evolMC demo

Grady Weyenberg

December 5, 2013

evolMC is a framework for doing Monte-Carlo simulations. The package, as of this writing, is not particularly optimized for speed, but to allow for the user to easily construct arbitrary MCMC schemes.

1.1 Univariate and multivariate sampling

We wish to use a metropolis sampler to draw from a distribution with density

$$f(x) \propto \frac{\sin(x)}{x} \cdot 1_{(0,\pi)}(x).$$

We can use a uniform distribution on $(-1, 1)$ to propose distances to jump from the current location.

Since the proposal distribution is symmetric, this is enough information to implement a Metropolis updater. We must implement a log-density function[1] that defines the distribution. We must also implement a proposal generation function.

```
ln.d <- function(x) log(sin(x)/x * (0 < x) * (x < pi))
propose <- function(x) x + runif(length(x), -1, 1)
```

The `metropolis` function accepts the functions that we have defined, and returns a new function. The returned function is an implementation of the Metropolis updating scheme you have defined.[2]

```
updater <- metropolis(ln.d, propose)
```

A Markov chain is formed by iteratively calling the updating function starting with some initial value. The `iterate` function forms a *chain object*, which is simply a list of states that the chain realized. If the state object is of a few particularly simple (but widely applicable) forms then there are convenient methods defined for several common R functions.

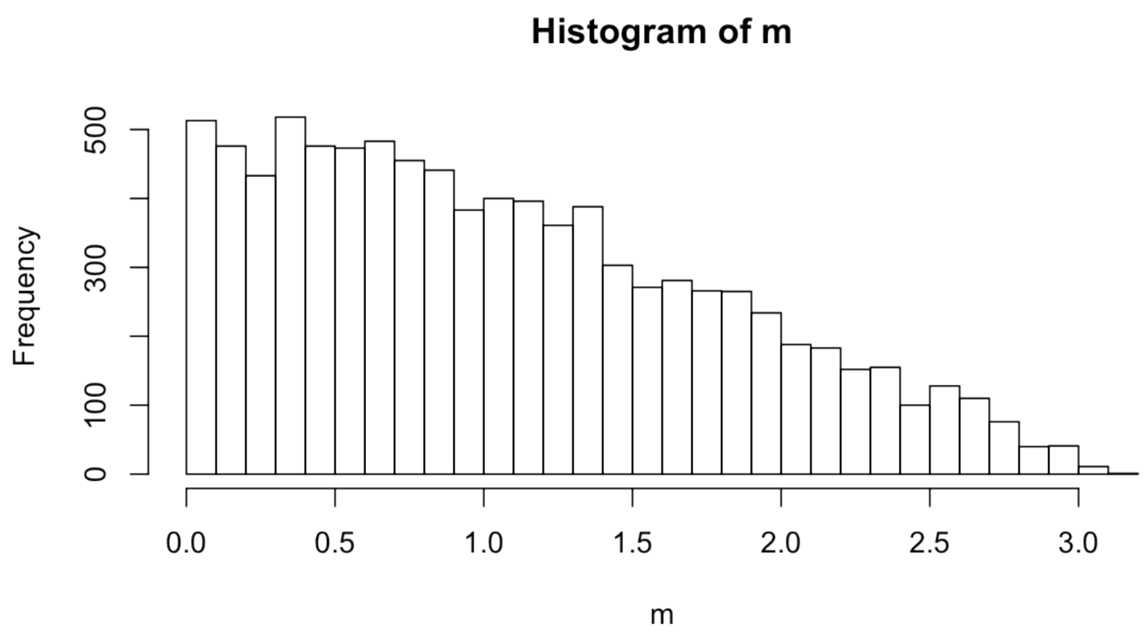
```
chain <- iterate(n = 10000, fn = updater, init = 1)
summary(chain, discard = 57)

## Discarding first 57 states.
##    mean      se    2.5%   97.5%
## 1.07001 0.73139 0.03456 2.63735
```

There is a method for the summary function that reports summary statistics for the chain. A method for the hist function will create a histogram from the chain object. By default, these methods discard the first 10% of values from the chain as a burn-in, but this may be altered if desired by the user.

[1] Actually, this function need only be known up to the normalizing constant.

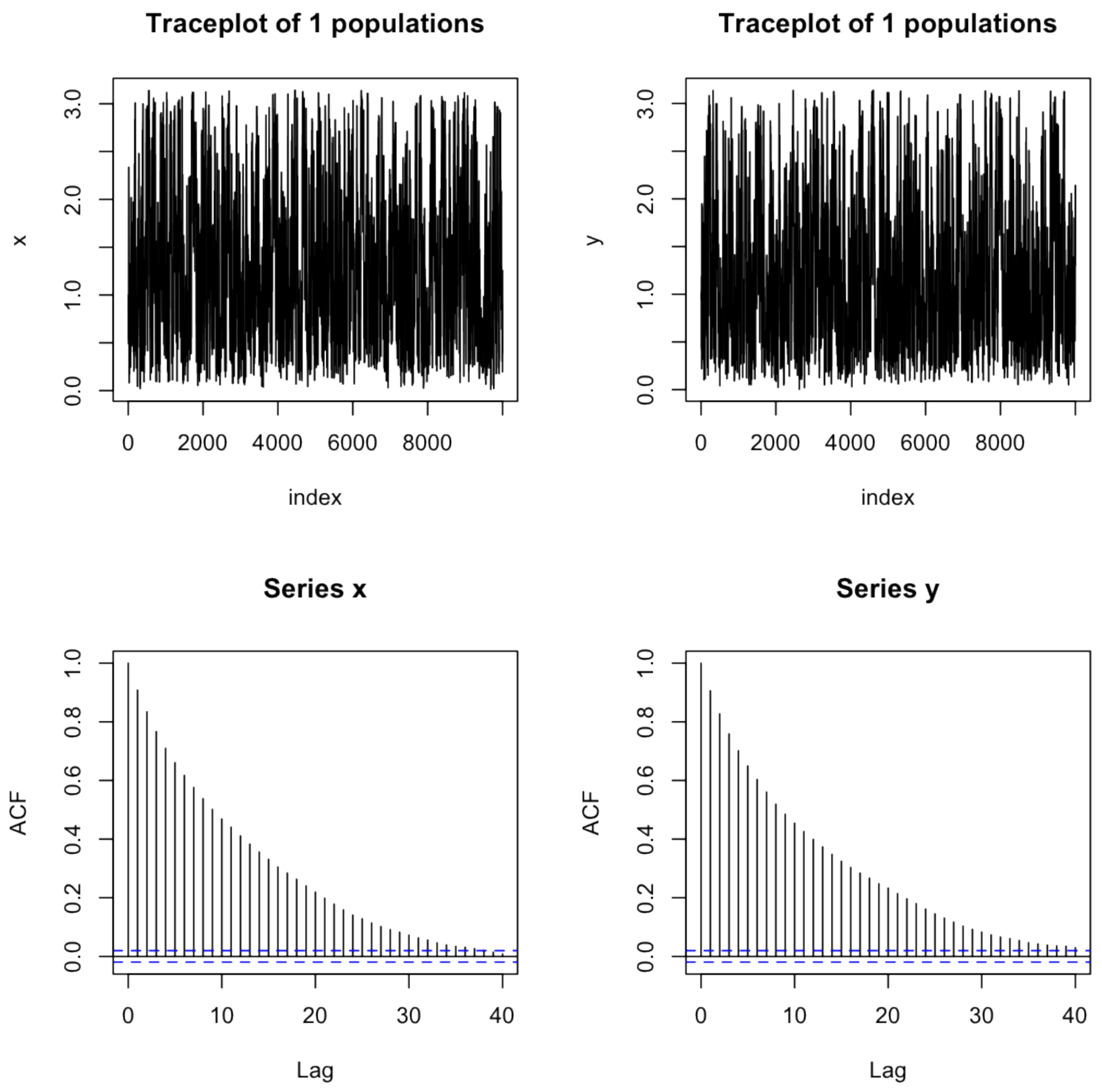
[2] If your proposal generator is not symmetric, then you must also supply a log-density for the proposal mechanism, and in this case the function returned will implement Metropolis-Hastings.

```
hist(chain, breaks = "fd")

## Discarding first 1000 states.
```

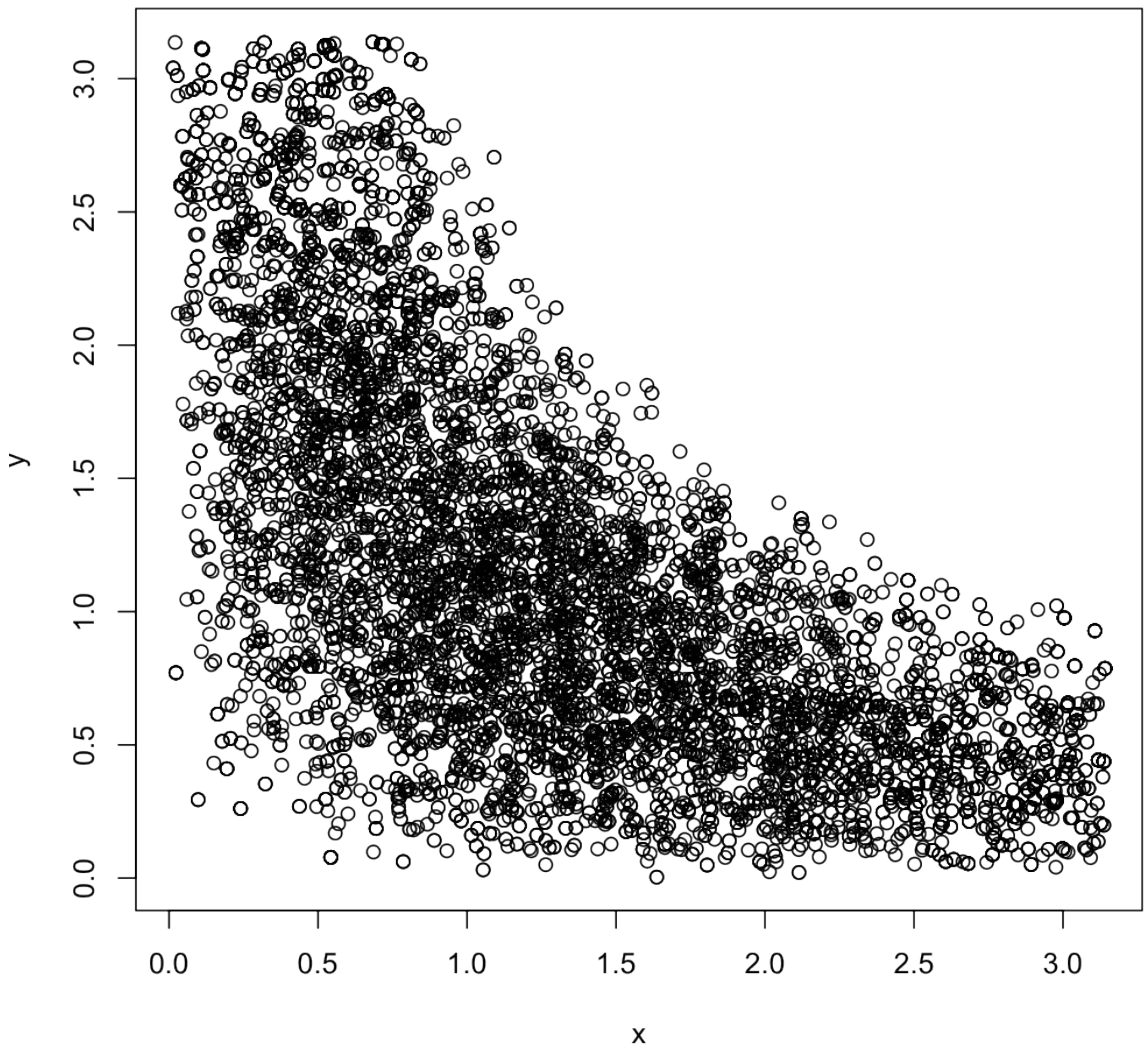


Of course, multivariate distributions may also be sampled. We need only define an appropriate multivariate log-density function. A quick review of the `propose` function we defined for the univariate sampler confirms that it will also work in the case when it is passed a vector. (Due to the `length(x)` rather than `1`.) We pass these functions to `metropolis` and obtain our updating function, which is used with `iterate` to form the Markov chain.

```
mvtarget <- function(x) log(prod(sin(prod(x))/x) * all(x > 0, x < pi, prod(x) <
    pi))
mvupdate <- metropolis(mvtarget, propose)
chain2 <- iterate(10000, mvupdate, cbind(x = 1, y = 1))
summary(chain2)

## Discarding first 1000 states.
##             x      y
## mean  1.3051 1.2055
## se    0.8184 0.7974
## 2.5%  0.1707 0.1665
## 97.5% 2.9813 2.9978

plot(chain2)
```



```r
plot(t(simplify2array(chain2)[1, , ]))
```

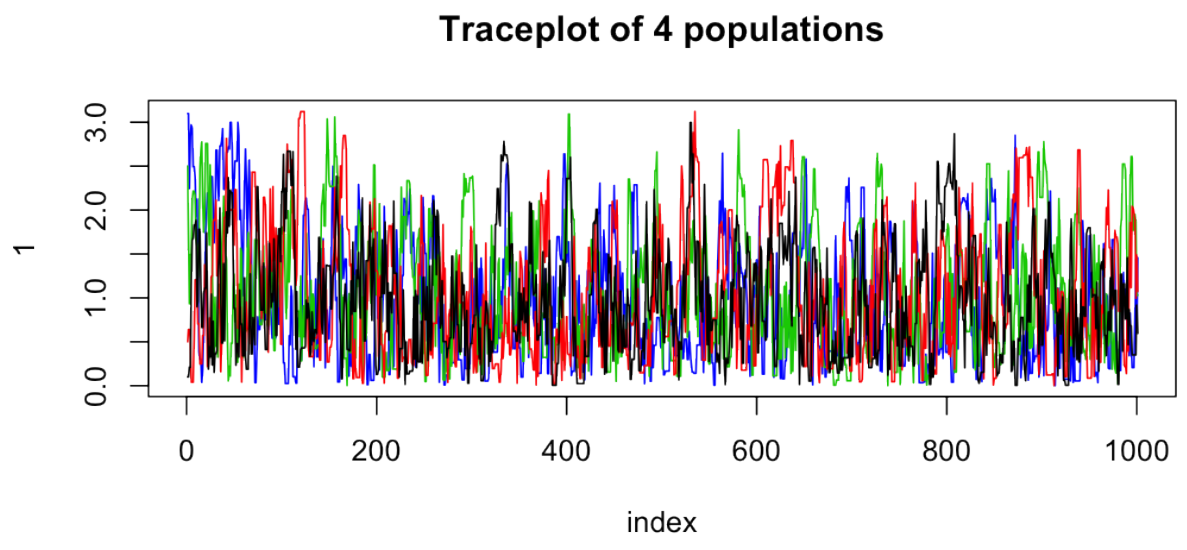


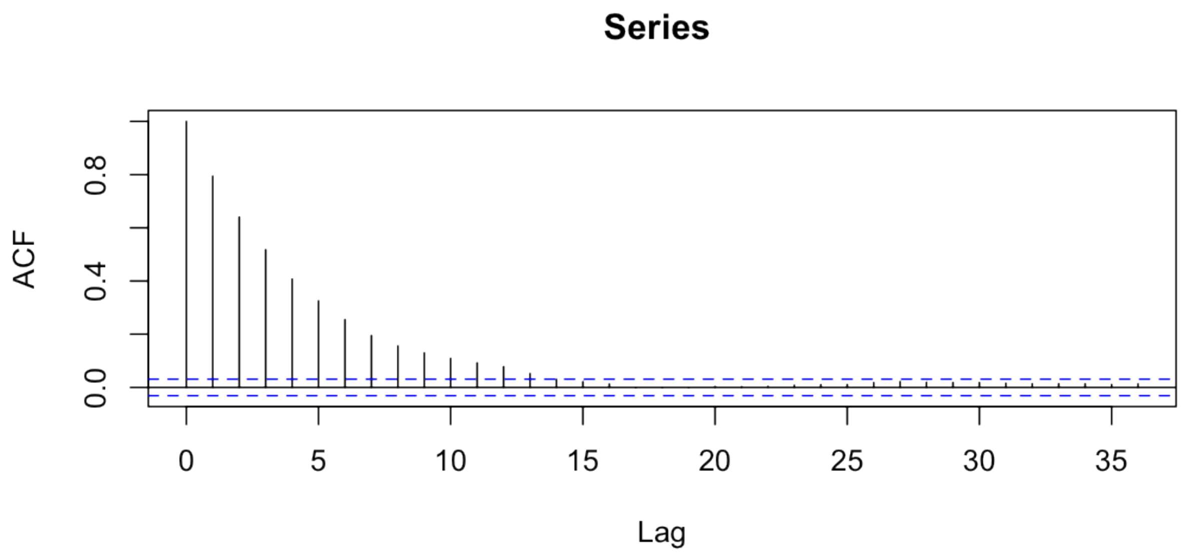
1.2 Multiple chains in parallel

If the target density returns a vector instead of a scalar, we can run multiple chains in parallel. Our proposal updater should update the entire population. If a state is a matrix (as created here by `rbind`), then plot (*et. al.*) assumes multiple parallel chains.[3]

```r
chain3 <- iterate(1000, updater, init = rbind(0.1, 0.5, 2.5, 3.1))
plot(chain3)
```

[3]The whole issue of what the "default" structure of a state is flexible at this point, and I haven't given much thought yet to what the optimal setup may be. On the todo list.

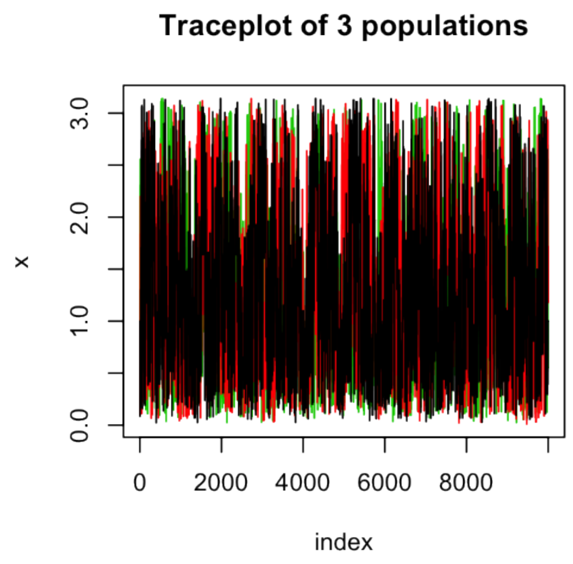


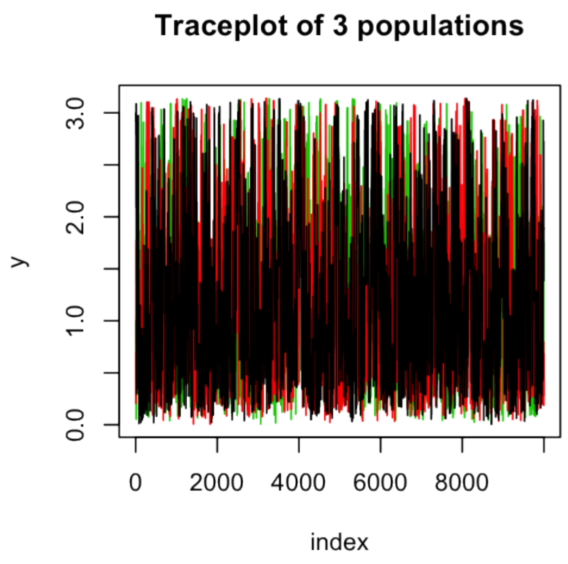


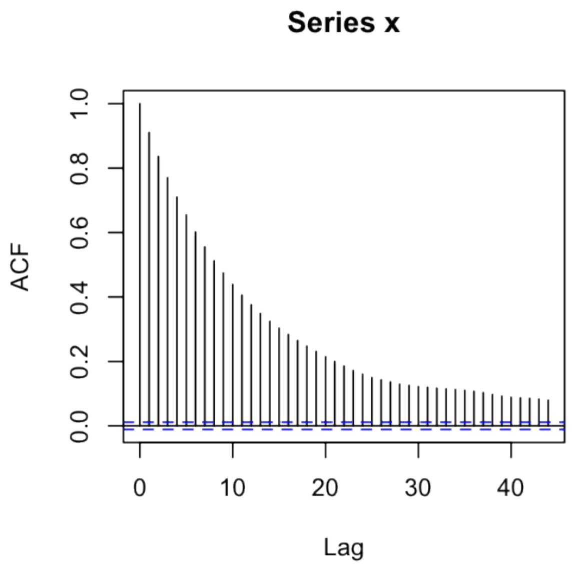
Multiple chains of multivariate samples are also possible, as long as a logical vector of the same length as that returned by the density function will "correctly" subset individuals from the population state object. [4]

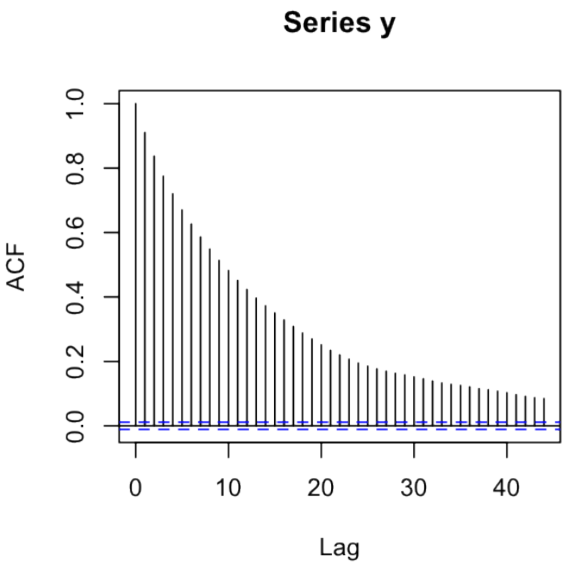
```
mvt2 <- function(x) apply(x, 1, mvtarget)
mvup2 <- metropolis(mvt2, propose)
init <- rbind(c(x = 1, y = 3), c(0.1, 0.5), c(1, 1))
chain4 <- iterate(10000, mvup2, init)
plot(chain4)
```

[4]Thus, a population can be a vector, a matrix with individuals in rows, or a list. Some convenience methods such as print, summary, plot, hist, etc., assume a population is a matrix, but this is not strictly required.









1.3 Gibbs sampling

A gibbs updater calls several updating functions sequentially. The following example problem is given in Casella and George (1992).[5] We wish to sample from a joint distribution

$$p(x, y) = \binom{n}{x} y^{x+\alpha-1}(1-y)^{n-x+\beta-1}.$$

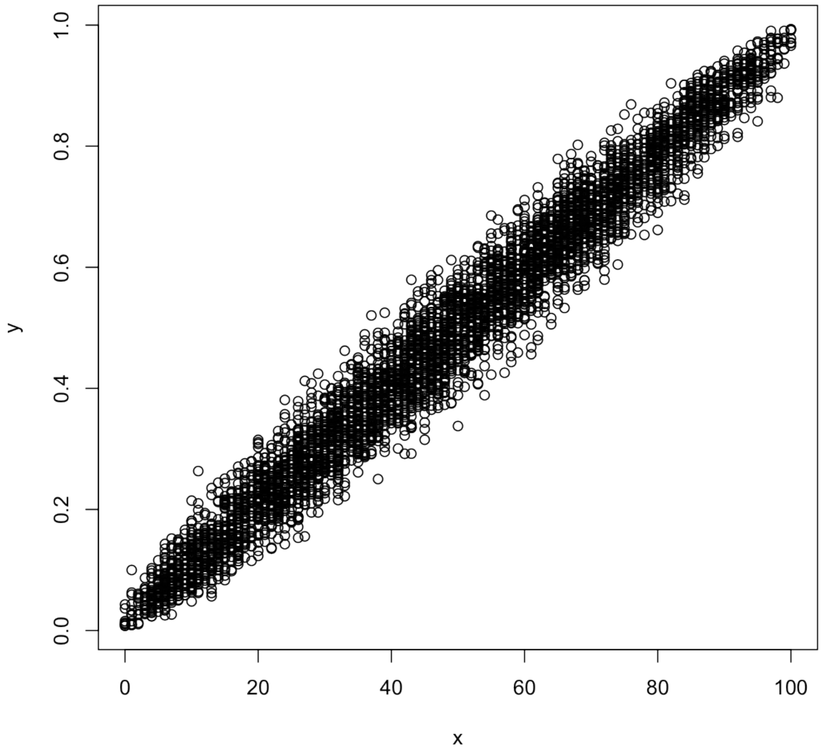
The conditional distributions are $x|y \sim \text{Bin}(n, y)$, and $y|x \sim \text{Beta}(x + \alpha, n - x + \beta)$. We let $n = 100$, and $\alpha = \beta = 2$ in this example.

[5] I actually found this at `http://web.mit.edu/~wingated/www/introductions/mcmc-gibbs-intro.pdf`.

```
## state = cbind(x,y)
x.y <- function(state) {
    state[, 1] <- rbinom(nrow(state), 100, state[, 2])
    state
}
y.x <- function(state) {
    state[, 2] <- rbeta(nrow(state), state[, 1] + 2, 100 - state[, 1] + 2)
    state
}
```

The gibbs function is used to create a function which acts as the Gibbs updater.

```
init <- cbind(x = 1:10, y = seq(0.1, 0.9, length.out = 10))
gibbs.updater <- gibbs(x.y, y.x)
chain5 <- iterate(5000, gibbs.updater, init)
plot(t(simplify2array(chain5)[1, , ]))
```



Of course, you can use a metropolis updater in place of a full conditional sampler as we did above. In the extreme case, the one- parameter-at-a-time metropolis updating scheme can be implemented using `gibbs`.

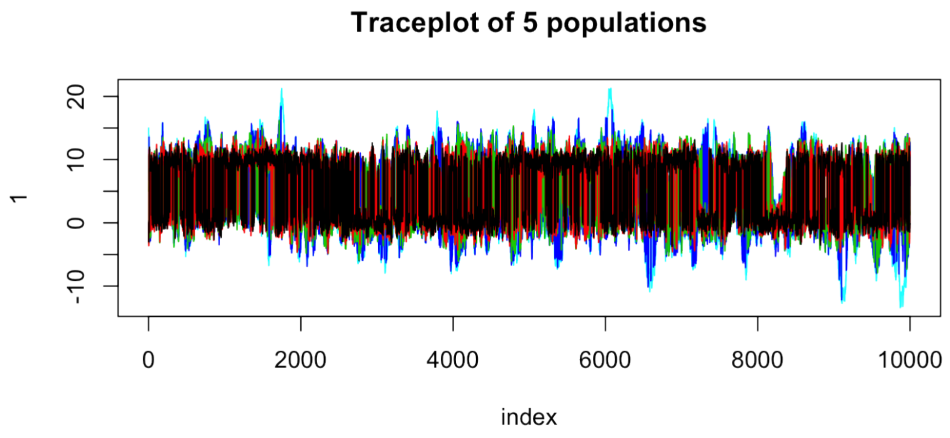
1.4 Parallel tempering

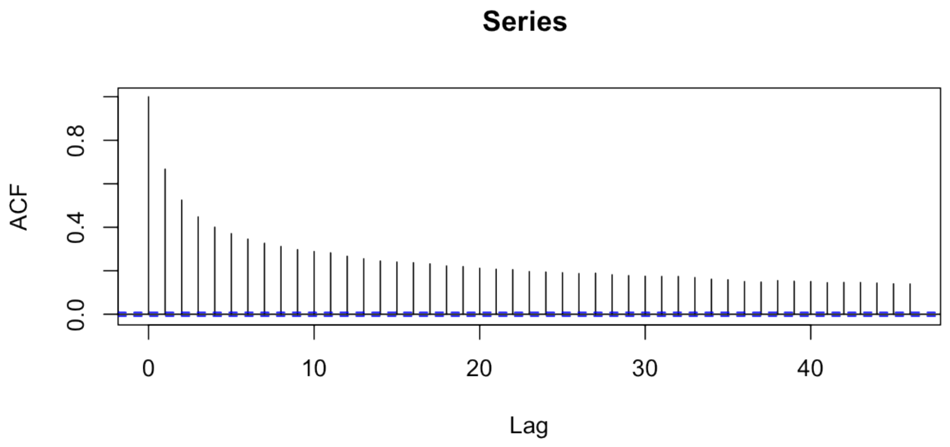
We want to sample from a distribution with separated modes. First we set up the target distribution function. This function should take a single individual and return the density of that individual.

```r
f <- function(state) log(dnorm(state) + dnorm(state, 10))
```

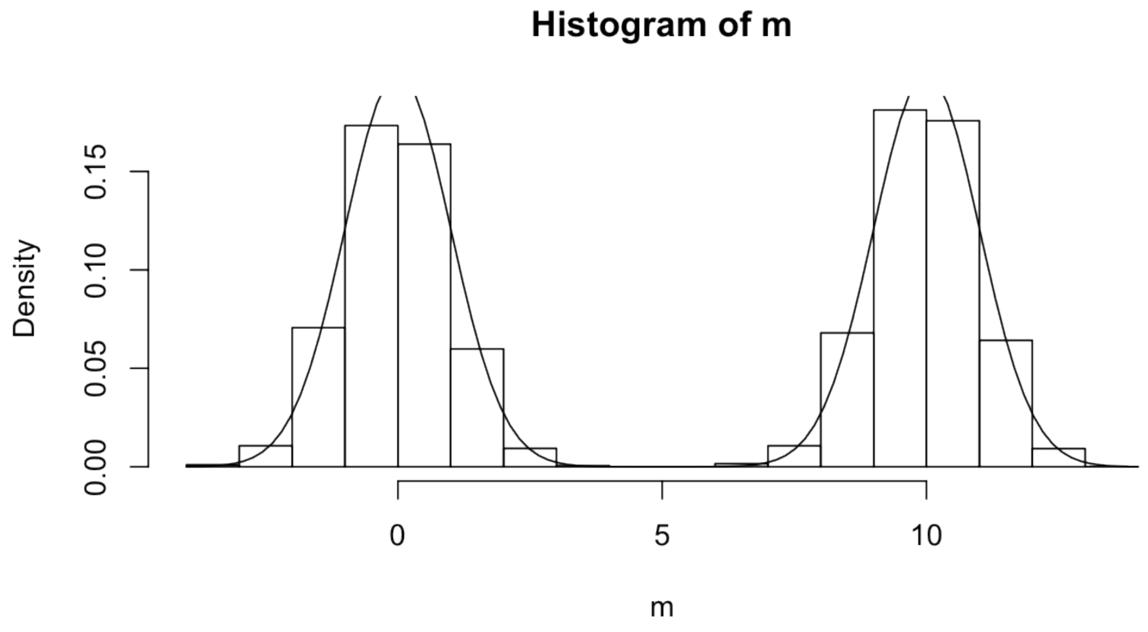
Now we can set up the parallel tempering updater. We define a population to have individuals in rows and multiple variates. First we use the `heat` function to create a function which will evaluate heated density values for each individual in the population. The `mutate` function does a metropolis update of each individual in the population using the supplied proposal function as described previously, then it attempts to exchange individuals between temperatures using an appropriate metropolis scheme.

```r
temps <- c(1, 2, 4, 8, 15)
hot.dens <- heat(f, temps)
updater <- mutate(hot.dens, propose)
init <- rbind(-3, 0, 5, 10, 15)
gch <- iterate(10000, updater, init)
plot(gch)
```





```r
xx <- prune(gch, 1, 1)  #extract chain with correct distribution
hist(xx, freq = FALSE, breaks = "fd")
curve((dnorm(x) + dnorm(x, 10))/2, add = TRUE)
```

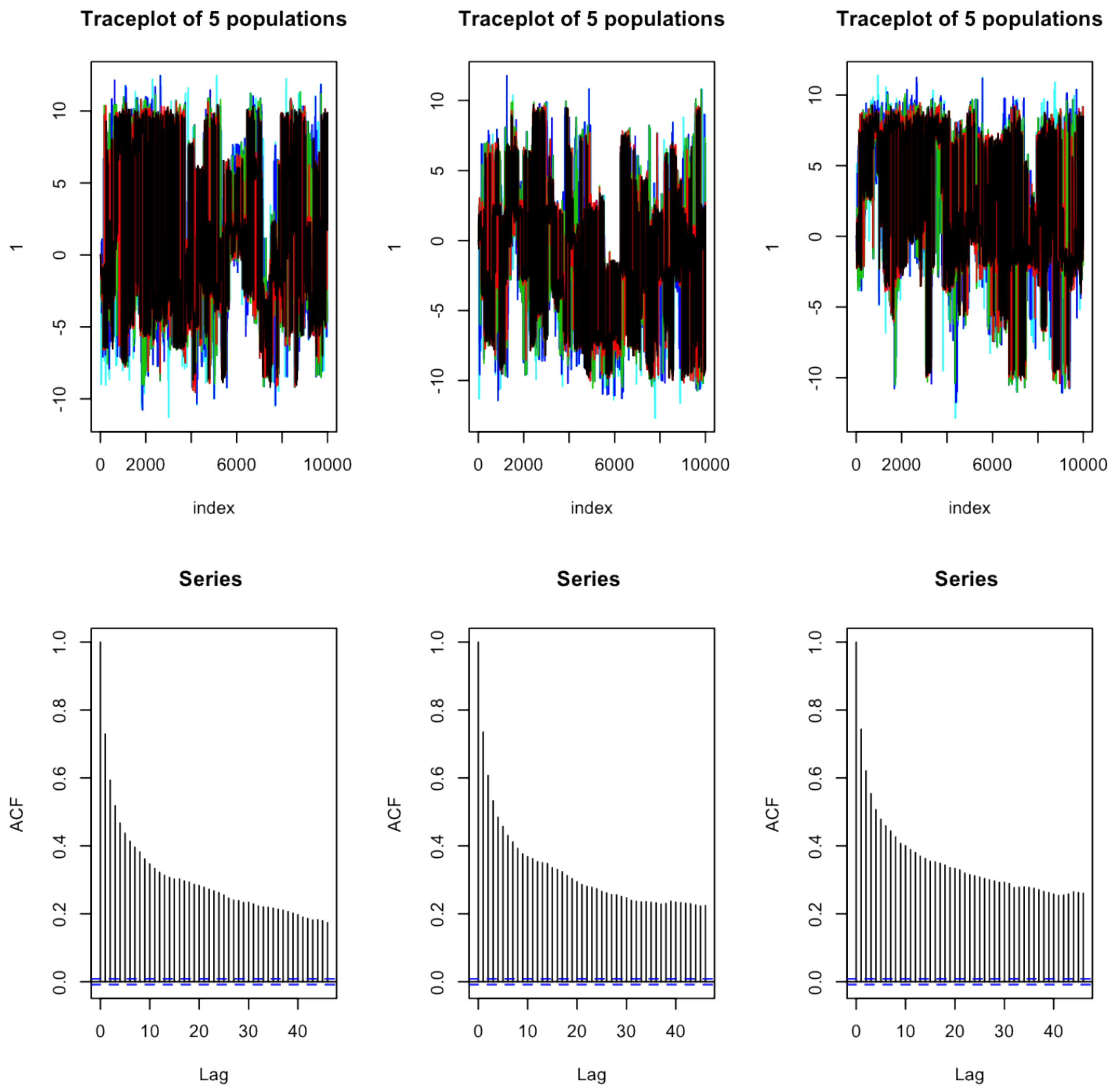


1.5 Evolutionary MC

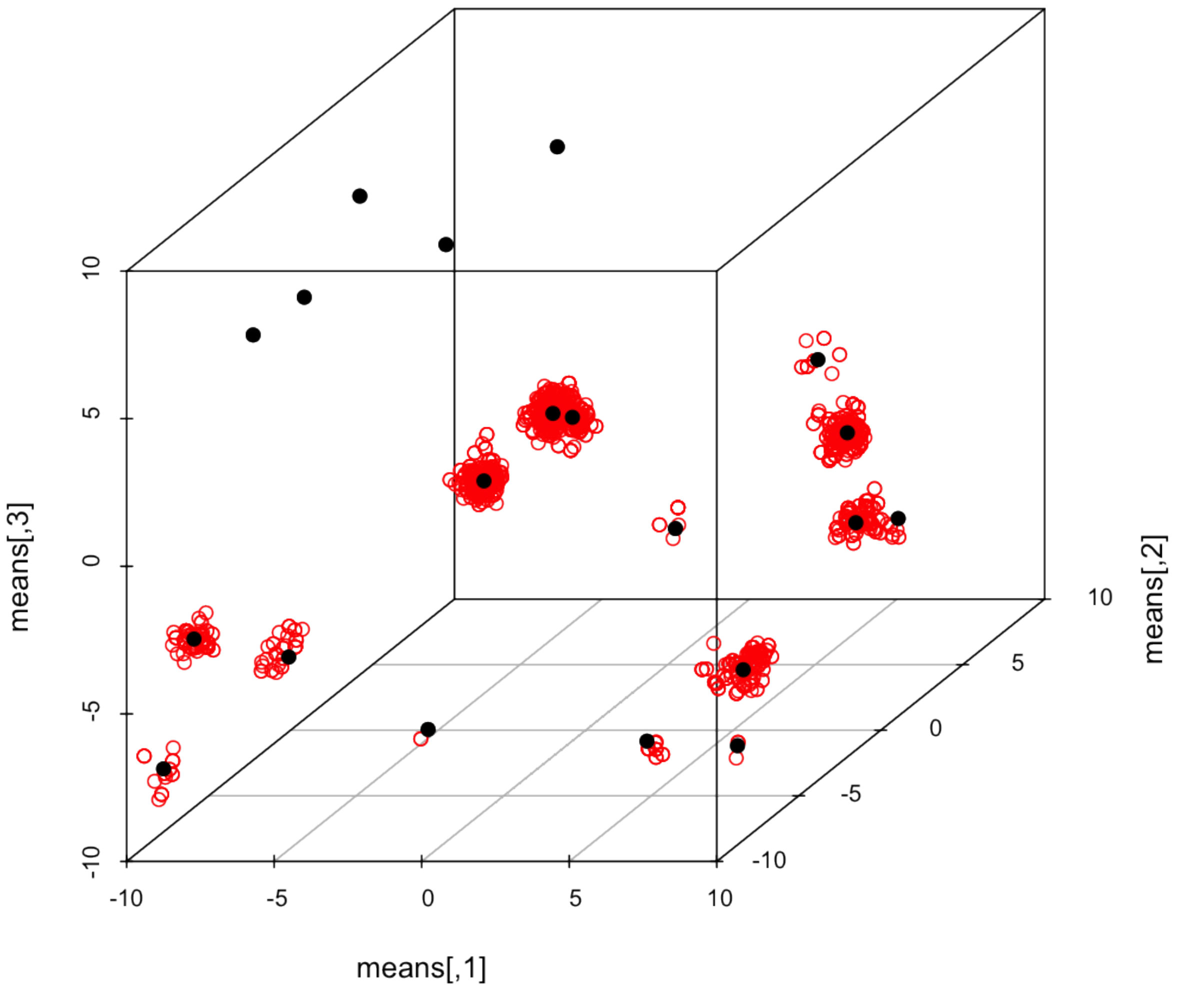
Here we go with the real-deal. Twenty part trivariate-gaussian mixure. The target function is a part of the evolMC package.

```
means <- t(replicate(20, runif(3, -10, 10)))
dmix <- function(x) target(x, means)
ladder <- heat(dmix, temps)
e <- seq(0.2, 3, length = 5)
runif3 <- function(x) x + runif(length(x), -e, e)
runif4 <- function(x) x + rnorm(length(x), 0, 0.25 * temps)
evolve <- gibbs(reproduce(ladder), mutate(ladder, runif4), p = c(1, 1))
ptemp <- mutate(ladder, runif4)
wt.evolve <- gibbs(weighted.reproduce(ladder), mutate(ladder, runif4), p = c(1,
    1))

init <- matrix(0, 5, 3)
# invisible(runif(100)) lineage <- iterate(2000, evolve, init, bar=1)
# pt.lineage <- iterate(2000, ptemp, init, bar=1)
wt.lineage <- iterate(2000, wt.evolve, init, bar = 1)

## ===========================================================================
```

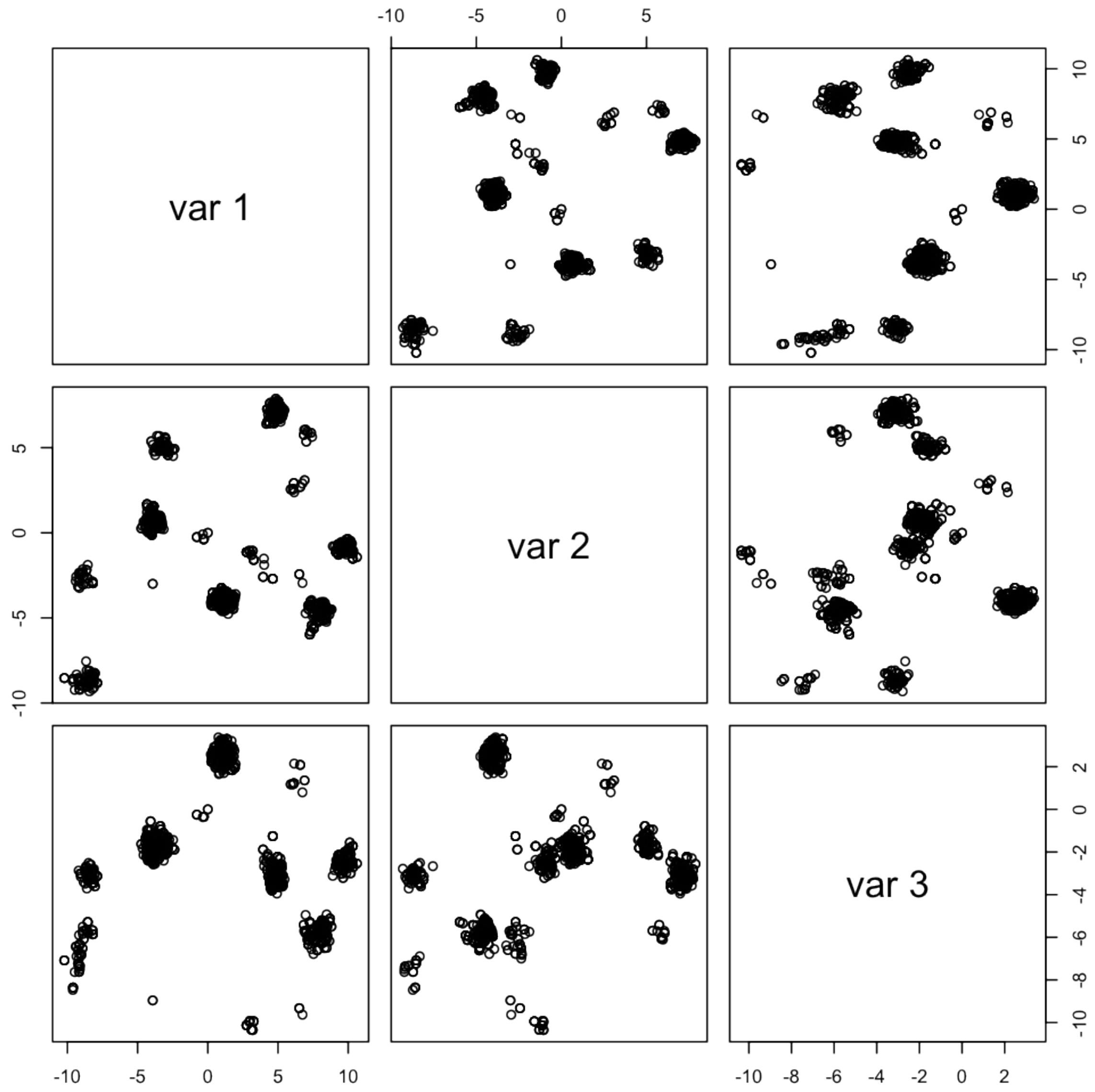
```
plot(lineage)
```



```
m <- aperm(simplify2array(wt.lineage))
## wt.m <- aperm(simplify2array(wt.lineage)) pt.m <-
## aperm(simplify2array(pt.lineage))
library(scatterplot3d)
pp <- scatterplot3d(means)
# invisible(sapply(5:1,function(i) pp£points3d(m[,,i],col=i+1)))
pp$points3d(m[, , 1], col = 2)
## pp£points3d(wt.m[,,1],col=3) pp£points3d(pt.m[,,1],col=3)
## pp£points3d(m[,,1],col=2)

pp$points3d(means, col = 1, pch = 19)
```



```
pairs(m[, , 1])
```



```
## library(rgl) plot3d(m[,,1]) plot3d(means,pch=19,size=15)
## sapply(1:5,function(i) points3d(m[,,i],col=i+1))
```

There is now an `as.mcmc.list` method that converts chain objects to `coda mcmc.list` objects. Here we compute the effective sample size of the chain with the desired stationary distribution.

```
effectiveSize(as.mcmc.list(lineage)[[1]])

##  var1  var2  var3
## 186.9 169.4 135.6
```