SC1 Assessment 1

Sam Bowyer

Gibbs Sampling

Introduction

In the final week's content of this module we saw that we can use the Metropolis-Hastings algorithm to sample from some distribution $P(x)$ given a function f proportional to its density and a proposal/jumping distribution $Q(z|x)$ whose density $q(z|x)$ we know and from which we can generate samples. The key motivation behind this MCMC method is that P may be a distribution for which sampling is generally difficult, but using this algorithm we only need to sample from Q, which we may choose to be something simple (often a Gaussian) from which we can easily sample.

One problem that can arise, particularly in very high dimensions, is that of finding a suitable jumping distribution Q—choosing the wrong Q (which can be especially difficult if the different dimensions behave differently to one another) can lead to slow mixing times, meaning you have to generate a large number of samples before they start to accurately approximate P. A related MCMC technique known as Gibbs sampling is very commonly used in this situation. It works by sampling each dimension individually from its conditional distribution (the conditioning occurs based on the most recent samples from every other dimension), rather than attempting to sample the entire joint distribution of Q. Before we go into the details and implementation of Gibbs sampling, we will first provide a brief recap of the Metropolis-Hastings algorithm.

Metropolis-Hastings MCMC

Suppose we want to generate samples from a distribution $P(x)$ whose density is proportional to a function $f(x)$. Given a proposal distribution $Q(z|x)$ with density $q(z|x)$ the Metropolis-Hastings algorithm allows us to generate k samples from $P_{\text{MH}}(x)$ via the following algorithm:

1. Choose some arbitrary point x_1 to be the first sample.
2. For each iteration t:

- Sample a candidate point $z \sim Q(\cdot|x)$.
- Calculate the acceptance ratio

$$\alpha_{\text{MH}}(x, z) = \min\left(1, \frac{f(z)q(x|z)}{f(x)q(z|x)}\right)$$

With probability $\alpha_{\text{MH}}(x, z)$ accept the candidate and set $x_{t+1} = z$, otherwise set $x_{t+1} = x_t$.

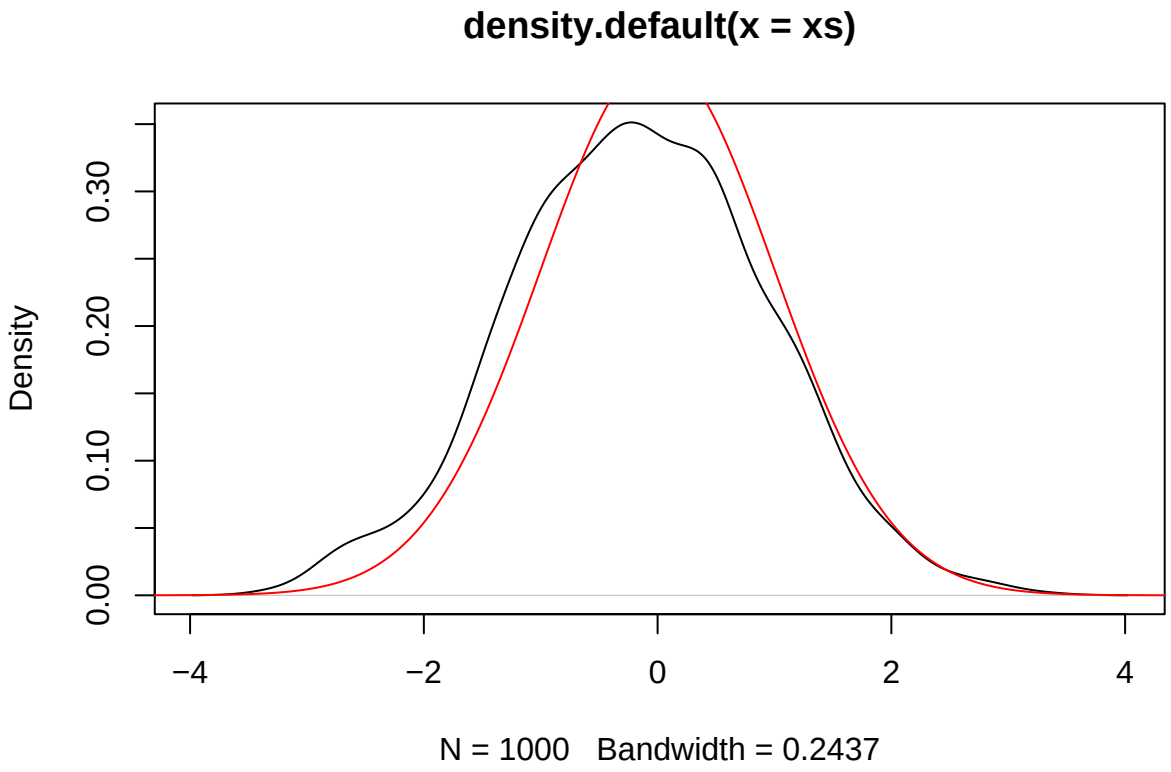
In the code below we make the common choice of using a univariate Gaussian centered at x with some standard deviation σ for our proposal distribution $Q(\cdot|x)$.

```
# Do MH-MCMC
normalProposal <- function(sigma){
  Q <- list()
  Q$sample <- function(x){
    x + sigma*rnorm(1)
  }
  Q$density <- function(x,y){
```

```
    dnorm(y-x, sd=sigma)
  }
  return(Q)
}

runMH <- function(f, Q, x0, n){
  q = Q$density
  xs = rep(0, n)
  x = x0
  for(i in 1:n){
    z = Q$sample(x)
    alpha = min(1, f(z)*q(z,x)/(f(x)*q(x,z)))
    if(runif(1)<alpha){
      x = z
    }
    xs[i] = x
  }
  return(xs)
}
```

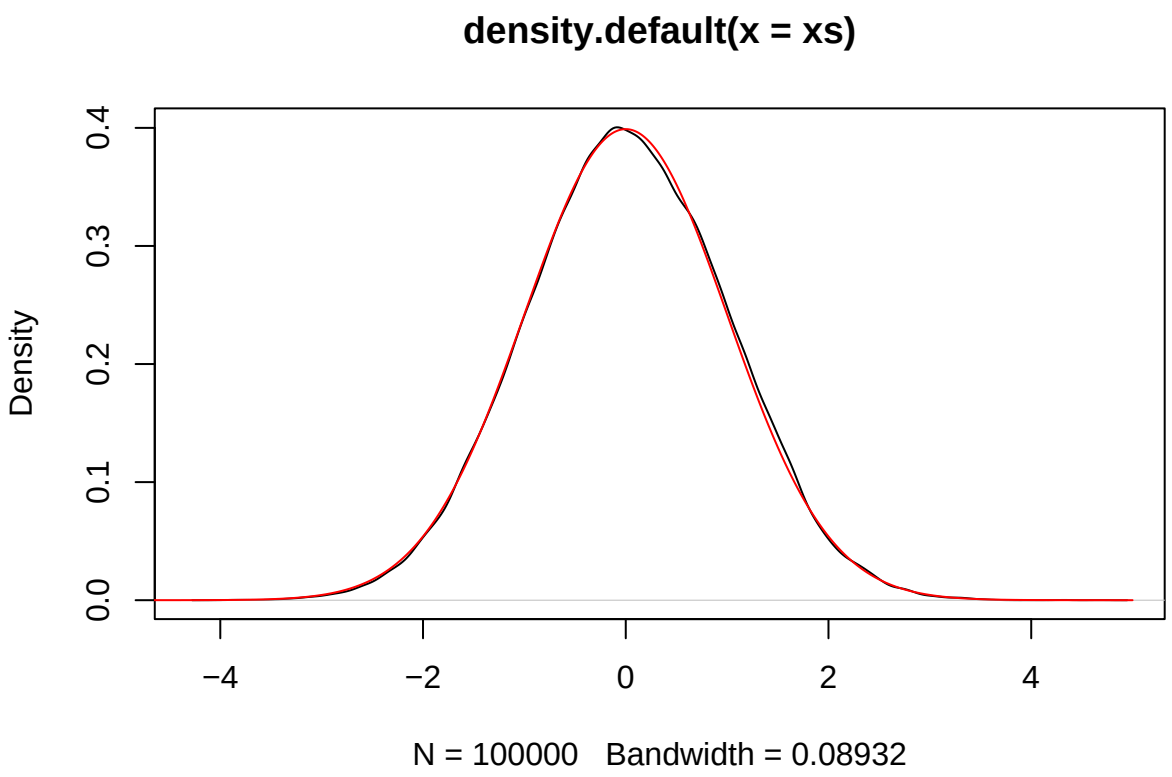
First we can check how well this fares at sampling from a standard univariate normal target distribution.

```
Q = normalProposal(1)
xs = runMH(dnorm, Q, 0, 1000)
plot(density(xs))
vs = seq(-5,5,0.01)
lines(vs, dnorm(vs), col="red")
```



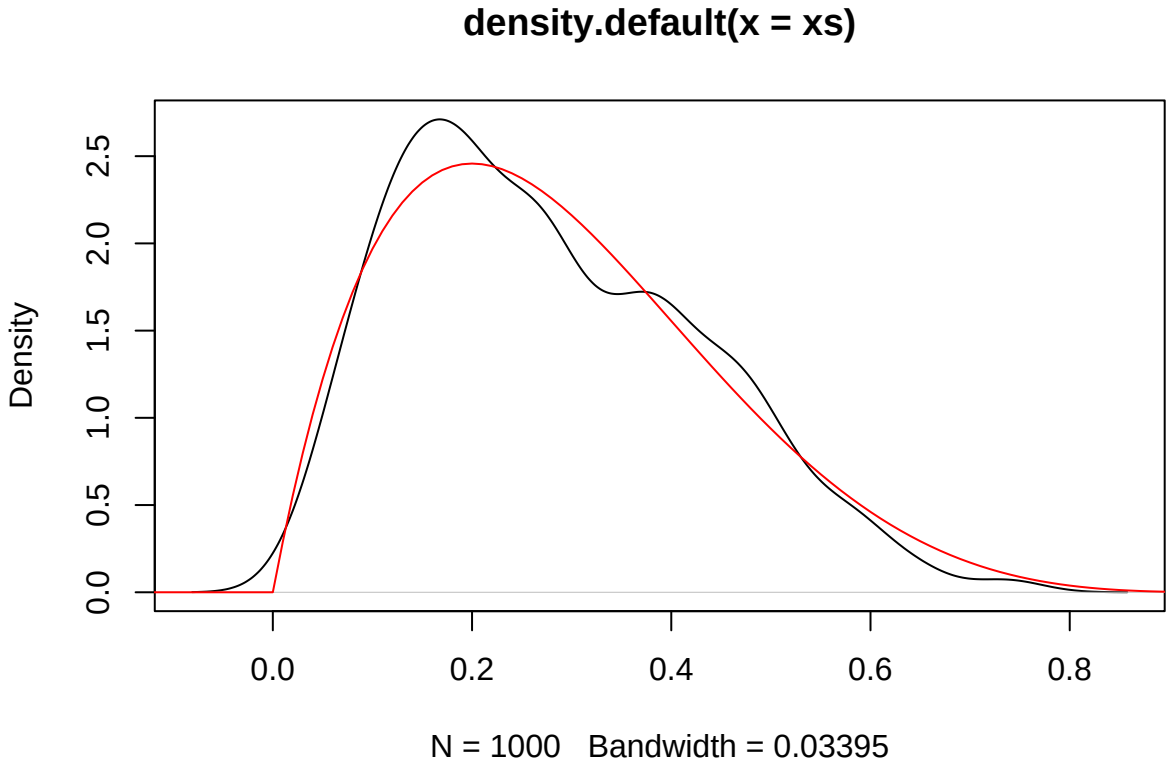
Clearly this has done fairly well, but by taking more samples we can even more accurately represent the target distribution.

```
xs = runMH(dnorm, Q, 0, 100000)
plot(density(xs))
vs = seq(-5,5,0.01)
lines(vs, dnorm(vs), col="red")
```



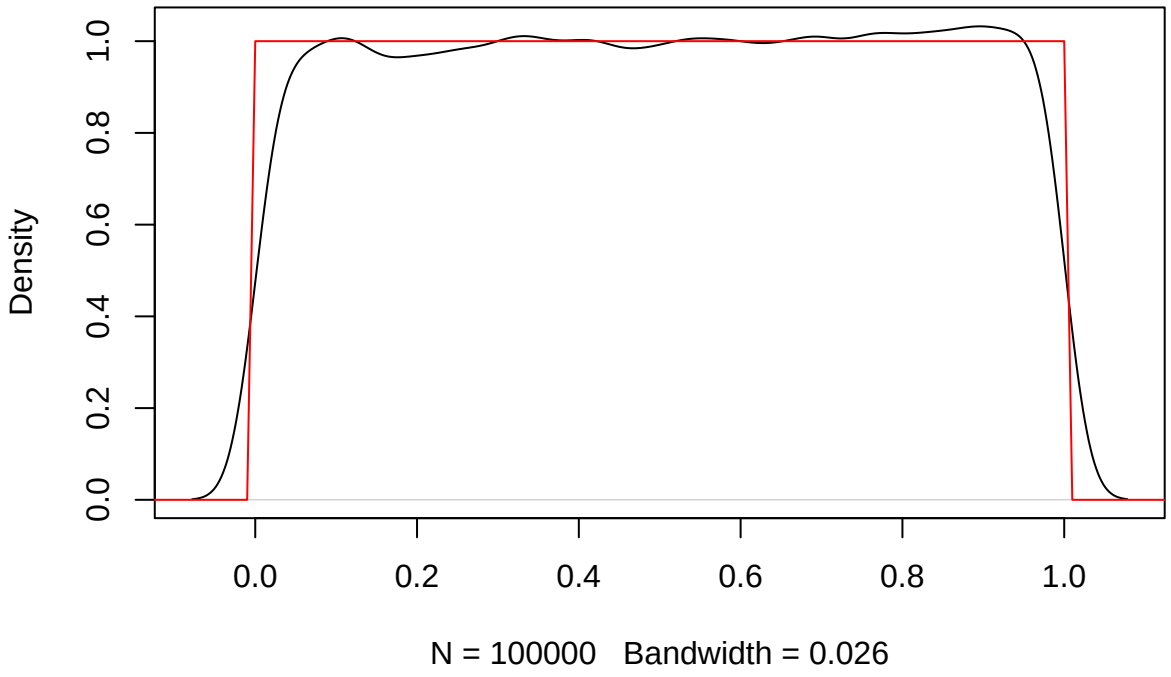
Or we can look at a more complicated distribution, here we use a Beta$(2, 5)$ distribution. (We have to be careful when using target distributions whose supports are not the entire real line (as was the case with the univariate Gaussian above). To avoid a divide-by-zero error when calculating α_{MH} we must ensure that $f(x_0) > 0$.)

```
Q = normalProposal(0.5)
xs = runMH(function(x) dbeta(x, 2, 5), Q, 0.5, 1000)
plot(density(xs))
vs = seq(-1.5,1.5,0.01)
lines(vs, dbeta(vs, 2, 5), col="red")
```

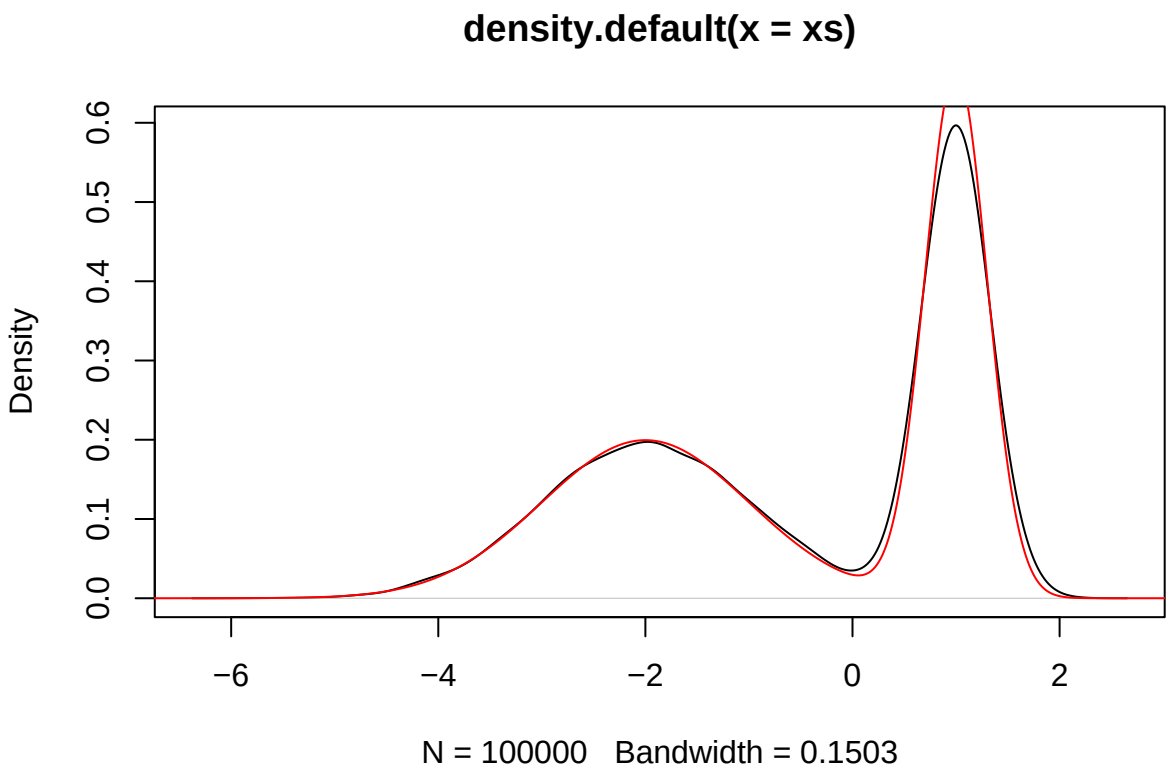


Two final interesting cases are that of the uniform distribution on $[0, 1]$ and the sum of two different univariate Guassians (note how we have changed the value of σ for the different distributions in order to get good results each time).

```
Q = normalProposal(0.25)
xs = runMH(dunif, Q, 0.5, 100000)
plot(density(xs))
vs = seq(-1.5,1.5,0.01)
lines(vs, dunif(vs), col="red")
```




```
Q = normalProposal(0.5)
f <- function(x) 0.5*(dnorm(x, -2, 1) + dnorm(x, 1, 0.3))
xs = runMH(f, Q, 0.5, 100000)
plot(density(xs))
vs = seq(-10,10,0.01)
lines(vs, f(vs), col="red")
```

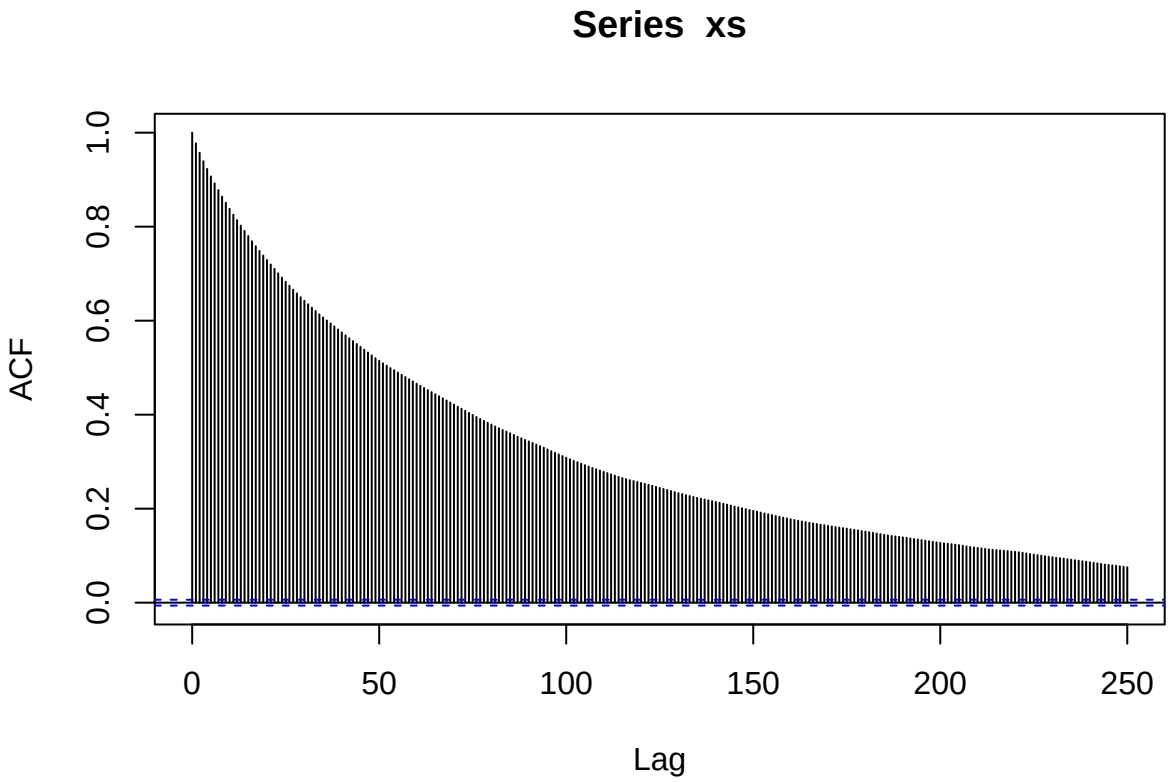


Some Considerations

There are a few important things to note about the samples generated by the Metropolis-Hastings algorithm. Firstly, it can take some time before the samples begin to accurately represent the target distribution P, and so it is very common to discard the first few samples—called the burn-in period.

Secondly, whilst we'd like to use all samples produced by the algorithm, if we want independent samples from P we shouldn't take samples that are too close to each other as they will be highly correlated. This can be seen by examining the autocorrelation function on our full set of generated functions, note that as the lag (x-axis) increases the correlation between samples that distance apart decreases.

```
acf(xs, lag.max=250)
```



Only taking every, say, 100th generated sample (after the burn-in period of, say, 1000 samples) is known as *thinning*, and whilst it may be inefficient, it can give us more accurate, less correlated samples which is often worth it if we want to simulate independent samples.

One other consideration is that when P is multivariate (particularly when it has a very high dimension), choosing the right jumping/proposal distribution Q can be very difficult, so it may be better to consider how we'd tackle this problem using the conditional distributions. This leads us onto Gibbs sampling.

Gibbs Sampling

Suppose that we want to obtain k samples of $\mathbf{x} = (x_1, ..., x_n)$ with a joint distribution $p(x_1, ..., x_n)$. First we must come up with an initial value $\mathbf{x}^{(0)} = (x_1^{(0)}, ..., x_n^{(0)})$, and then given the tth sample $\mathbf{x}^{(t)} = (x_1^{(t)}, ..., x_n^{(t)})$, to get the $(t+1)$th sample we perform the following:

- Sample $x_1^{(t+1)}$ from the conditional distribution
$$p(x_1^{(t+1)}|x_2^{(t)}, ..., x_n^{(t)}).$$
- Sample $x_2^{(t+1)}$ from the conditional distribution
$$p(x_2^{(t+1)}|x_1^{(t+1)}, x_3^{(t)} ..., x_n^{(t)}).$$
- Continue until every component of $\mathbf{x}^{(t+1)}$ has been sampled conditioned on the most recent values of all other components, that is, sample $x_i^{(t+1)}$ from
$$p(x_i^{(t+1)}|x_1^{(t+1)}, ..., x_{i-1}^{(t+1)}, x_{i+1}^{(t)}, ..., x_n^{(t)}).$$

Thus instead of having to sample Q from a very high dimension once every time step, instead we sample many simpler conditional distributions every time step—this is particularly useful if each component of $\mathbf{x}$ is conditioned only on very few other components (e.g. in most hierarchical models). We can see that in fact this basic version of Gibbs sampling is just a special case of the Metropolis-Hastings algorithm where we

accept the new proposal every time, since, denoting $x_{-i}^{(t)} = x_1^{(t+1)}, ..., x_{i-1}^{(t+1)}, x_{i+1}^{(t)}, ..., x_n^{(t)}$ and our proposal x_i^*, the proposal distribution is $p(x_i^*|x_{-i}^{(t)})$ which leads to the acceptance ratio of

$$\begin{aligned}
\alpha_{\text{MH}}(x, z) = \min\left(1, \frac{f(z)q(x|z)}{f(x)q(z|x)}\right) &= \min\left(1, \frac{p(x_i^*, x_{-i}^{(t)})p(x_i^{(t)}|x_{-i}^{(t)})}{p(x_i^{(t)}, x_{-i}^{(t)})p(x_i^*|x_{-i}^{(t)})}\right) \\
&= \min\left(1, \frac{p(x_i^*|x_{-i}^{(t)})p(x_{-i}^{(t)})p(x_i^{(t)}|x_{-i}^{(t)})}{p(x_i^{(t)}|x_{-i}^{(t)})p(x_{-i}^{(t)})p(x_i^*|x_{-i}^{(t)})}\right) \\
&= \min(1, 1) \\
&= 1.
\end{aligned}$$

Hence $x_i^{(t+1)} = x_i^*$.

```
runGibbs <- function(conditionals, x0, k){
  n = length(conditionals)

  # We will output the samples as a matrix where each row
  # represents one sample from the joint distribution
  xs = matrix(rep(0, n*(k+1)), nrow=(k+1))
  x = x0

  for (t in 2:(k+1)){
    xs[t, 1] = conditionals[[1]](xs[t-1, 2:n])

    if (n > 2){
      for (i in 2:(n-1)){
        xs[t, i] = conditionals[[i]](c(xs[t, 1:(i-1)], xs[t-1, (i+1):n]))
      }
    }

    xs[t, n] = conditionals[[n]](xs[t, 1:(n-1)])
  }
  return(xs[2:(k+1),])
}
```

We will now move onto two common examples of using Gibbs samplers, inspired by articles [1] and [2].

Example 1 - Bivariate Gaussian

In the bivariate case we can assign the first component as X and the second as Y, at which point Gibbs sampling is simply the procedure of alternately sampling from $p(Y^{(t)}|X^{(t)})$ and $p(X^{(t+1)}|Y^{(t)})$ (given some initial $X^{(0)}$—we can sample $Y^{(0)}$ using this, however, in keeping with the more general code we've written above, we will supply both $X^{(0)}$ and $Y^{(0)}$ to our Gibbs sampler) to create a sequence of pairs $(X^{(0)}, Y^{(0)}), (X^{(1)}, Y^{(1)}), ...$ forming a Markov chain with stationary distribution $p(X, Y)$.

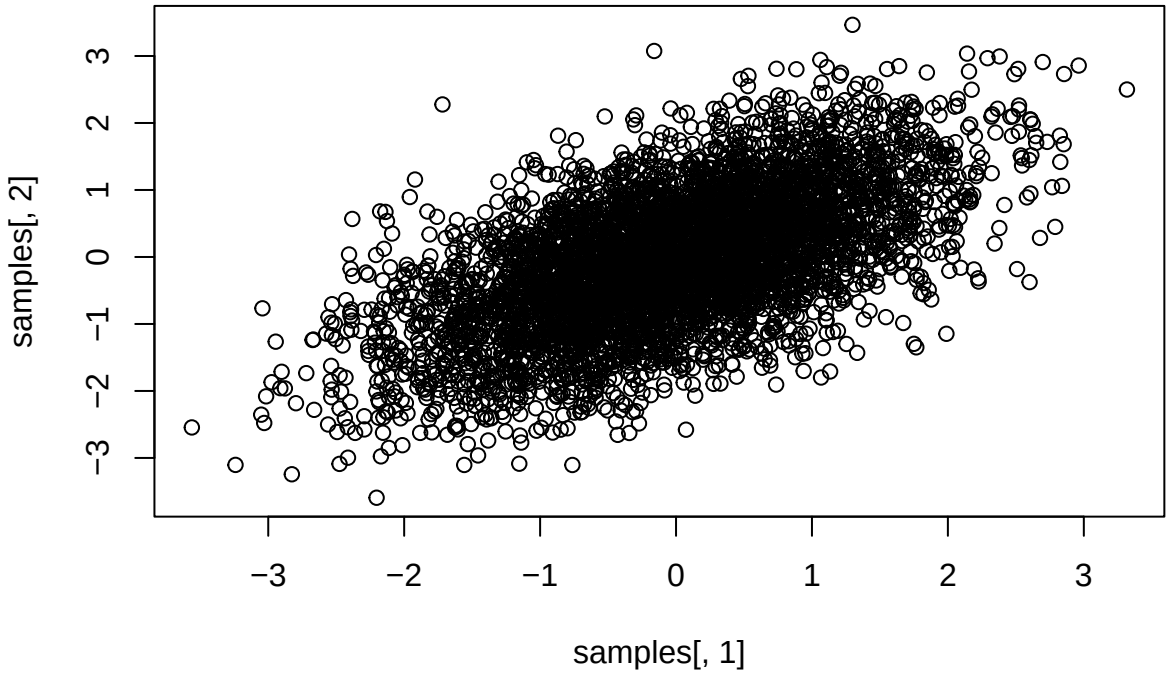
Here we consider the case where (X, Y) is distributed as a bivariate normal, each with zero mean and unit variance, but with a correlation $\rho = 0.6$. This leads to the conditional distributions:

$$X|Y \sim \mathcal{N}(\rho Y, \sigma)$$

$$Y|X \sim \mathcal{N}(\rho X, \sigma)$$

where $\sigma = \sqrt{1 - \rho^2}$.

```
rho = 0.6
sigma = sqrt(1-rho^2)

x_given_y <- function(y) rnorm(1, rho*y, sigma)
y_given_x <- function(x) rnorm(1, rho*x, sigma)

samples = runGibbs(c(x_given_y, y_given_x), c(0,0), 5000)
plot(samples[,1], samples[,2])
```




```
c(mean(samples[,1]), mean(samples[,2])); c(sd(samples[,1]), sd(samples[,2]))
```

```
## [1] -0.03229382 -0.04610127
```

```
## [1] 0.9866246 1.0123817
```

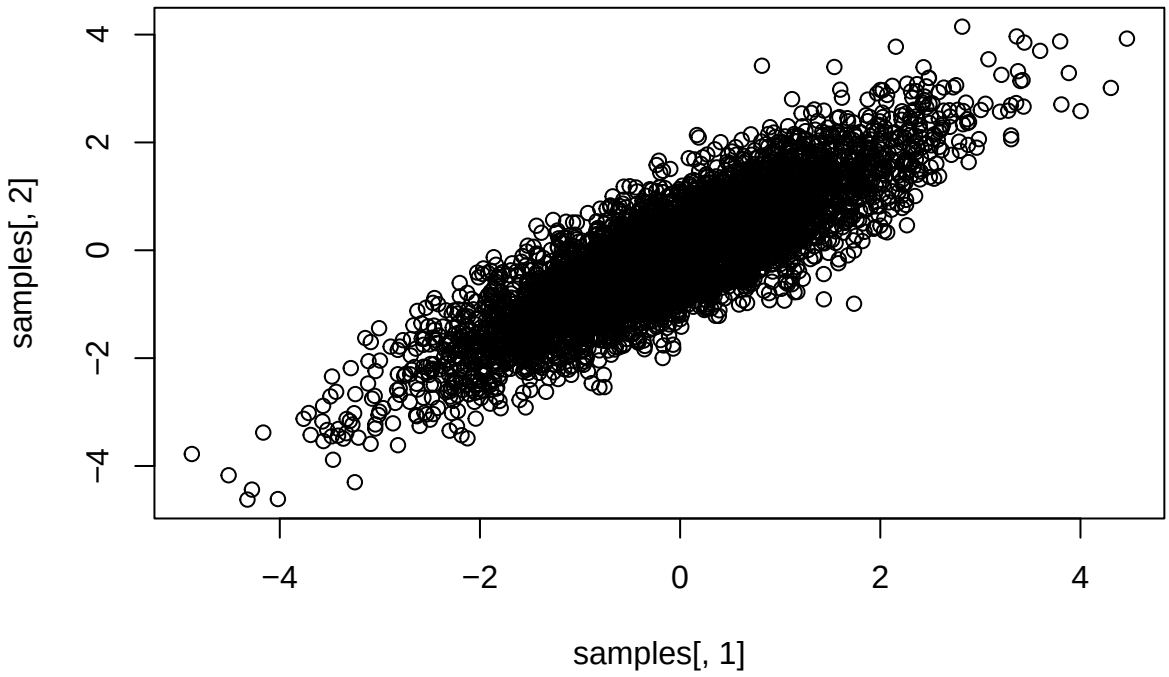
```
cor(samples[,1], samples[,2])
```

```
## [1] 0.5996558
```

We can see that the means and standard deviations of the samples of X and Y are close to the true values, as is the correlation between the two.

Note that we also get similar results if we use our Metropolis-Hastings algorithm to form the conditional distributions, however, this is significantly improved when we include a burn in period of 100 rather than 10 (even though the longer burn-in period will be worse for efficiency).

```
Q = normalProposal(0.25)

# Choose the 11th sample to allow for a 10-sample burn-in
x_given_y <- function(y) runMH(function(x) dnorm(x, rho*y, sigma), Q, y, 11)[11]
y_given_x <- function(x) runMH(function(y) dnorm(y, rho*x, sigma), Q, x, 11)[11]

samples = runGibbs(c(x_given_y, y_given_x), c(0,0), 5000)
plot(samples[,1], samples[,2])
```




```
c(mean(samples[,1]), mean(samples[,2])); c(sd(samples[,1]), sd(samples[,2]))
```

```
## [1] -0.05500824 -0.05769101
```

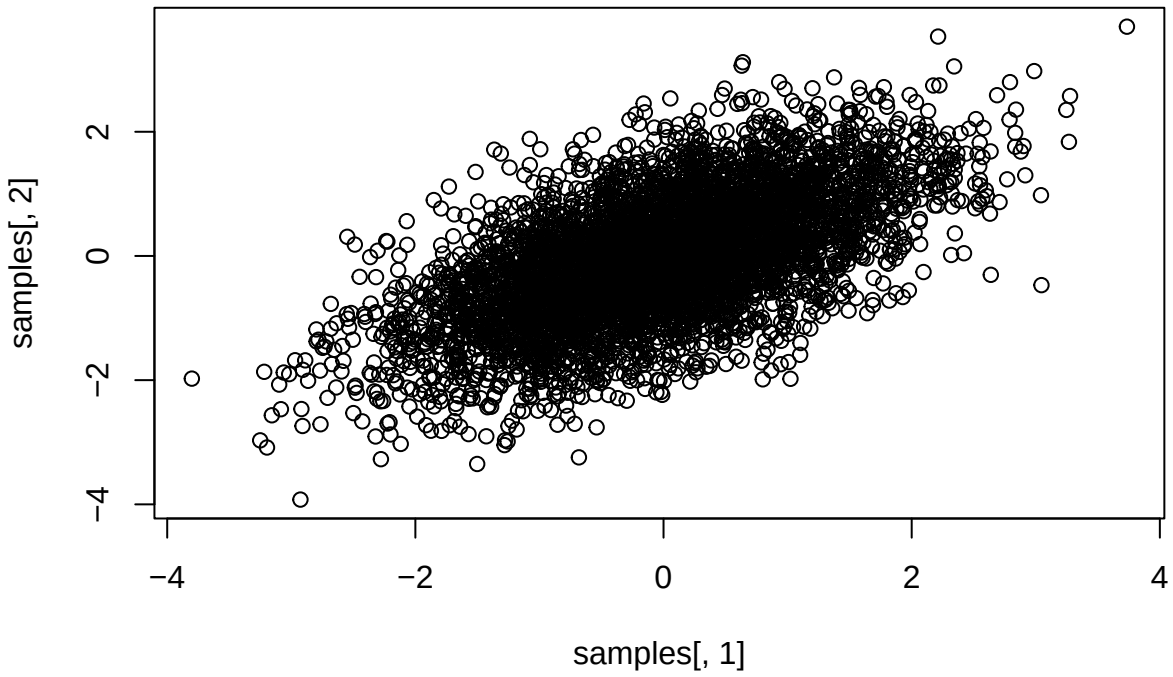
```
## [1] 1.15452 1.14984
```

```
cor(samples[,1], samples[,2])
```

```
## [1] 0.8474292
```

Now with a 100-sample burn-in period we see much better results in the plot, sample mean, standard deviation and correlation coefficient.

```
# Choose the 101st sample to allow for a 100-sample burn-in
x_given_y <- function(y) runMH(function(x) dnorm(x, rho*y, sigma), Q, y, 101)[101]
y_given_x <- function(x) runMH(function(y) dnorm(y, rho*x, sigma), Q, x, 101)[101]

samples = runGibbs(c(x_given_y, y_given_x), c(0,0), 5000)
plot(samples[,1], samples[,2])
```




```
c(mean(samples[,1]), mean(samples[,2])); c(sd(samples[,1]), sd(samples[,2]))
```

```
## [1] -0.04802877 -0.04709324
```

```
## [1] 1.008521 1.005978
```

```
cor(samples[,1], samples[,2])
```

```
## [1] 0.6071763
```

Example 2 - Hierarchical Model

The other example we will discuss is that of a Bayesian hierarchical model, for which it is very common to use Gibbs sampling. Specifically, suppose we have the data $y_1, ..., y_n$ giving us the number of failures of n pieces of equipment, along with the times t_i at which each piece of equipment was observed. We want to model with a Poisson likelihood where the ith piece of equipment has λ_i expected number of failures per unit time. This likelihood is

$$\prod_{i=1}^{n} \text{Poisson}(\lambda_i t_i)$$

(note that we're scaling the expected number of failures by the observation times). Next suppose that we put a prior on each λ_i:

$$\lambda_i \sim \text{Gamma}(\alpha, \beta)$$

with $\alpha = 1.2$, and let us put the following hyperprior on β:

$$\beta \sim \text{Gamma}(\gamma, \delta)$$

with $\gamma = 0.02$ and $\delta = 1.1$. Hence we have $n + 1$ unknown parameters $(\lambda_1, ..., \lambda_n, \beta)$ with the posterior

$$p(\lambda, \beta | \mathbf{y}, \mathbf{t}) = \prod_{i=1}^{n} \text{Poisson}(\lambda_i t_i)\text{Gamma}(\alpha, \beta)\text{Gamma}(\gamma, \delta).$$

This leads to the following conditional distributions which we will use in our Gibbs sampler (we denote $\lambda_{-i} = (\lambda_1, .., \lambda_{i-1}, \lambda_{i+1}, ..., \lambda_n)$):

$$p(\lambda_i|\lambda_{-i},\beta,\mathbf{y},\mathbf{t}) = \text{Gamma}(y_i+\alpha, t_i+\beta)$$
$$p(\beta|\lambda,\mathbf{y},\mathbf{t}) = \text{Gamma}(n\alpha+\gamma, \delta+\sum_{i=1}^{n}\lambda_i).$$

First we shall generate some data $\mathbf{y},\mathbf{t}$ with $n=5$ with the true value of β set to 1 and with the observation times given as

$$\mathbf{t} = (t_1,...,t_5) = (20,40,60,80,100).$$

```
# known values
n = 5
alpha = 1.2
gamma = 0.02
delta = 1.1

ts = seq(20,100,20)

# true values to be estimated
beta = 1
lambdas = rgamma(n, alpha, rate=beta)
lambdas
```

```
## [1] 0.9573291 0.3951090 1.1187935 0.5024527 0.8425616
```

```
# generate data y
ys = rep(0, n)
for (i in 1:n){
  ys[i] = rpois(1, lambdas[i]*ts[i])
}
```

Now we can implement sampling from the conditional distributions.

```
lambdaConditional <- function(paramEstimates, i) {
  betaEstimate = paramEstimates[length(paramEstimates)]
  rgamma(1, ys[i]+alpha, ts[i]+betaEstimate)
}

betaConditional <- function(paramEstimates) {
  lambdasEstimate = paramEstimates[-length(paramEstimates)]
  rgamma(1, n*alpha + gamma, delta + sum(lambdasEstimate))
}
```

Finally we can implement Gibbs sampling and see how well the samples line up with the true values of the parameters $\lambda_1,...,\lambda_9,\beta$.

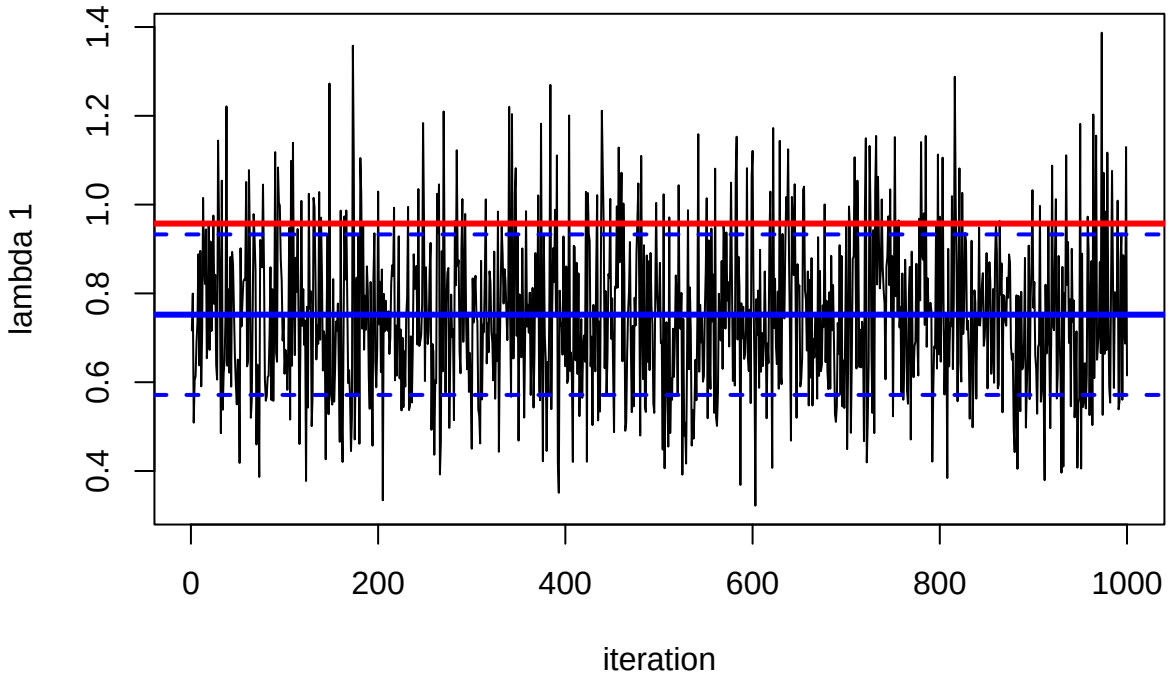
```
k = 1000
conditionals = c(function(xs) lambdaConditional(xs, 1),
                 function(xs) lambdaConditional(xs, 2),
                 function(xs) lambdaConditional(xs, 3),
                 function(xs) lambdaConditional(xs, 4),
                 function(xs) lambdaConditional(xs, 5),
                 betaConditional)

samples = runGibbs(conditionals, c(ys/ts, betaConditional(ys/ts)), k)

paramNames = c()
```

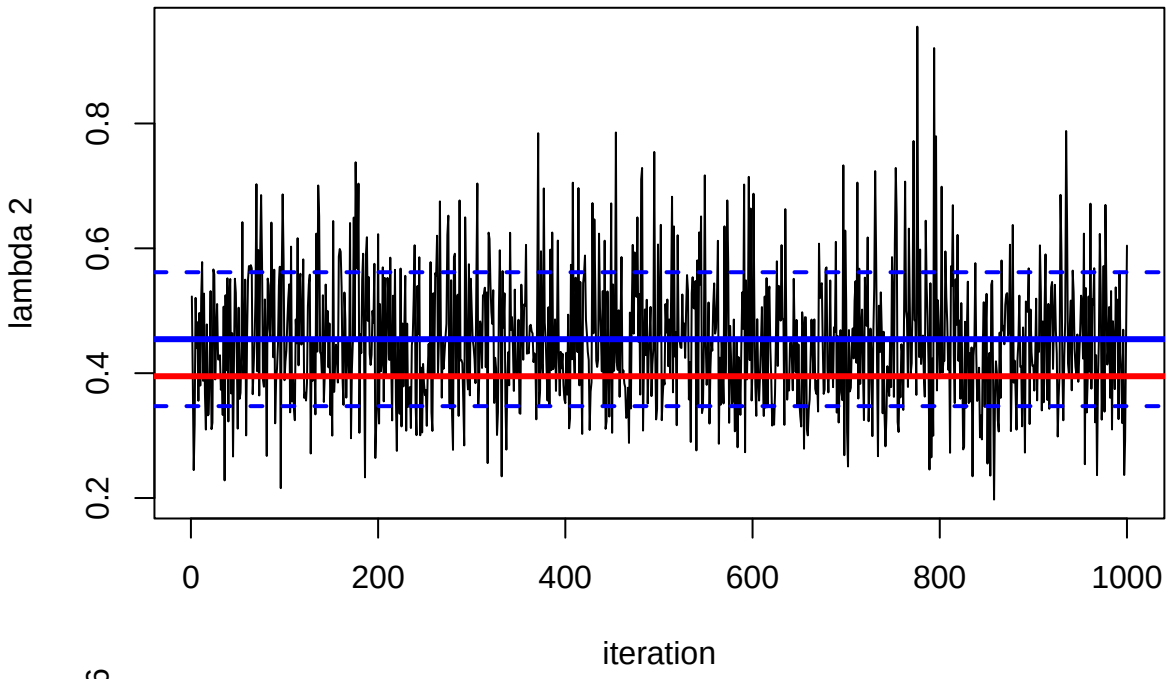
```
for (i in 1:n){
  paramNames = c(paramNames, paste("lambda", i))
}
paramNames = c(paramNames,"beta")

trueParams = c(lambdas, beta)

results = cbind(trueParams, apply(samples, 2, mean), apply(samples, 2, sd))
dimnames(results)=list(paramNames, c("true value", "sample mean", "sample sd"))
results
```

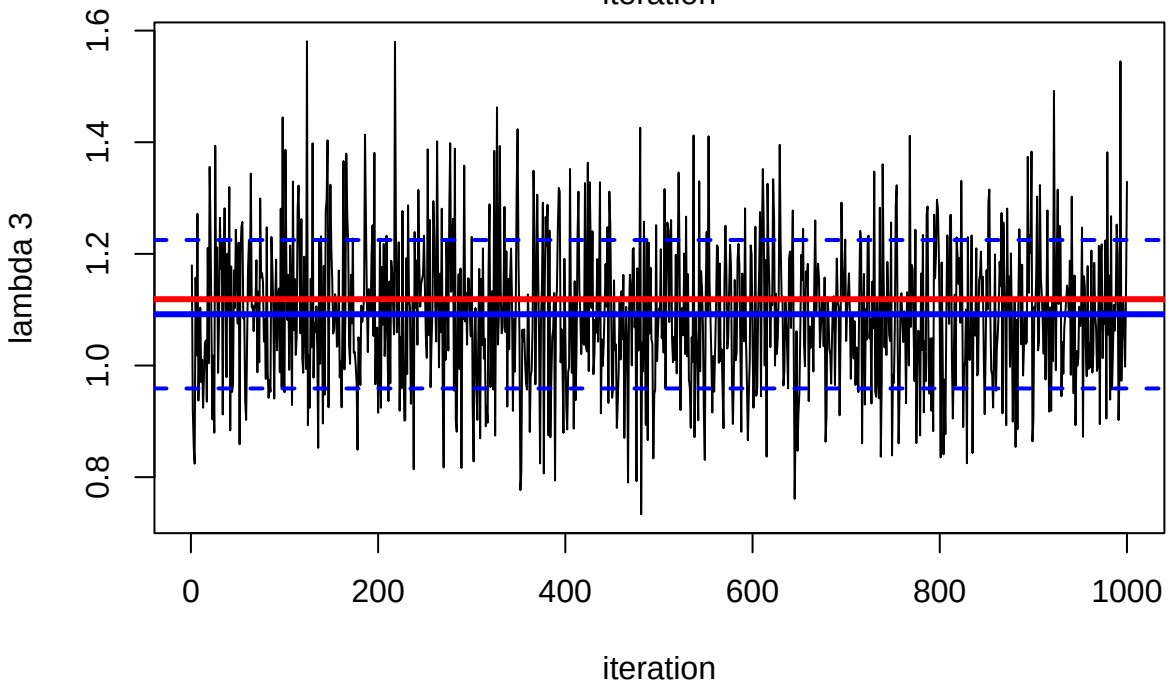
```
##          true value sample mean  sample sd
## lambda 1  0.9573291   0.7520922 0.18076055
## lambda 2  0.3951090   0.4544543 0.10727260
## lambda 3  1.1187935   1.0918686 0.13297286
## lambda 4  0.5024527   0.5072006 0.07982294
## lambda 5  0.8425616   1.0092992 0.10263309
## beta      1.0000000   1.5609853 0.65143453
```

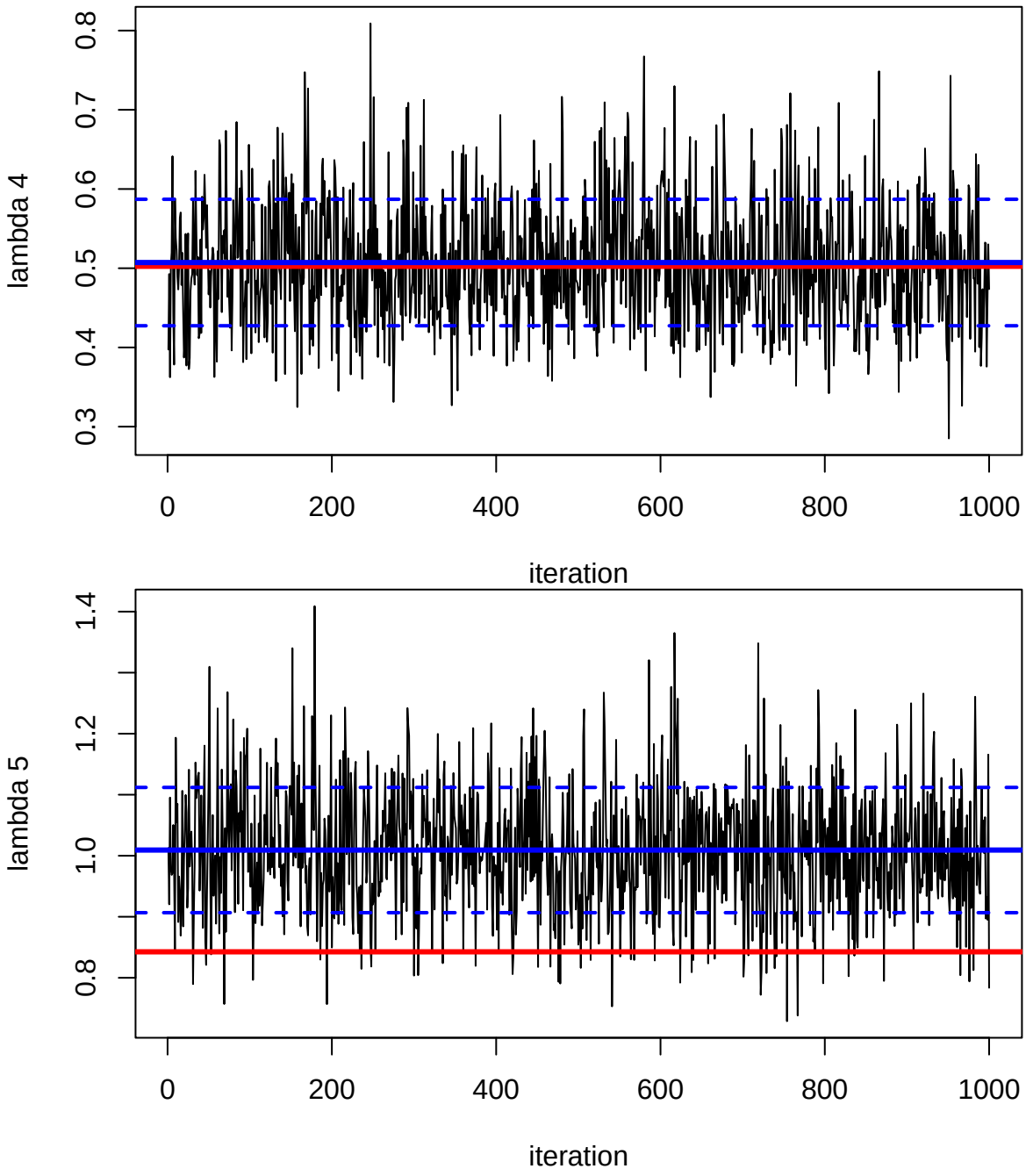
Whilst some of these estimates are quite accurate, many of them are far off their true values, and the standard deviation of the samples is very high, which is perhaps due to the small amount of data and the low number of samples generated which may have not been enough for the mixing time of the Gibbs sampler's Markov chain. Below we present the samples as a time series with the true parameter value as a red line and the sample mean as a blue line with dotted blue lines indicating one standard deviation away from this mean.

```
for (i in 1:n){
  plot(1:k, samples[,i], xlab="iteration", ylab=paste("lambda", i), type="l")
  abline(h=lambdas[i], col="red", lwd=3)
  abline(h=results[i,2], col="blue", lwd=3)

  abline(h=results[i,2]+results[i,3], col="blue", lty=2, lwd=2)
  abline(h=results[i,2]-results[i,3], col="blue", lty=2, lwd=2)
}
```

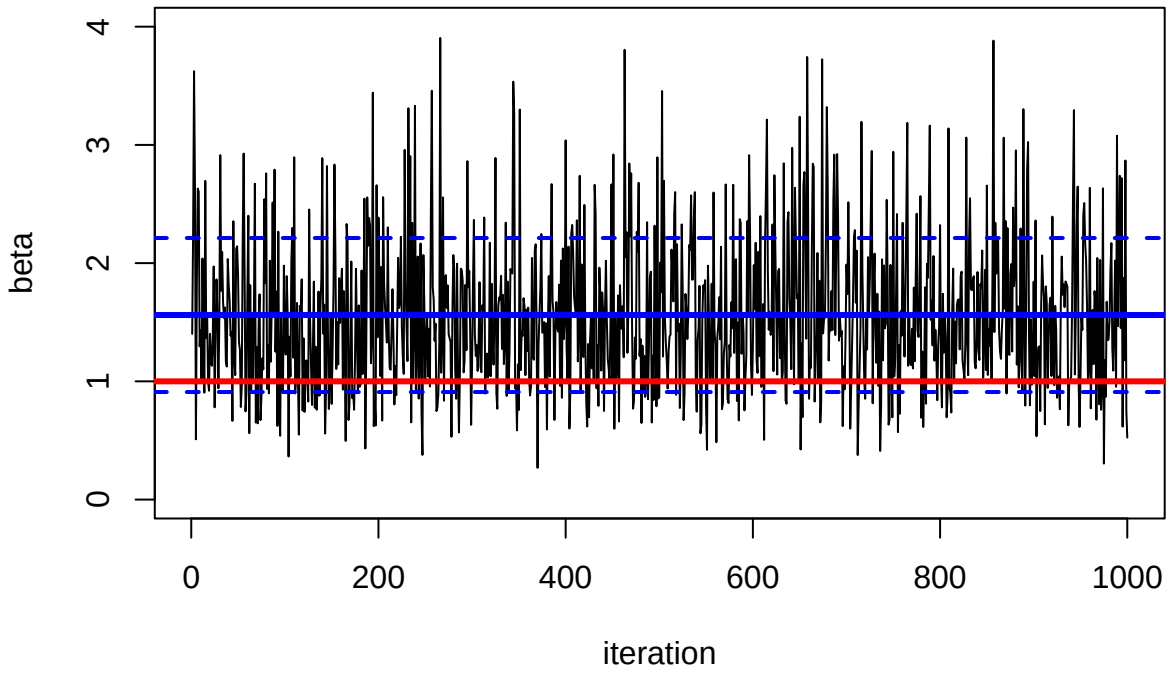



lambda 2
0.2
0.4
0.6
0.8
0
200
400
600
800
1000
iteration


lambda 3
0.8
1.0
1.2
1.4
1.6
0
200
400
600
800
1000
iteration




```
plot(1:k, samples[,n+1], xlab="iteration", ylab="beta", ylim=c(0,4), type="l")
abline(h=beta, col="red", lwd=3)
abline(h=results[n+1,2], col="blue", lwd=3)

abline(h=results[n+1,2]+results[n+1,3], col="blue", lty=2, lwd=2)
abline(h=results[n+1,2]-results[n+1,3], col="blue", lty=2, lwd=2)
```



References

[1] G. Larangeira, "Three Simple Applications of Markov Chains and Gibbs Sampling." [Online]. Available: http://rstudio-pubs-static.s3.amazonaws.com/279858_010f9da7c8d744988019397e3fe51cb2.html

[2] C. Chan and J. McCarthy, "Metropolis and Gibbs Sampling — Computational Statistics in Python," 2016. [Online]. Available: https://people.duke.edu/~ccc14/sta-663-2016/16A_MCMC.html#The-Gibbs-sampler