Debiasing random walk Metropolis-Hastings

PEJ, JOL, YFA

August 2017

Setting

This script accompanies the article "Unbiased Markov chain Monte Carlo with couplings", by Pierre E. Jacob, John O'Leary and Yves F Atchade.

This script illustrates how the bias from a Metropolis-Hastings algorithm can be removed using coupled chains, on a simple example.

We begin by loading the package, registering multiple cores, setting the random number generator, etc.

```r
# load package
library(debiasedmcmc)
# register parallel cores
registerDoParallel(cores = 4)
# remove all
rm(list = ls())
# set RNG seed
set.seed(11)
```

Target distribution and MCMC

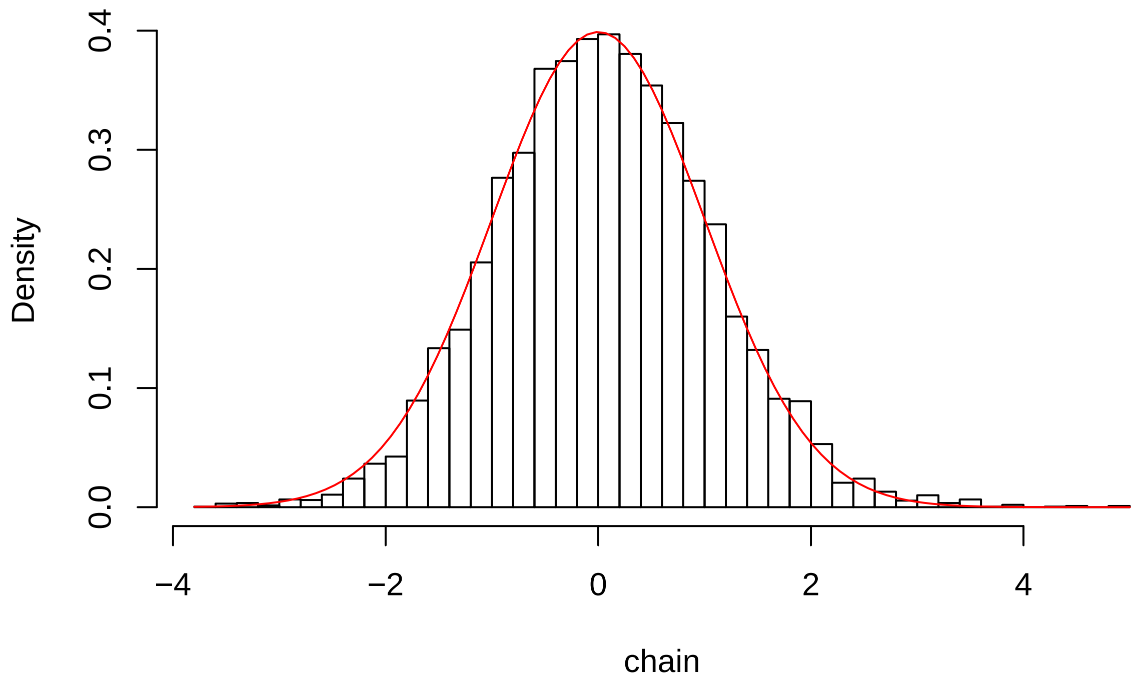
The target distribution π is defined as $\mathcal{N}(0,1)$. For our baseline MCMC kernel, we consider an MH algorithm with a proposal standard deviation of 1. The initial distribution is chosen as $\mathcal{N}(1,1)$. The following code defines the target and the MH kernel.

```r
logtarget <- function(x) dnorm(x, mean = 0, sd = 1, log = TRUE)
sd_proposal <- 1
rinit <- function() rnorm(1, 1, 1)
MH_kernel <- function(chain_state) {
    proposal_value <- rnorm(1, chain_state, sd_proposal)
    proposal_pdf <- logtarget(proposal_value)
    current_pdf <- logtarget(chain_state)
    if (log(runif(1)) < (proposal_pdf - current_pdf)) {
        return(proposal_value)
    } else {
        return(chain_state)
    }
}
```

We can check that the algorithm is correctly implemented by sampling many iterations, and plotting the histogram of the chain overlaid with the target probability density function.

```r
niterations <- 10000
chain <- rep(0, niterations)
chain_state <- rinit()
for (i in 1:niterations) {
    chain_state <- MH_kernel(chain_state)
    chain[i] <- chain_state
}
```

```r
hist(chain, prob = TRUE, nclass = 40, main = "")
curve(exp(logtarget(x)), add = TRUE, col = "red")
```



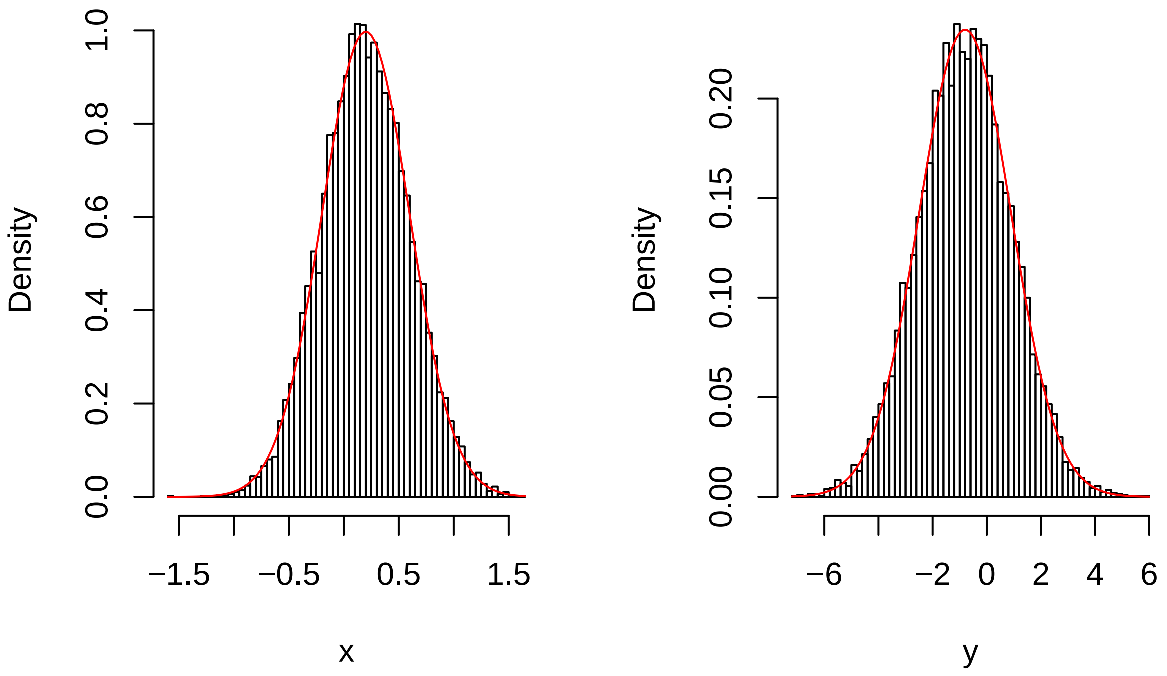
Coupled kernel

We now introduce a coupled MH kernel, by maximally coupling the Normal proposals. The code to sample from a maximal coupling of two Normals is provided below.

```r
rnorm_max_coupling <- function(mu1, mu2, sigma1, sigma2) {
    x <- rnorm(1, mu1, sigma1)
    if (dnorm(x, mu1, sigma1, log = TRUE) + log(runif(1)) < dnorm(x, mu2, sigma2,
        log = TRUE)) {
        return(c(x, x))
    } else {
        reject <- TRUE
        y <- NA
        while (reject) {
            y <- rnorm(1, mu2, sigma2)
            reject <- (dnorm(y, mu2, sigma2, log = TRUE) + log(runif(1)) < dnorm(y,
                mu1, sigma1, log = TRUE))
        }
        return(c(x, y))
    }
}
```

We can test that the above code works, as follows.

```r
mu1 <- 0.2
mu2 <- -0.8
sigma1 <- 0.4
sigma2 <- 1.7
xy <- foreach(i = 1:10000) %dorng% {
    rnorm_max_coupling(mu1, mu2, sigma1, sigma2)
}
par(mfrow = c(1, 2))
hist(sapply(xy, function(x) x[1]), prob = TRUE, nclass = 50, main = "", xlab = "x")
curve(dnorm(x, mu1, sigma1), add = TRUE, col = "red")
hist(sapply(xy, function(x) x[2]), prob = TRUE, nclass = 50, main = "", xlab = "y")
curve(dnorm(x, mu2, sigma2), add = TRUE, col = "red")
```



```
print(mean(sapply(xy, function(x) x[1]) == sapply(xy, function(x) x[2])))
```

```
## [1] 0.3476
```

Here the probability of the event $\{X = Y\}$ is about 35% for the given means and standard deviations.

We define a coupled MH kernel using maximally coupled proposals as follows.

```
coupledMH_kernel <- function(chain_state1, chain_state2) {
    proposal_value <- rnorm_max_coupling(chain_state1, chain_state2, sd_proposal,
        sd_proposal)
    proposal_pdf1 <- logtarget(proposal_value[1])
    proposal_pdf2 <- logtarget(proposal_value[2])
    current_pdf1 <- logtarget(chain_state1)
    current_pdf2 <- logtarget(chain_state2)
    logu <- log(runif(1))
    if (is.finite(proposal_pdf1)) {
        if (logu < (proposal_pdf1 - current_pdf1)) {
            chain_state1 <- proposal_value[1]
        }
    }
    if (is.finite(proposal_pdf2)) {
        if (logu < (proposal_pdf2 - current_pdf2)) {
            chain_state2 <- proposal_value[2]
        }
    }
    return(list(chain_state1 = chain_state1, chain_state2 = chain_state2))
}
```

Coupled chains and meeting times

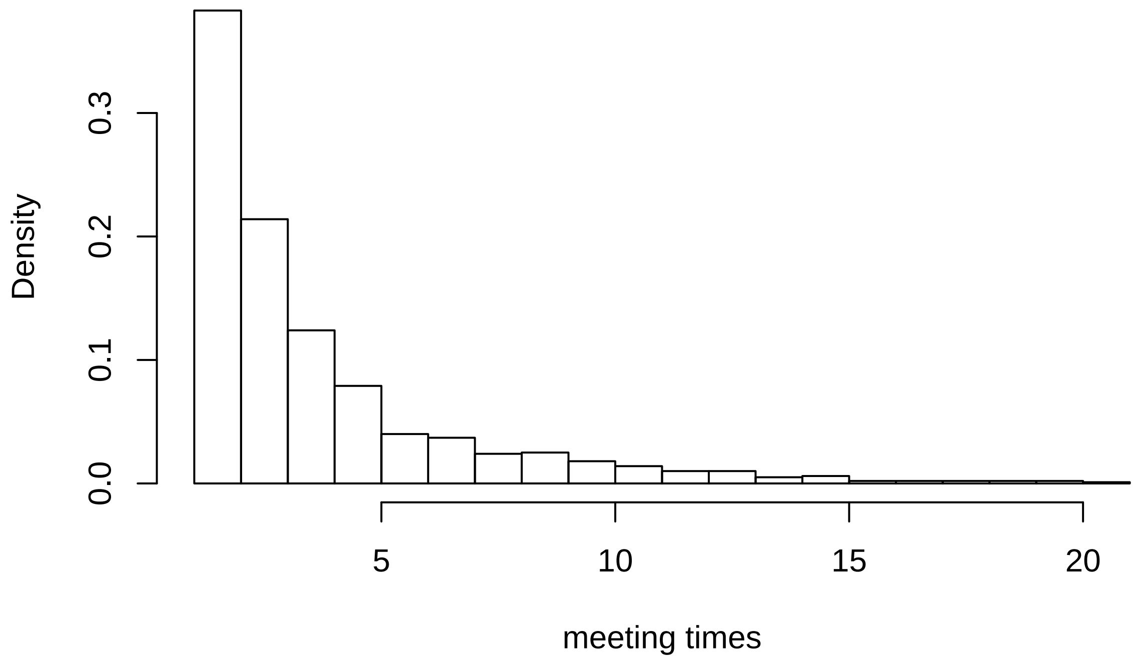
We can then run coupled chains. The following code uses the function coupled_chains to run chains until they meet. It returns the two chains, in 'samples1' and 'samples2', the meeting time, and the total number of iterations performed, which here is equal to the meeting time.

```
coupled_chains(MH_kernel, coupledMH_kernel, rinit)
```

```
## $samples1
##             [,1]
## [1,] -1.07453575
## [2,]  0.46375243
## [3,] -0.04411925
## [4,] -0.28161197
##
## $samples2
##           [,1]
## [1,]  1.098958
## [2,]  1.098958
## [3,] -0.281612
##
## $meetingtime
## [1] 3
##
## $iteration
## [1] 3
##
## $finished
## [1] TRUE
```

We then run 1000 independent copies of the chains, and plot a histogram of the meeting times.

```
nsamples <- 1000
c_chains_ <- foreach(irep = 1:nsamples) %dorng% {
    coupled_chains(MH_kernel, coupledMH_kernel, rinit)
}
meetingtime <- sapply(c_chains_, function(x) x$meetingtime)
hist(meetingtime, breaks = 1:max(meetingtime), prob = TRUE, main = "", xlab = "meeting times")
```

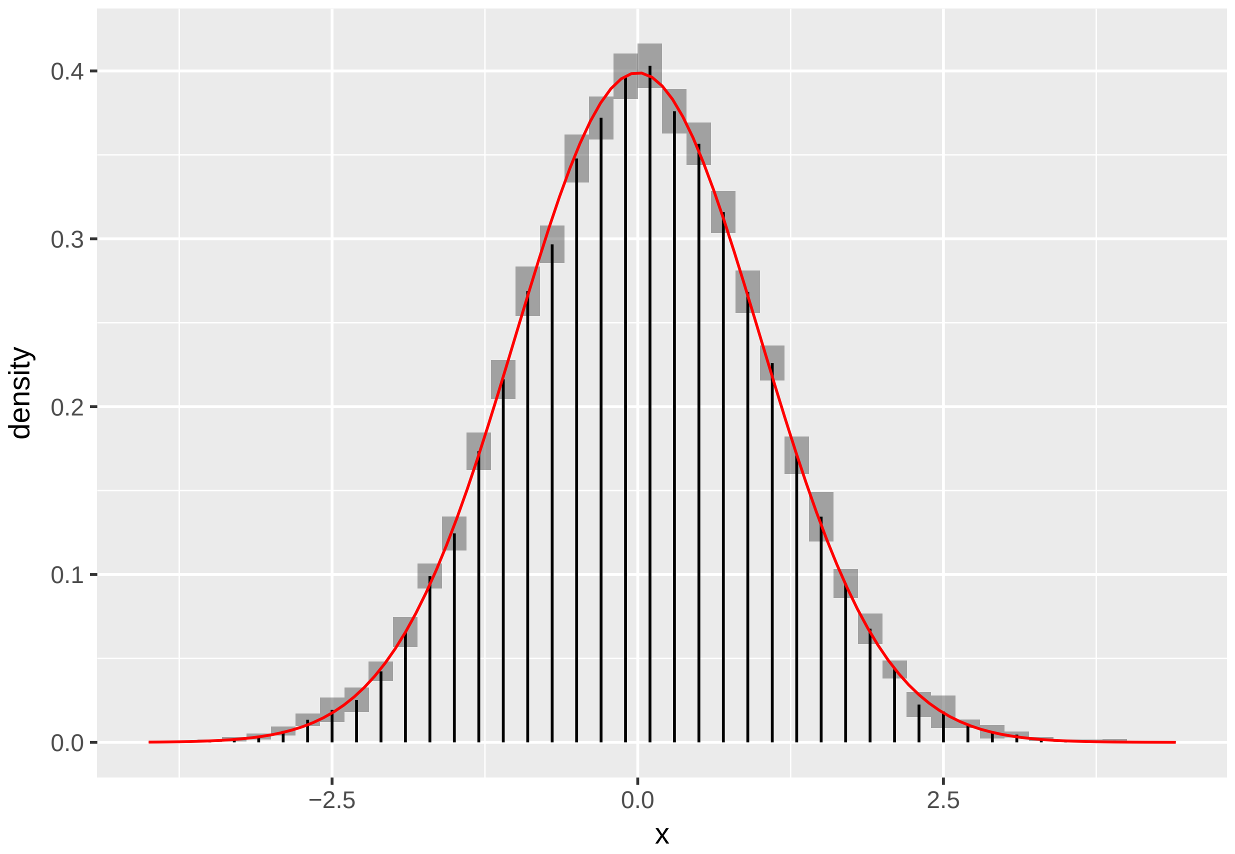


We see that the meetings tend to occur pretty quickly. Following the heuristics given in the article, we define k as the 95% quantile of the meeting times, and m as $10k$.

Histogram of the target

We now use coupled chains to produce an approximation of the target with the following code.

```r
k <- as.numeric(quantile(meetingtime, 0.95))
m <- 10 * k
nsamples <- 1000
c_chains_m <- foreach(irep = 1:nsamples) %dorng% {
    coupled_chains(MH_kernel, coupledMH_kernel, rinit, K = m)
}
histogram <- histogram_c_chains(c_chains_m, 1, k, m, nclass = 50)
g <- plot_histogram(histogram, with_bar = T)
g <- g + stat_function(fun = function(x) exp(logtarget(x)), colour = "red",
    alpha = 1)
g
```



```
cat("Histogram produced with k =", k, "and m =", m, ".\n")
```

```
## Histogram produced with k = 11 and m = 110 .
```

So in this case, each estimator is produced by running the coupled chains for 110 steps, and averaging after 11 steps.

The 'histogram_c_chains' takes a list of coupled chains, a component index (here equal to 1), the values of k and m, and a desired number of classes. Alternatively, one can specify the breaks of the histogram as a vector, with the argument 'breaks'.