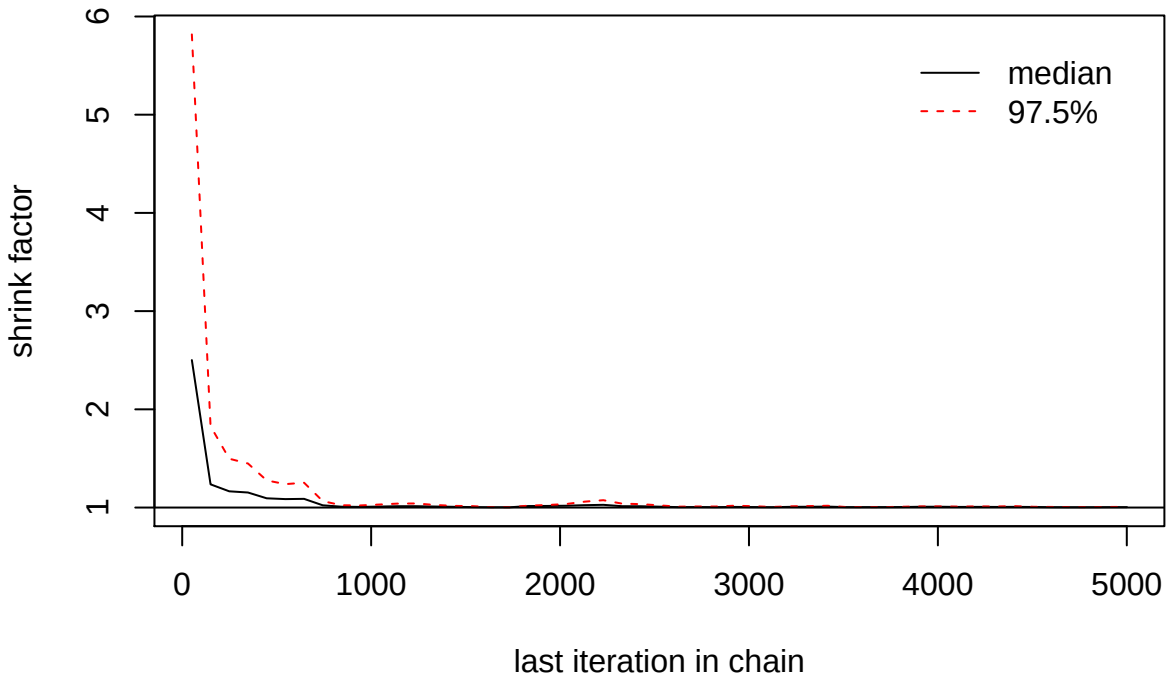
This diagnostic indicates convergence has likely been reached when the upper limit is close to 1. In all three cases, the upper limit is close to one, so I think we can say that convergence has been reached.

Gelman-Rubin-Brooks plot

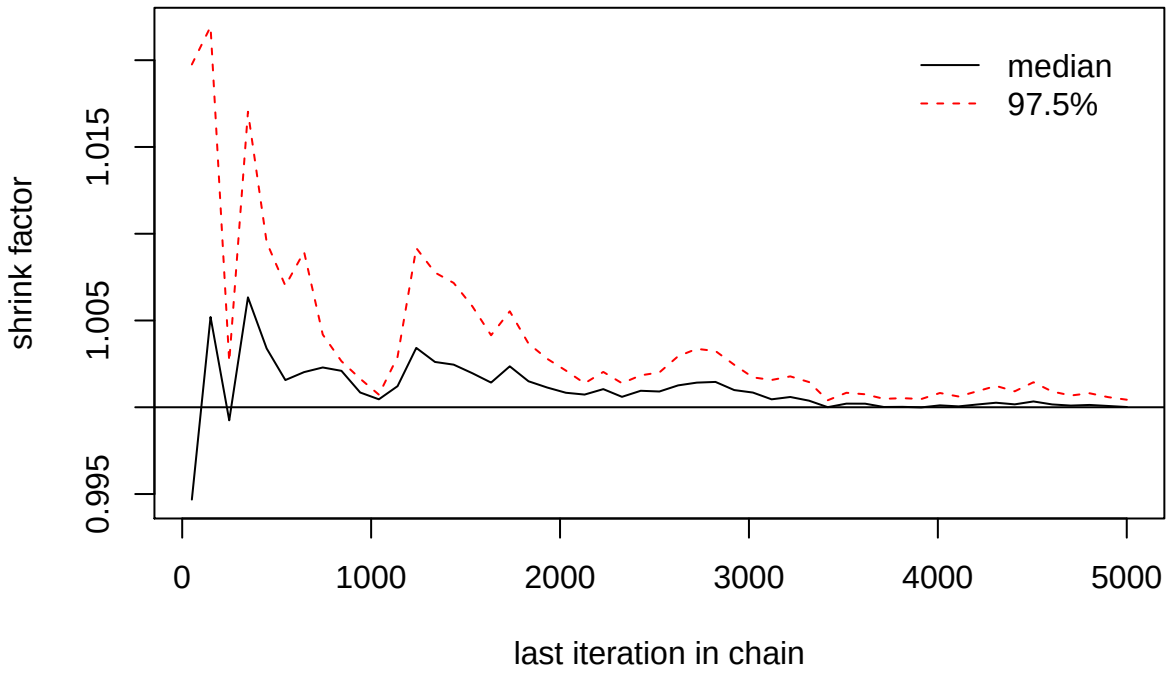
```
gelman.plot(mcmc_list_2)
```




```
gelman.plot(mcmc_list_3)
```




```
gelman.plot(mcmc_list_5)
```



This plot is another check of convergence. The shrink factor is another statistic that we want to converge along with our chains. The plot of the MCMC with the gibbs algorithm is the bumpiest of the plots. It still appears to converge, so I think convergence has been reached.

Effective sample size

```
effsizes_2 = sapply(mcmc_list_2,effectiveSize)
effsizes_2
```

```
##     var1     var1     var1     var1
## 168.2554 140.1375 124.1877 154.4740
```

```
mean(effsizes_2)
```

```
## [1] 146.7637
```

```
effsizes_3 = sapply(mcmc_list_3,effectiveSize)
effsizes_3
```

```
##     var1     var1     var1     var1
## 173.9864 201.4564 225.5159 184.9685
```

```
mean(effsizes_3)
```

```
## [1] 196.4818
```

```
effsizes_5 = sapply(mcmc_list_5,effectiveSize)
effsizes_5
```

```
##     var1     var1     var1     var1
## 4459.657 4528.542 4494.313 4805.953
```

```
mean(effsizes_5)
```

```
## [1] 4572.116
```

The MCMC runs with the metropolis algorithms have average effective sample sizes under 300 even though we ran them for 5000 iterations past burnin. This indicates that there is a lot of autocorrelation between the

draws of the algorithm. In contrast, the average effective sample size for the gibbs MCMC is close to 5000 which indicates it was drawing points that were pretty independent.

7

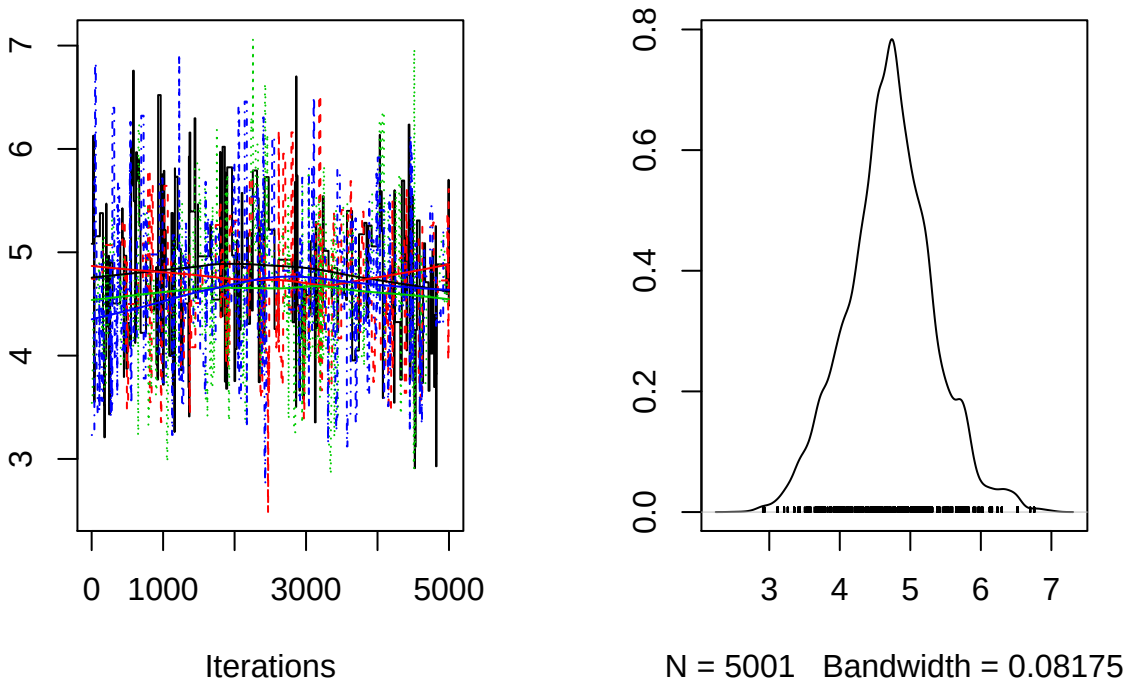
The computation time remains the same for this section of the questions. We are adjusting the statistics we are looking at, but using the same runs of the MCMC.

Comparing the estimation of μ/σ for the different algorithms

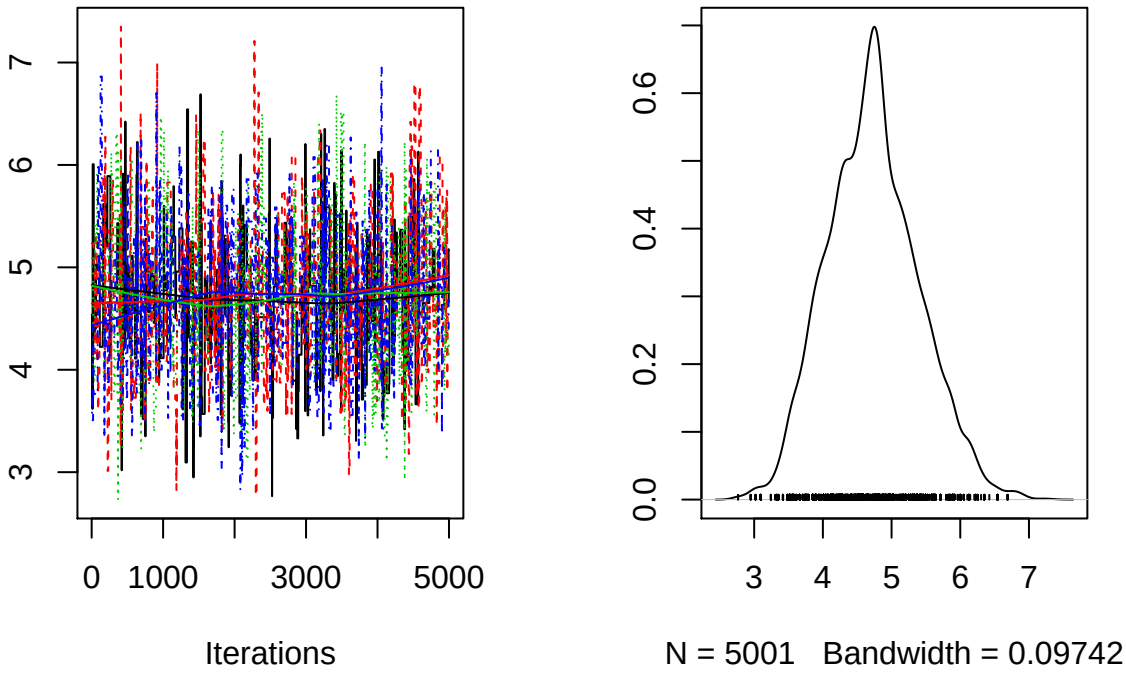
```
ratio_mcmc_list_2 = mcmc.list(chain_list_2[[1]][,1]/sqrt(chain_list_2[[1]][,2]),chain_list_2[[2]][,1]/s

ratio_mcmc_list_3 = mcmc.list(chain_list_3[[1]][,1]/sqrt(chain_list_3[[1]][,2]),chain_list_3[[2]][,1]/s

ratio_mcmc_list_5 = mcmc.list(chain_list_5[[1]][,1]/sqrt(chain_list_5[[1]][,2]),chain_list_5[[2]][,1]/s
```

Trace plots

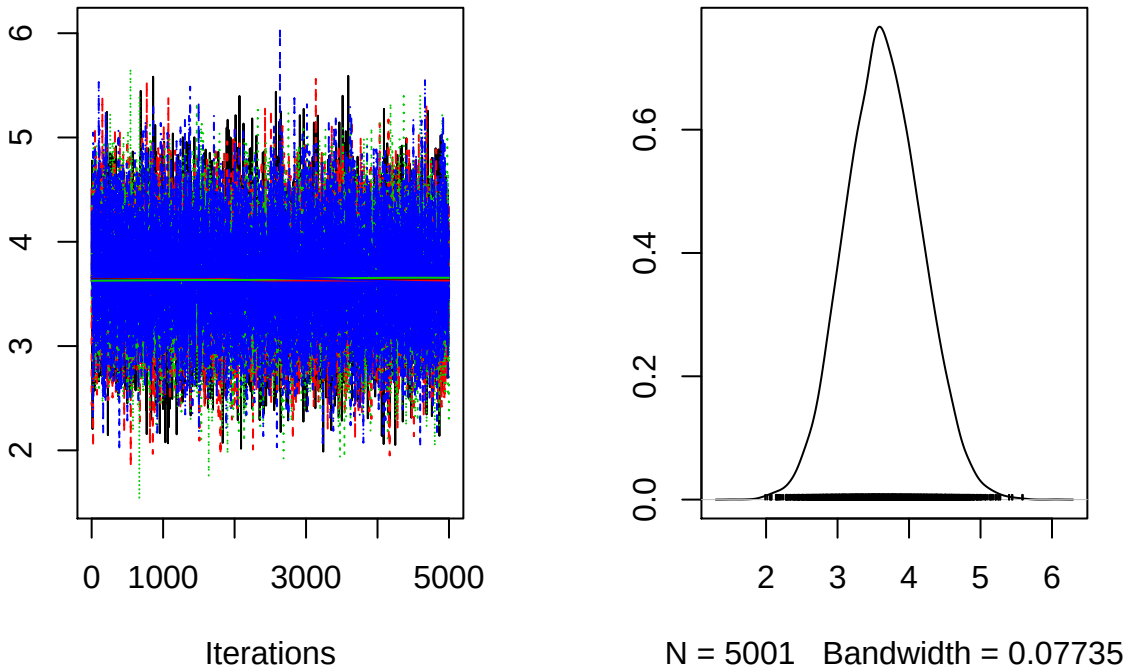
```
plot(ratio_mcmc_list_2)
```




```
plot(ratio_mcmc_list_3)
```




```
plot(ratio_mcmc_list_5)
```



Again, the gibbs algorithm has the smoothest curve, and the most overlab between the chains in the trace plots.

Gelman and Rubin's convergence diagnostic

```
gelman.diag(ratio_mcmc_list_2)
```

```
## Potential scale reduction factors:
##
##      Point est. Upper C.I.
```

```
## [1,]       1.01       1.02
```

```
gelman.diag(ratio_mcmc_list_3)
```

```
## Potential scale reduction factors:
##
##      Point est. Upper C.I.
## [1,]       1.01       1.02
```

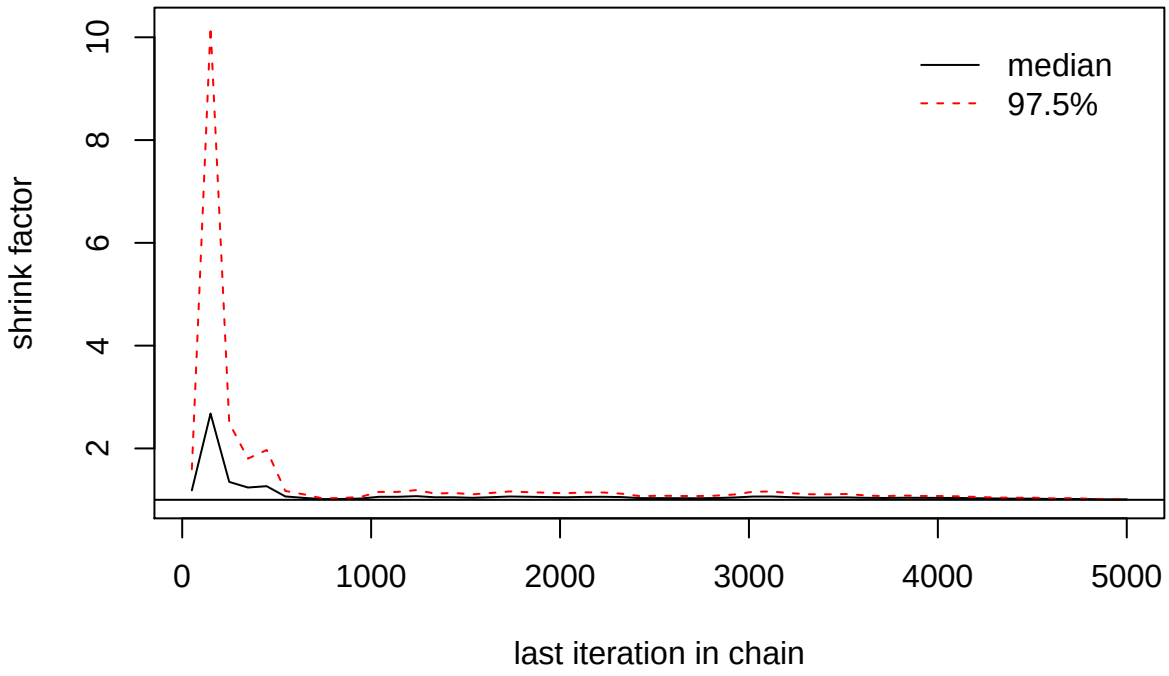
```
gelman.diag(ratio_mcmc_list_5)
```

```
## Potential scale reduction factors:
##
##      Point est. Upper C.I.
## [1,]          1          1
```

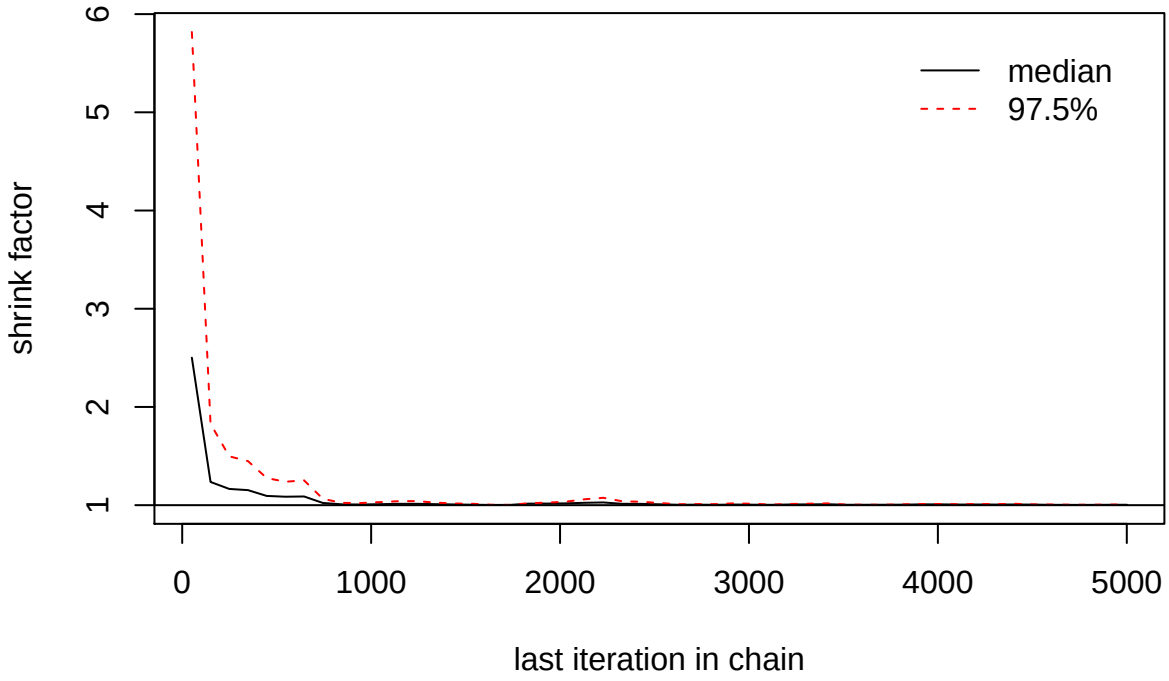
Similarly to problem 6, all three cases have an upper limit close to one, so I think we can say that convergence has been reached.

Gelman-Rubin-Brooks plot

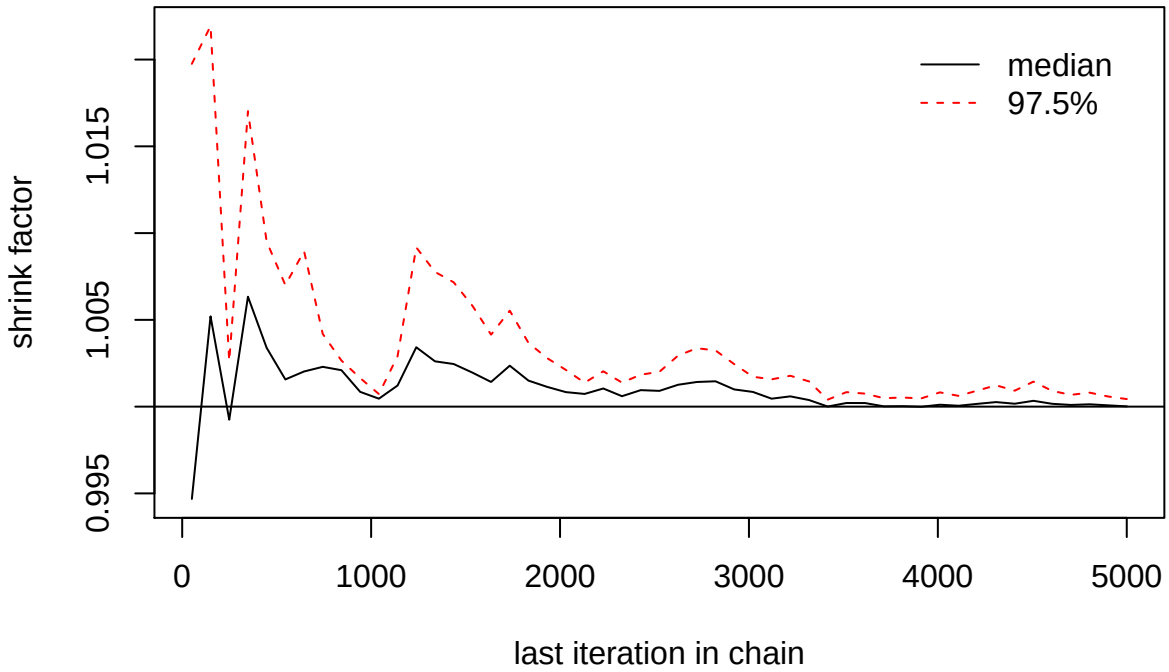
```
gelman.plot(mcmc_list_2)
```




```
gelman.plot(mcmc_list_3)
```




```
gelman.plot(mcmc_list_5)
```



This plot is another check of convergence. The shrink factor is another statistic that we want to converge along with our chains. All three are fairly smooth, so I think convergence has been reached.

Effective sample size

```
effsizes_2 = sapply(ratio_mcmc_list_2,effectiveSize)
effsizes_2
```

```
##     var1     var1     var1     var1
## 144.2497 116.6514 115.3850 142.8855
```

```r
mean(effsizes_2)
```

```
## [1] 129.7929
```

```r
effsizes_3 = sapply(ratio_mcmc_list_3,effectiveSize)
effsizes_3
```

```
##     var1     var1     var1     var1
## 191.9052 180.3575 228.7732 175.2224
```

```r
mean(effsizes_3)
```

```
## [1] 194.0646
```

```r
effsizes_5 = sapply(ratio_mcmc_list_5,effectiveSize)
effsizes_5
```

```
##     var1     var1     var1     var1
## 4601.385 4646.826 4715.763 4685.510
```

```r
mean(effsizes_5)
```

```
## [1] 4662.371
```

The effective sample sizes for the metropolis MCMC runs are a bit higher in this case, but are still much lower than the gibbs effective sample size.

My conclusions

Overall, the gibbs sampler had the fastest run time and the biggest effective sample size while performing similar estimation to the metropolis algorithms. I think this indicates that a gibbs sampler is a really powerful tool and in a scenario where more chains or more iterations were needed, the gibbs would be a more efficient tool.