SMSTC (2018/19)

Statistics

Lecture 19: MCMC Methods 4: MCMC within R

Valentin Popov
School of Mathematics and Statistics
University of St Andrews

www.smstc.ac.uk

Contents

19.1 Introduction

It is often possible to use WinBUGS to obtain estimates of posterior distributions of interest. However, WinBUGS is not a panacea for all implementations of MCMC. For example, simulations can sometimes require an infeasible amount of time within WinBUGS. In these circumstances it can be necessary to write bespoke code in an alternative language. We consider the writing of MCMC code within the language R. This allows a greater deal of flexibility in the updating algorithms that are used within the MCMC simulations. In addition, R code typically runs much faster than WinBUGS.

19.2 Simulating random variates in R

The ability to simulate random variables from given distributions is very important within MCMC simulations. R has intrinsic functions that allows us to generate observations from a large number of common statistical distributions. Possible distributions include:

`rnorm` (Normal);

`runif` (Uniform);

`rgamma` (Gamma);

`rbeta` (Beta);

`rbinom` (Binomial);

`rpois` (Poisson)

The first input parameter to all such functions is the number of random deviates to be simulated. The additional input parameters are those of the distribution. Note that the input parameter values for the Normal distribution is the mean and *standard deviation* within R. For example the command `rnorm(1,2,0.5)` will simulate 1 random deviate from a $N(2, 0.25)$ distribution.

Alternatively (as we will see with the Metropolis-Hastings algorithm) we will often want to evaluate the probability mass or density function of given distributions at particular values. These can be explicitly written in R, but it is often easier to use the intrinsic functions in R. These are similar to the above, but uses an "`d`" instead of an "`r`". For example `dnorm` instead of `rnorm`. For example the command `dnorm(2.5,2,0.5)` will evaluate the probability density function of a $N(02, 0.25)$ distribution at the value 2.5.

Use `help("name")` for further information.

19.3 Simple Example: Gibbs Sampler

Consider the simple univariate example in Section 17.3.1 where we observe data $x_1, \ldots, x_n$ from a $N(\mu, \tau^{-1})$ distribution with μ and τ unknown. We specify the priors,

$$\begin{aligned} \mu &\sim N(\mu_0, \sigma_0^2); \\ \tau &\sim \Gamma(\alpha_0, \beta_0). \end{aligned}$$

The posterior conditional distributions for μ and τ are:

$$\pi(\mu|\tau, \boldsymbol{x}) \sim N\left(\frac{n\bar{x}\tau + \mu_0\tau_0}{n\tau + \tau_0}, (n\tau + \tau_0)^{-1}\right)$$

where $\tau_0 = (\sigma_0^2)^{-1}$; and

$$\pi(\tau|\mu, \boldsymbol{x}) \sim \Gamma\left(\alpha_0 + \frac{n}{2}, \beta_0 + \frac{S_n}{2}\right),$$

where $S_n = \sum_{i=1}^{n}(x_i - \mu)^2$.
We set $n = 100$ and generate 100 observations from a $N(0,1)$ distribution, so that $\mu = 0$ and $\tau = 1$. We use the following R routine to implement the Gibbs sampler for this example:

```
gibbs <- function(nits,x,mu,tau) {

    alpha0 <- 0.1              # Set prior parameter values
    beta0 <- 0.01
    mu0 <- 0
    tau0 <- 1

    param <- array(0,c(nits+1,2))

    param[1,1] <- mu         # Set initial parameter values
    param[1,2] <- tau

    n <- length(x)
    xbar <- mean(x)

    for (t in 2:(nits+1)) { # Perform MCMC updates using Gibbs

        mumean <- (n*xbar*param[t-1,2] + mu0*tau0)/
                        (n*param[t-1,2] + tau0)

        muvar <- 1/(n*param[t-1,2] + tau0)

        param[t,1] <- rnorm(1,mumean, sqrt(muvar))

        sn <- sum((x-param[t,1])^2)

        taualpha <- alpha0 + n/2
        taubeta <- beta0 + sn/2

        param[t,2] <- rgamma(1,taualpha,taubeta)
    }

    par(mfrow=c(2,1))

    plot(1:length(param[,1]),param[,1],type="l",xlab="t",ylab="mu")
    plot(1:length(param[,2]),param[,2],type="l",xlab="t",ylab="tau")

param

}
```

Then to simulate data and run the Gibbs sampler we use the commands:

```
x <- rnorm(100)
out <- gibbs(1000,x,5,0.5)
```

This code "Gibbs.R" is downloadable from the SMSTC website.

Exercise: Decide what is a suitable burn-in and obtain posterior summary statistics/density estimate plots for the parameters μ and τ.

19.4 Simple Example: Metropolis-Hastings

We consider the same example as above, but use the Metropolis-Hastings random walk to update the parameters within the MCMC algorithm, instead of the Gibbs sampler (the following code, "MHRW.R" is provided on the website).

```
met.has <- function(nits,x,mu,tau) {

    alpha0 <- 0.001             # Set prior parameter values
    beta0 <- 0.001
    mu0 <- 0
    tau0 <- 1

    param1 <- array(0,c(nits+1,nparam))
    param <- array(0,nparam)
    newparam <- array(0,nparam)

    param[1] <- mu          # Set current value of chain
    param[2] <- tau
    param1[1,1] <- mu         # Record values at each iteration
    param1[1,2] <- tau

    n <- length(x)
    xbar <- mean(x)

    for (t in 2:(nits+1)) { # Perform MCMC updates using MH

        for (i in 1:2) {    # Cycle through each parameter

            for (j in 1:2) {
                newparam[j] <- param[j]    # Record current parameter value
            }

            u <- rnorm(1,0,0.1)   # Simulate from Normal distribution

            newparam[i] <- param[i] + u     # Set proposed parameter value

            if (newparam[2] >= 0) {         # Need variance > 0

#   To calculate the log(numerator) of the acceptance probability
#   (ignoring constants)
#   Likelihood term (in simplified form):

                sn <- sum((x-newparam[1])^2)

                num <- n/2*log(newparam[2]) - newparam[2]*sn/2

#   Prior terms:
```

```
                num <- num + log(dnorm(newparam[1],mu0,sqrt(1/tau0)))
                num <- num + log(dgamma(newparam[2],alpha0,beta0))
#   Proposal terms cancel (symmetric distribution)
```

```
#   To calculate the log(denominator) of the acceptance probability
#   (ignoring constants)
#   Likelihood term (in simplified form):

                sn <- sum((x-param[1])^2)

                den <- n/2*log(param[2]) - param[2]*sn/2

#   Prior terms:

                den <- den + log(dnorm(param[1],mu0,sqrt(1/tau0)))

                den <- den + log(dgamma(param[2],alpha0,beta0))

#   Proposal terms cancel (symmetric distribution)

#   Calculate the acceptance probability:

                accept <- min(1,exp(num-den))

            }

            else {
                accept <- 0
            }

            v <- runif(1)

            if (v <= accept) {

# accept proposed parameter value
                param[i] <- newparam[i]
            }

            param1[t,i] <- param[i]
        }

    }

    par(mfrow=c(1,2))

    plot(1:length(param1[,1]),param1[,1],type="l",xlab="t",ylab="mu")
    plot(1:length(param1[,2]),param1[,2],type="l",xlab="t",ylab="tau")

param1

}
```

Note that the MCMC algorithm can be made more computationally efficient by considering the updating of the parameters separately and simplifying the acceptance probabilities for each case. However, in practice, for complex likelihood expressions, it is often easier to simply write a subroutine with the given likelihood and use this in the updating of each parameter.

19.5 Example: Rats

We return to the rats example introduced in Section 18.3. An R code, "R-ratsMH.R" is downloadable which implements an MCMC algorithm for this example, using the Metropolis-Hastings algorithm.

Main R program

The base program follows the standard format for the Metropolis-Hastings algorithm:

```
# Name of function and input parameter nt = number of iterations

ratsMH <- function(nt) {

# Read in the response data

    data <- matrix(c(
    151, 199, 246, 283, 320,
    145, 199, 249, 293, 354,
    147, 214, 263, 312, 328,
    .........................
    153, 200, 244, 286, 324),nrow=30,byrow=T)

# Read in the explanatory variables and mean value

    x <- c(8,15,22,29,36)
    xbar <- 22

# Centre the explanatory variables (x's):

    xcent <- x-xbar

# Read in number of individuals and number of times

    N = 30
    T = 5

# Read in the priors:

# For alpha:

    alphaprior <- c(0,10^6)

# For beta:

    betaprior <- c(0,10^6)

# For sigma^2

    varprior <- c(0.001,0.001)

# Parameters for MH updates (Uniform random walk) for alpha, beta and sigma^2:
```

```
    delta <- c(10,0.1,10)

# Set initial parameter values:

    alpha <- 250
    beta <- 6
    sigma2 <- 250

# Put these into vector of parameters

    param <- c(alpha,beta,sigma2)

# Define vector for storing simulated values
# (rows = iterations; columns = parameter)

    output <- array(0,dim=c(nt,3))

# Output the time to the screen at beginning of simulations

    cat("Start time", format(Sys.time(),"%X"), "\n")

# Iterations:

    for (t in 1:nt) {

# Cycle through each parameter in turn:

        for (i in 1:3) {

# Call the updating function to perform MH step

newvalue <- MHstep(param,N,T,data,xcent,alphaprior,betaprior,varprior,delta)

# Update parameter value

            param[i] <- newvalue[i]
        }

# Store parameter values in "output"

        output[t, ] <- newvalue[ ]
    }

# Output the time to the screen at end of simulations

    cat("Finish time", format(Sys.time(),"%X"), "\n")

# Output the set of parameter values at each iteration stored in "output"

output
}
```

Thus, all we need to do now is to write the subroutine that updates the parameters (`MHstep`).

MH updating step

One possible MH updating algorithm is as follows:

```
MHstep <- function(param,N,T,data,xcent,alphaprior,betaprior,varprior,delta){

# MH step (using single-update random walk) for each parameter

    for (j in 1:length(param)) {
        newparam <- param

# Propose new parameter value - using Uniform proposal distribution

        u <- runif(1,-delta[j],delta[j])
        newparam[j] <- param[j] + u

# Note that since sigma^2 > 0 we reject is the proposed value for sigma^2 < 0
# (i.e. accept with probability 0)

        if (newparam[3] > 0) {

# Evaluate the log-likelihood at current and proposed parameter values

            loglik <- likhoodfn(param,N,T,data,xcent)
            newloglik <- likhoodfn(newparam,N,T,data,xcent)

# Evaluate the log-prior at current and proposed parameter values

            logprior <- priorfn(param,alphaprior,betaprior,varprior)
            newlogprior <- priorfn(newparam,alphaprior,betaprior,varprior)

# Evaluate the log proposal density for proposed and current parameters

logprop1 <- log(dunif(newparam[j],param[j]-delta[j],param[j]+delta[j]))
logprop2 <- log(dunif(param[j],newparam[j]-delta[j],newparam[j]+delta[j]))

# Calculate the acceptance probability:

            num <- newloglik + newlogprior + logprop2
            den <- loglik + logprior + logprop1
            acc <- exp(num-den)

# Accept/reject step. Simulate random deviate from U[0,1] distribution

            v <- runif(1,0,1)

# If v <= acc then we accept the proposed parameter value

            if (v <= acc) {     # Update the parameter value

                param <- newparam
            }
        }
```

```
    }

# Output the set of updated parameter values:

param
}
```

Likelihood function

The final subroutine we need is simply the likelihood function (compare with the specification within the WinBUGS code given in Section 18.3).

```
likhoodfn <- function(param,N,T,data,xcent) {

# Function to calculate the log-likelihood evaluated at given parameter values

    likhood <- 0

# Cycle through each individual and time and evaluate the log
# of the pdf of the given Normal distribution
# (alpha = param[1]; beta = param[2]; sigma^2 = param[3])

    for (j in 1:N) {
        for (k in 1:T) {
            mean <- param[1] + param[2]*xcent[k]
            likhood <- likhood + log(dnorm(data[j,k],mean,sqrt(param[3])))
        }
    }

# Output the likelihood evaluated at given parameter values

likhood
}
```

Note that again this MH updating algorithm is not the most efficient. For example, we calculate the full posterior distribution when updating the parameter α, even though the corresponding prior distributions for β and σ^2 cancel in the acceptance probability. In addition, we could improve the efficiency by keeping track of the log-likelihood value so that we do not recalculate this unnecessary.

19.6 Final Comments

19–1. When using R remember to change the directory (`File` → `Change dir...`) to the one that you wish to work in, i.e. that contains your saved R codes etc. (The default is the directory containing the program R).

19–2. When writing R functions it is typically easier to write the file in a different package (e.g. Notepad), and save them as a ".R" file (so that it is easily identifiable as an R file).

19–3. In order to read in a function to R, type `source("filename")`. If you edit the function in an external package (such as Notepad), you need to read the file back into R in order for the changes to take effect.