evolMC demo

Grady Weyenberg

November 22, 2013

evolMC is a framework for doing Monte-Carlo simulations.

0.1 Univariate and multivariate sampling

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

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

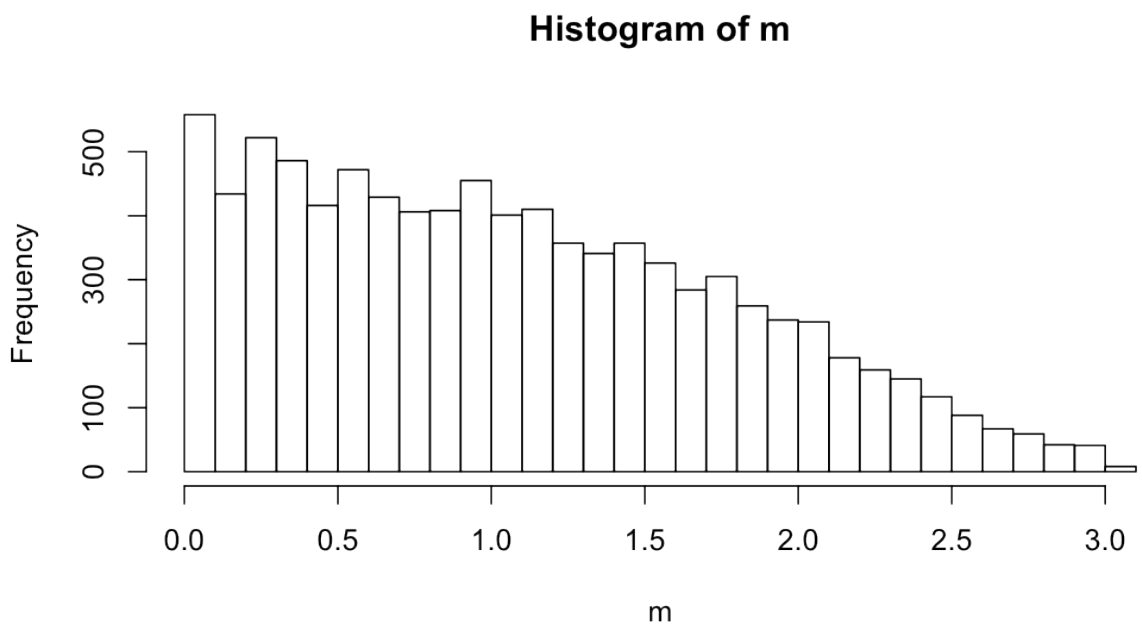
We can use a uniform distribution on $(-1, 1)$ to propose distances to jump from the current location.

Since the proposal distribution is symmetric, this is enough information to implement a Metropolis updater.

```
fn <- function(x) log(sin(x)/x * (0 < x) * (x < pi))
propose <- function(x) x + runif(length(x), -1, 1)
updater <- metropolis(fn, propose)
```

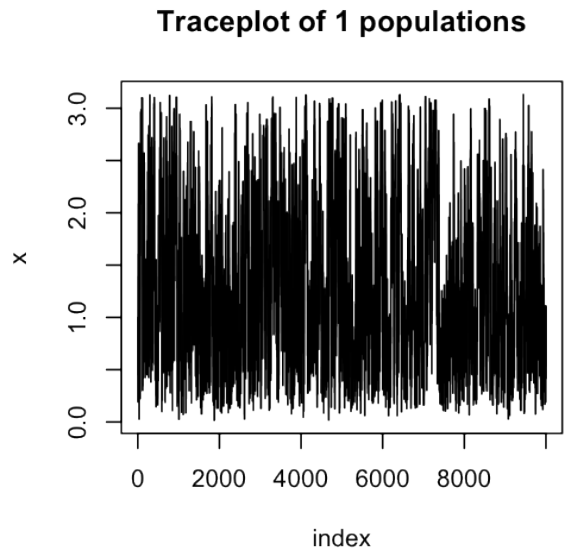
A Markov chain is formed by iteratively calling the updating function starting with some initial value.

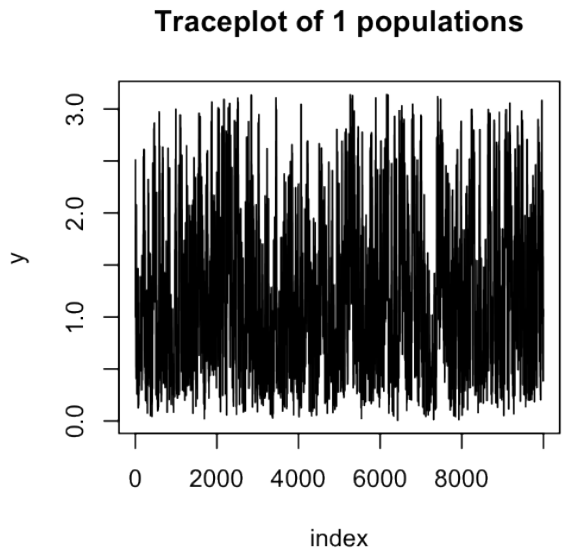
```
chain <- iterate(n = 10000, fn = updater, init = 1)
summary(chain)

## Discarding first 1000 states.
##    mean      se    2.5%   97.5%
## 1.07100 0.72155 0.03758 2.58650

hist(chain, breaks = "fd")

## Discarding first 1000 states.
```

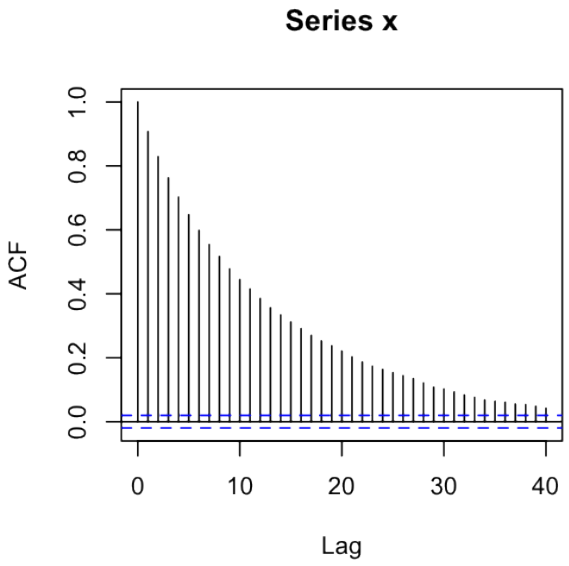


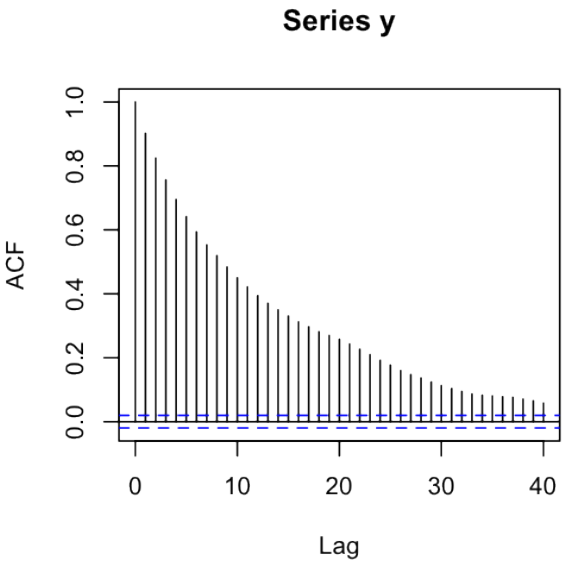
Of course, multivariate distributions may also be sampled.

```
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  1.2738 1.2330
## se    0.8105 0.7910
## 2.5%  0.1703 0.1816
## 97.5% 3.0076 2.9277

plot(chain2)
```

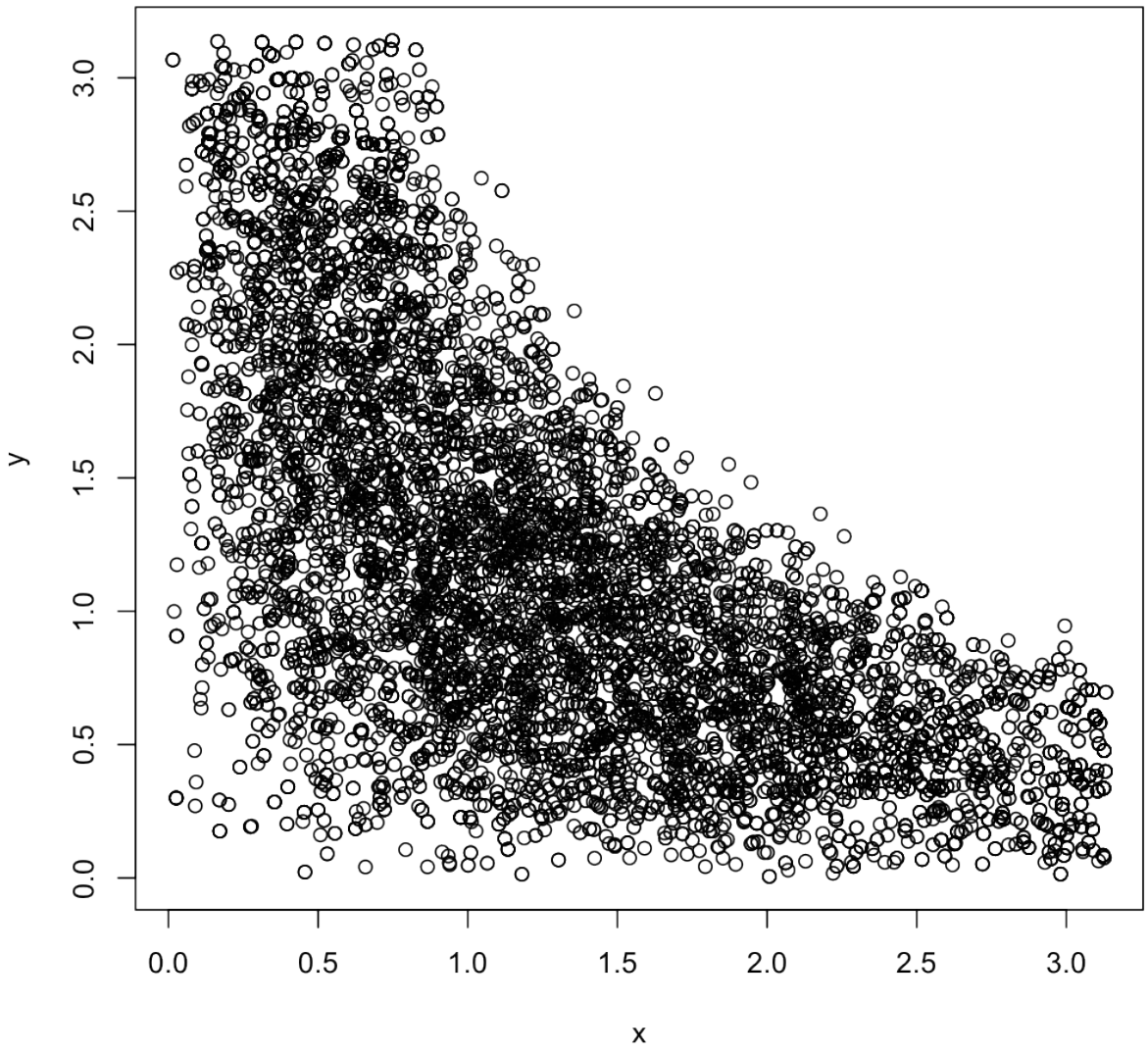









```
plot(t(simplify2array(chain2)[1, , ]))
```



0.2 Multiple chains in parallel

If the target density returns a vector instead of a scalar, we can run multiple chains in parallel. (In this case the state object is a matrix with individuals in rows.) In this situation, the proposal updater should update the entire population.

```
chain3 <- iterate(100, updater, init = rbind(0.1, 0.5, 2.5, 3.1))
```

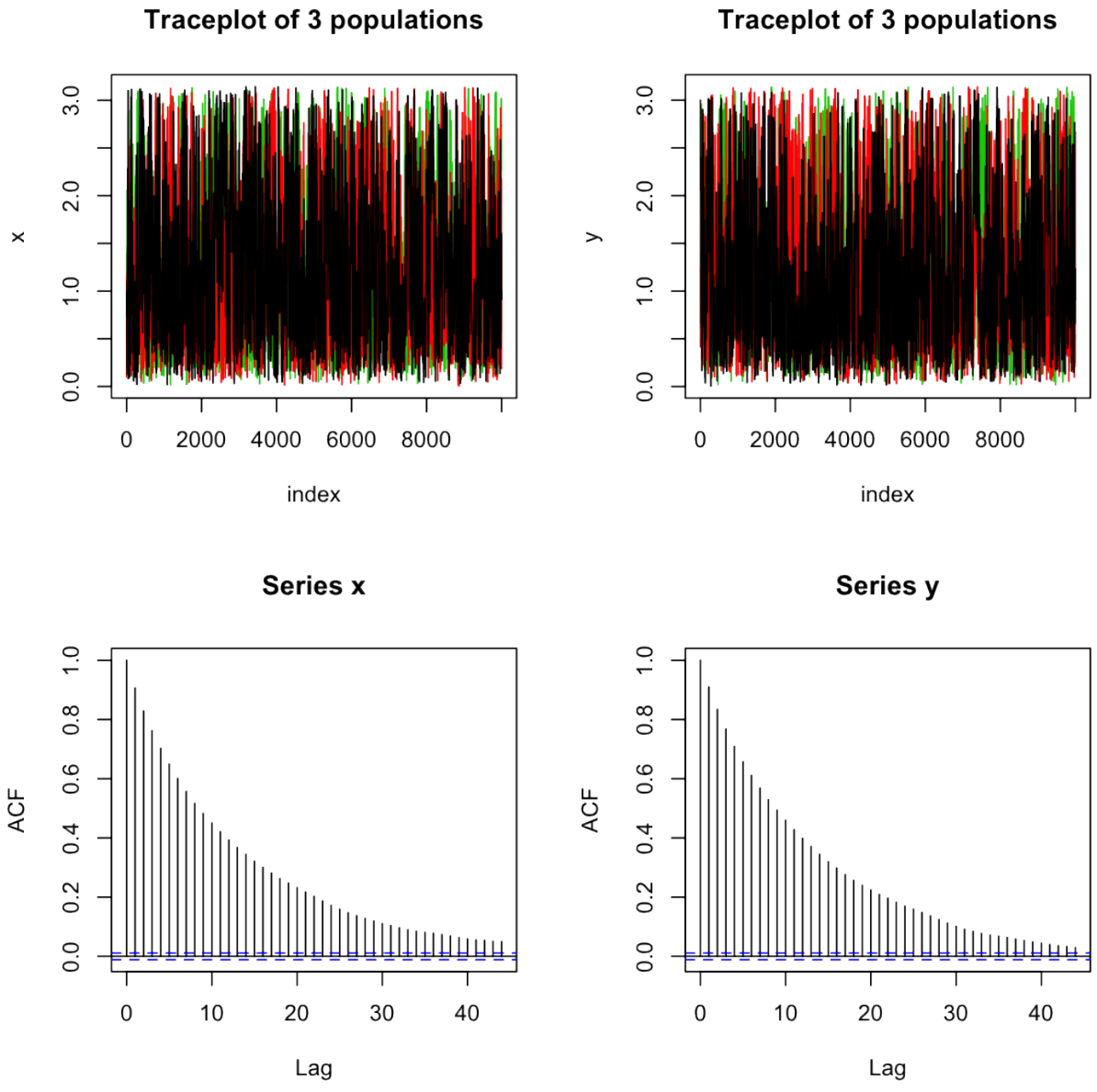
```
## Error:  invalid arguments
```

```
plot(chain3)
```

```
## Error:  object 'chain3' not found
```

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. [1]

```
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)
```



[1] 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.

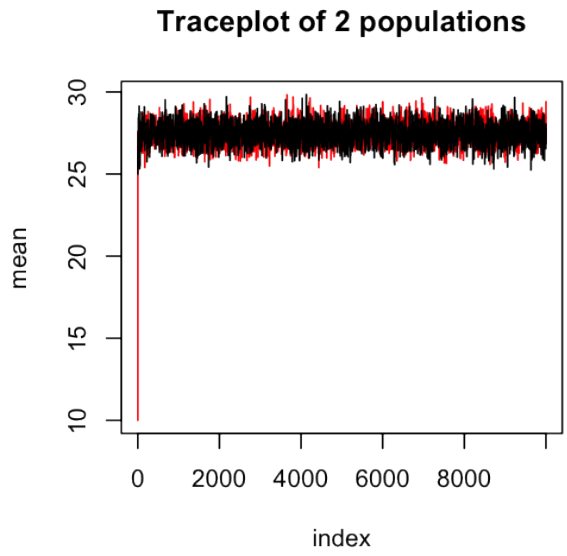
0.3 Gibbs sampling

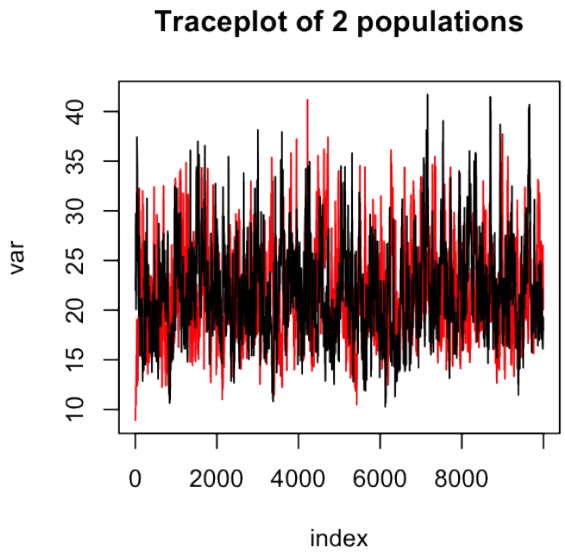
A gibbs updater calls several updating functions sequentially. Here we find Baysean location-scale parameter estimates for a t-distribution likelihood with known degrees of freedom 5/2. The model specification in this case is $y|\mu, w \sim N(\mu, w^{-1})$ and $w|\sigma^2, \nu \sim G(\nu/2, \sigma^2\nu/2)$, with priors $\mu \sim N$ and $\sigma^2 \sim G$.

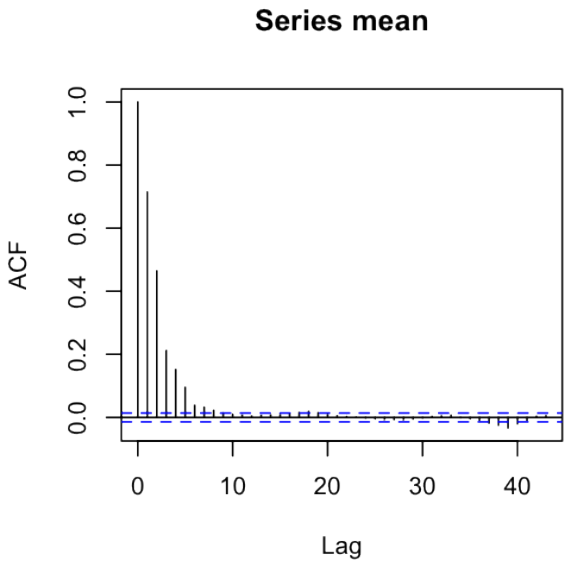
```
newcomb <- c(28, -44, 29, 30, 26, 27, 22, 23, 33, 16, 24, 29, 24, 40, 21, 31,
    34, -2, 25, 19, 24, 28, 37, 32, 20, 25, 25, 36, 36, 21, 28, 26, 32, 28,
    26, 30, 36, 29, 30, 22, 36, 27, 26, 28, 29, 23, 31, 32, 24, 27, 27, 27,
    32, 25, 28, 27, 26, 24, 32, 29, 28, 33, 39, 25, 16, 23)
##' Full conditional updater for mu
f.m <- function(state, ...) {
    k <- seq_len(ncol(state) - length(newcomb))
    w <- rowSums(state[, -k])
    mu <- (state[, -k] %*% newcomb)/(1e-04 + w)
    sd <- 1/sqrt(1e-04 + w)
    n <- nrow(state)
    state[, 1] <- rnorm(n, mu, sd)
    state
}
##' Full conditional updater for sigma^2
f.v <- function(state, df = FALSE) {
    nu <- if (df)
        1/state[, 3] else 5
    k <- seq_len(ncol(state) - length(newcomb))
    n <- ncol(state[, -k])
    rates <- 0.1 + rowSums(state[, -k]) * nu/2
    state[, 2] <- rgamma(nrow(state), 0.1 + n * nu/2, rates)
    state
}
##' Full conditional updater for w
f.w <- function(state, df = FALSE) {
    nu <- if (df)
        1/state[, 3] else 5
    k <- seq_len(ncol(state) - length(newcomb))
    rate <- (nu * state[, 2] + t(apply(state, 1, function(row) (newcomb - row[1])^2)))/2
    state[, -k] <- rgamma(length(state[, -k]), (nu + 1)/2, rate)
    state
}
```

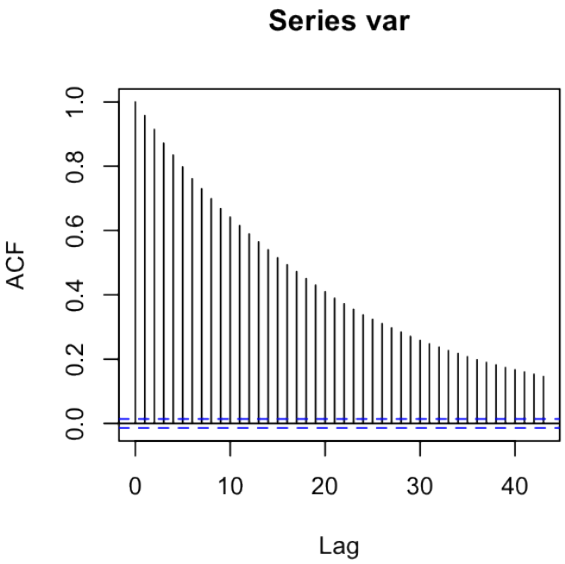
The gibbs function is used to create a function which acts as the Gibbs updater.

```
init <- rbind(c(mean = 25, var = 22, rgamma(length(newcomb), 5/2, 22 * 5/2)),
    c(10, 10, rgamma(length(newcomb), 5/2, 10 * 5/2)))
newcomb.gibbs <- gibbs(f.m, f.v, f.w)
chain5 <- iterate(10000, newcomb.gibbs, init)
chain5 <- prune(chain5, TRUE, 1:2)
plot(chain5)
```





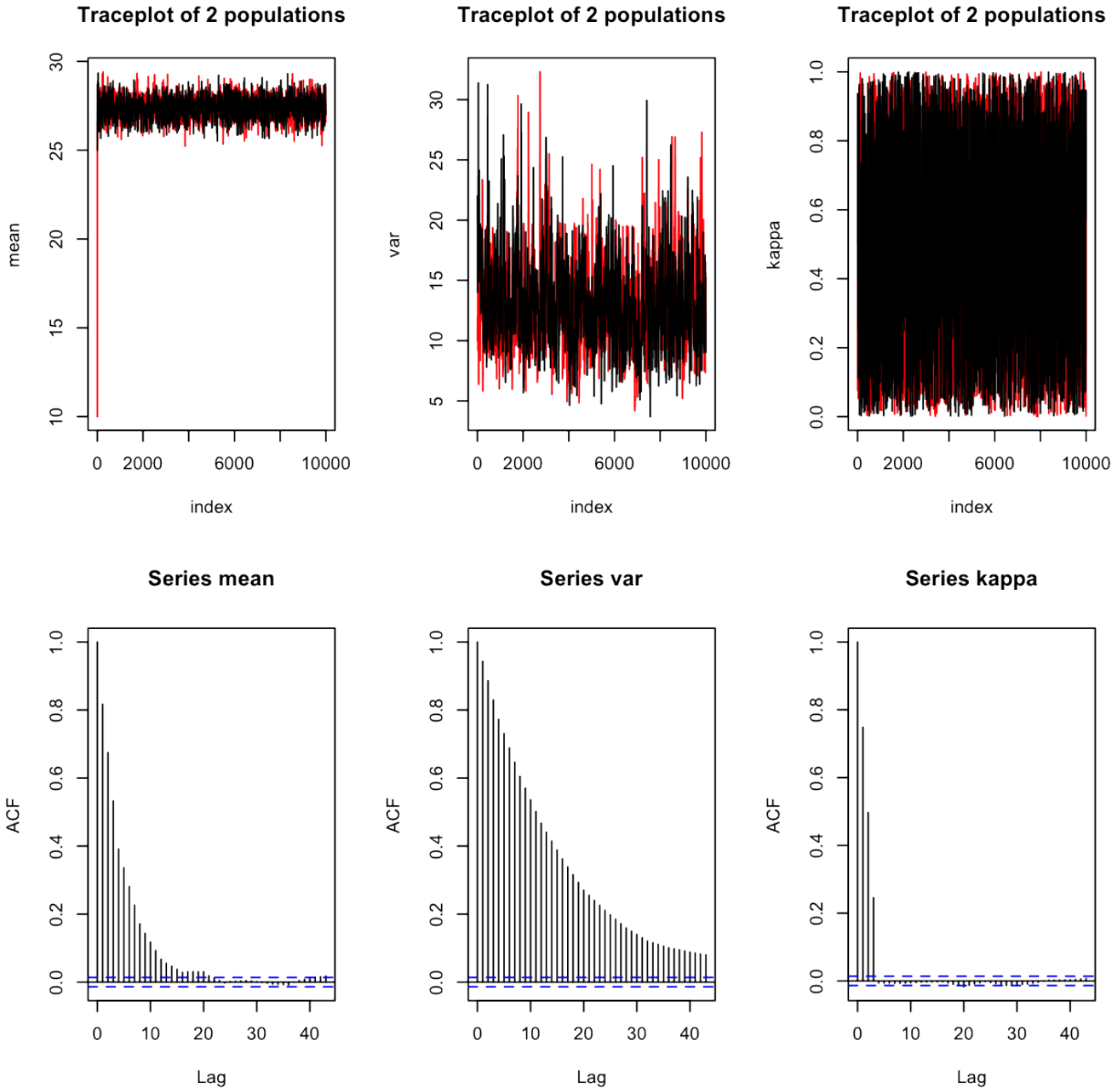




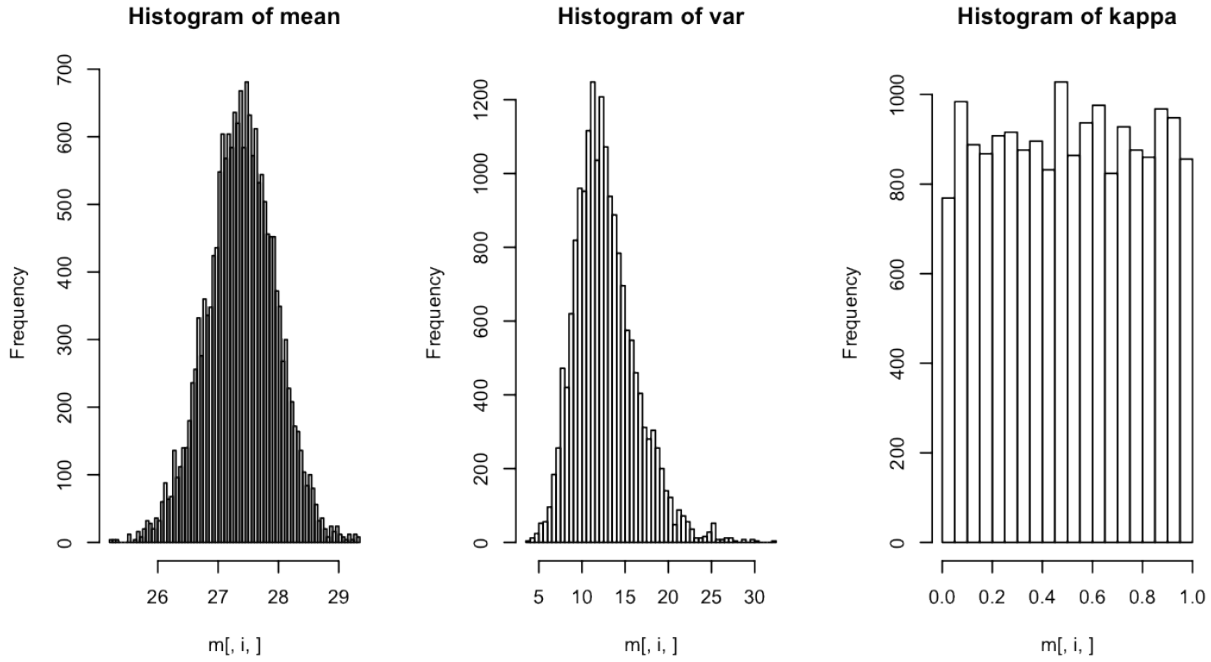

```
summary(chain5)

## Discarding first 1000 states.
##          mean    var
## mean  27.4908 21.995
## se     0.6638  4.782
## 2.5%  26.1954 14.250
## 97.5% 28.8291 33.006
```

Of course, you can use a metropolis updater in place of a full conditional. Here we put a prior on $\nu^{-1} \sim E(1)T(0,1)$

```
dtexp <- function(x, rate = 1, min = 0, max = Inf, log = FALSE) {
    d <- dexp(x, rate) * (x >= min) * (x <= max)
    c <- pexp(max, rate) - pexp(min, rate)
    if (log)
        log(d/c) else d/c
}
posterior <- function(state) {
    if (!is.matrix(state))
        state <- matrix(state, 1)
    f <- function(x) sum(dt((newcomb - x[1])/sqrt(x[2]), 1/x[3], log = TRUE) -
        log(x[2])/2)
    exp(apply(state, 1, f) + dnorm(state[, 1], sd = 100, log = TRUE) + dgamma(state[,
        2], 0.1, 0.1, log = TRUE) + dtexp(state[, 3], max = 1, log = TRUE))
}
kappa.prop <- function(state) {
    state[, 3] <- runif(nrow(state))
    state
}
init <- rbind(c(mean = 25, var = 22, kappa = 0.5, rgamma(length(newcomb), 5/2,
    22 * 5/2)), c(10, 10, 0.4, rgamma(length(newcomb), 5/2, 10 * 5/2)))
newcomb.chain3 <- gibbs(f.m, f.v, f.w, metropolis(posterior, kappa.prop))
chain4 <- iterate(10000, newcomb.chain3, init, df = TRUE)
chain4 <- prune(chain4, TRUE, 1:3)
summary(chain4)

## Discarding first 1000 states.
##          mean    var  kappa
## mean  27.3842 12.734 0.5026
## se     0.5751  3.550 0.2870
## 2.5%  26.2059  7.048 0.0316
## 97.5% 28.5032 20.956 0.9743

plot(chain4)
```




```
hist(chain4, breaks = "fd")

## Discarding first 1000 states.
```



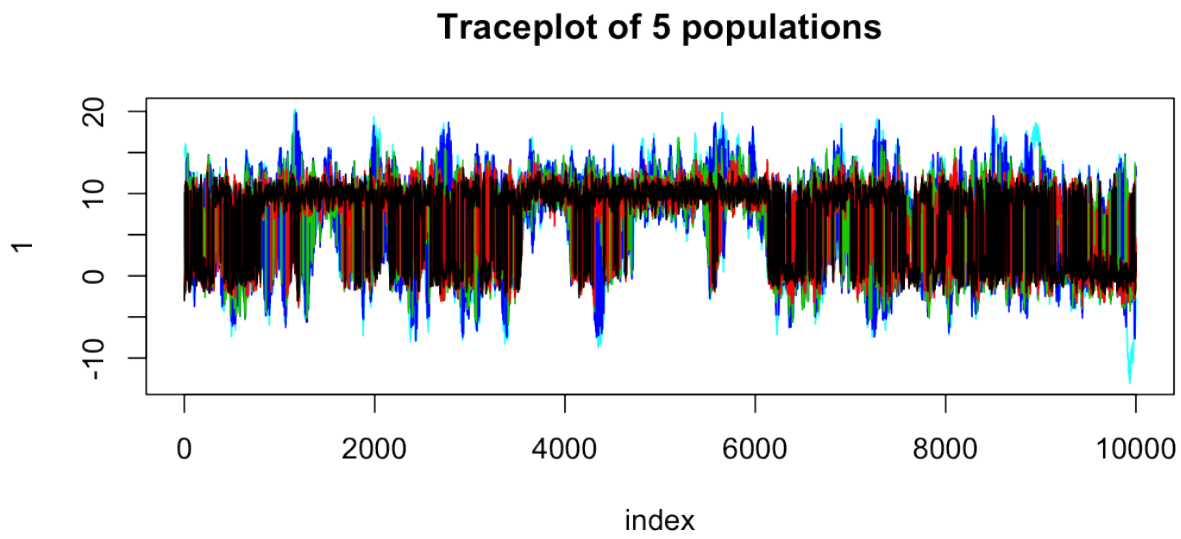
0.4 Parallel tempering

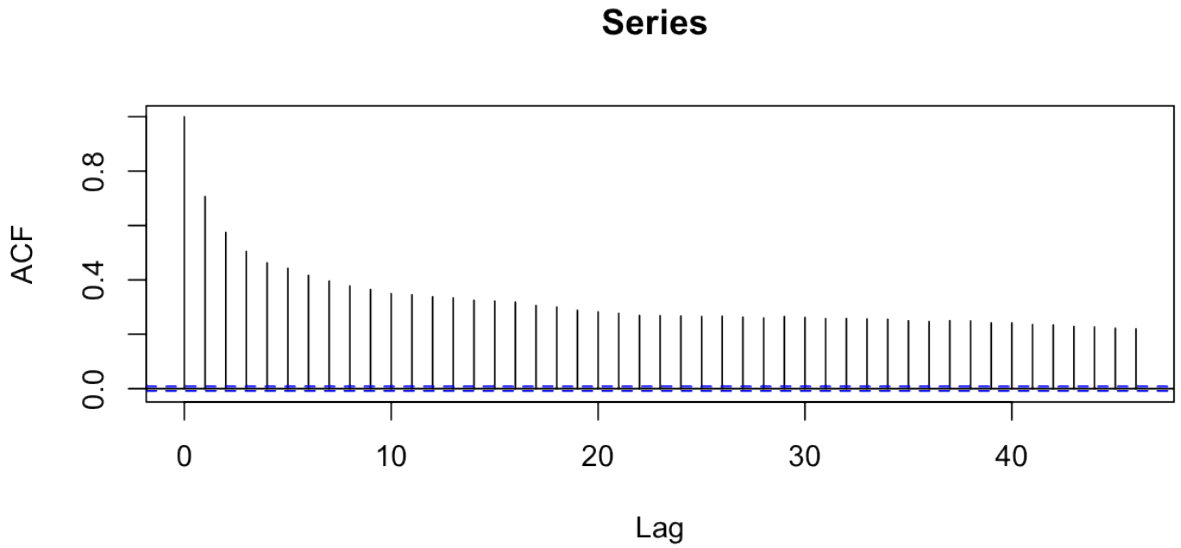
We want to sample from a distribution with separated modes. First we set up the target distribution function. This function should take a single individual and return the density of that individual.

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

Now we can set up the parallel tempering updater. Function temper is similar to metropolis, but you must also provide a list of temperatures. (It is assumed that the number of individuals in the population will be the same as the number of temperature levels.)

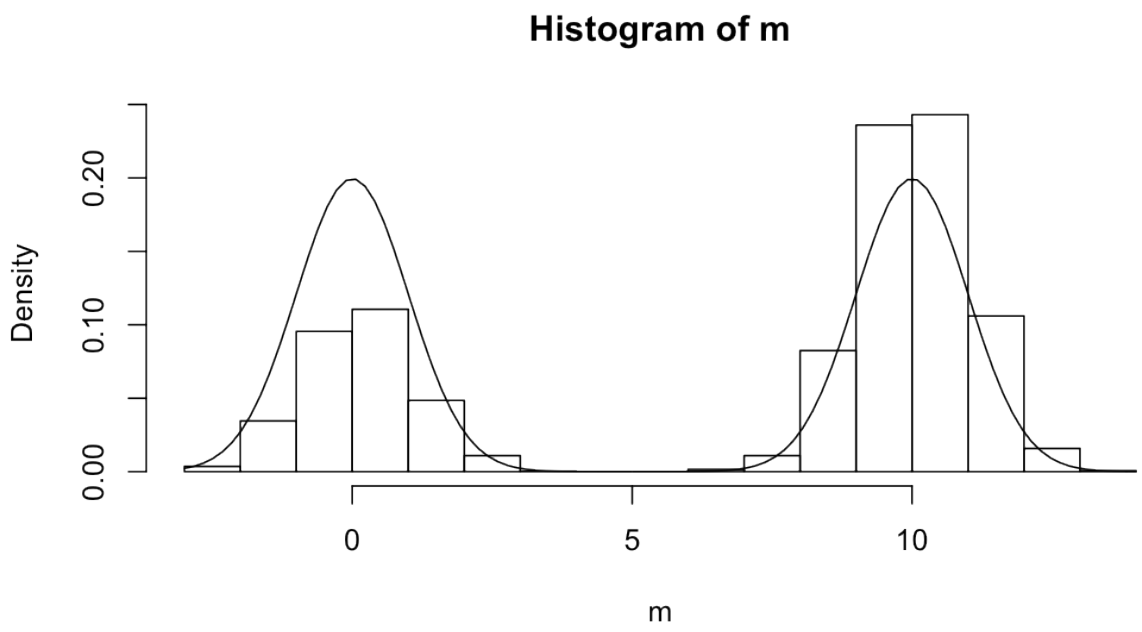
```
temps <- c(1, 2, 4, 8, 15)
updater <- mutate(heat(f, temps), propose)
init <- rbind(-3, 0, 5, 10, 15)
gch <- iterate(10000, updater, init)
plot(gch)
```






```
xx <- prune(gch, 1, 1)  #extract chain with correct distribution
```

```
hist(xx, freq = FALSE, breaks = "fd")

## Discarding first 1000 states.

curve((dnorm(x) + dnorm(x, 10))/2, add = TRUE)
```



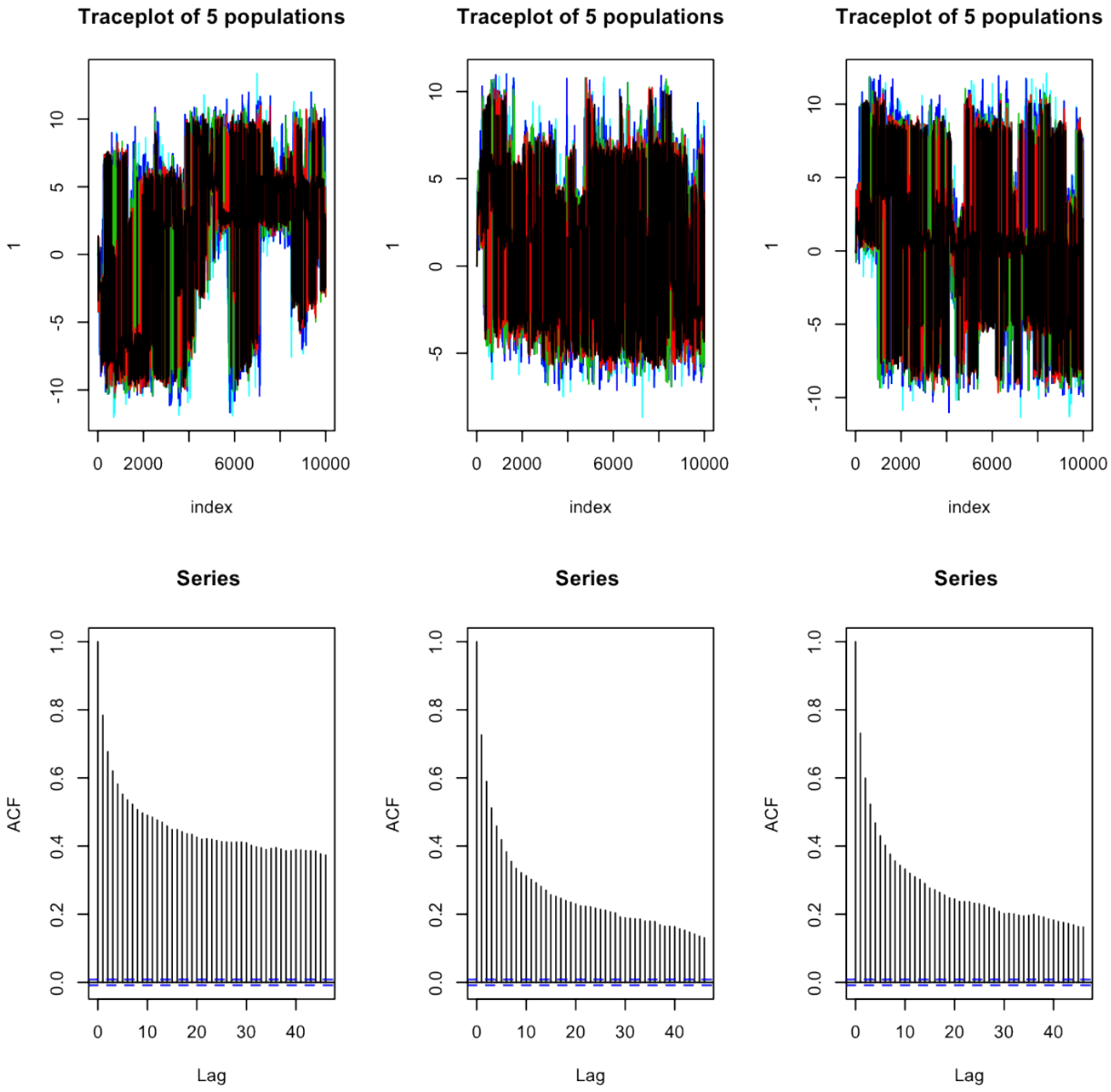
0.5 Evolutionary MC

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)
evolve <- gibbs(reproduce(ladder), mutate(ladder, runif3), p = c(1, 1))

init <- matrix(0, 5, 3)
runif(100)

##   [1] 0.520537 0.051885 0.391090 0.550450 0.416428 0.061312 0.359780
##   [8] 0.974365 0.149928 0.503230 0.251273 0.621008 0.176322 0.652557
##  [15] 0.904583 0.427867 0.544352 0.661578 0.950037 0.005459 0.362369
##  [22] 0.810577 0.851012 0.141126 0.007155 0.411257 0.236126 0.197149
##  [29] 0.941854 0.668238 0.177760 0.359667 0.331165 0.345900 0.454746
##  [36] 0.331254 0.439420 0.690425 0.167946 0.771433 0.098579 0.976490
##  [43] 0.214055 0.253464 0.615543 0.383458 0.889597 0.482701 0.303827
##  [50] 0.746164 0.789685 0.066584 0.769259 0.740305 0.954066 0.381051
##  [57] 0.950740 0.631242 0.071795 0.444387 0.578095 0.731366 0.759175
##  [64] 0.574198 0.989554 0.002832 0.113100 0.929381 0.572161 0.540584
##  [71] 0.199075 0.306391 0.537978 0.427963 0.858865 0.251045 0.472932
##  [78] 0.861176 0.824208 0.121688 0.476930 0.733241 0.048647 0.243439
##  [85] 0.840609 0.543595 0.355150 0.934983 0.924237 0.652280 0.484532
```

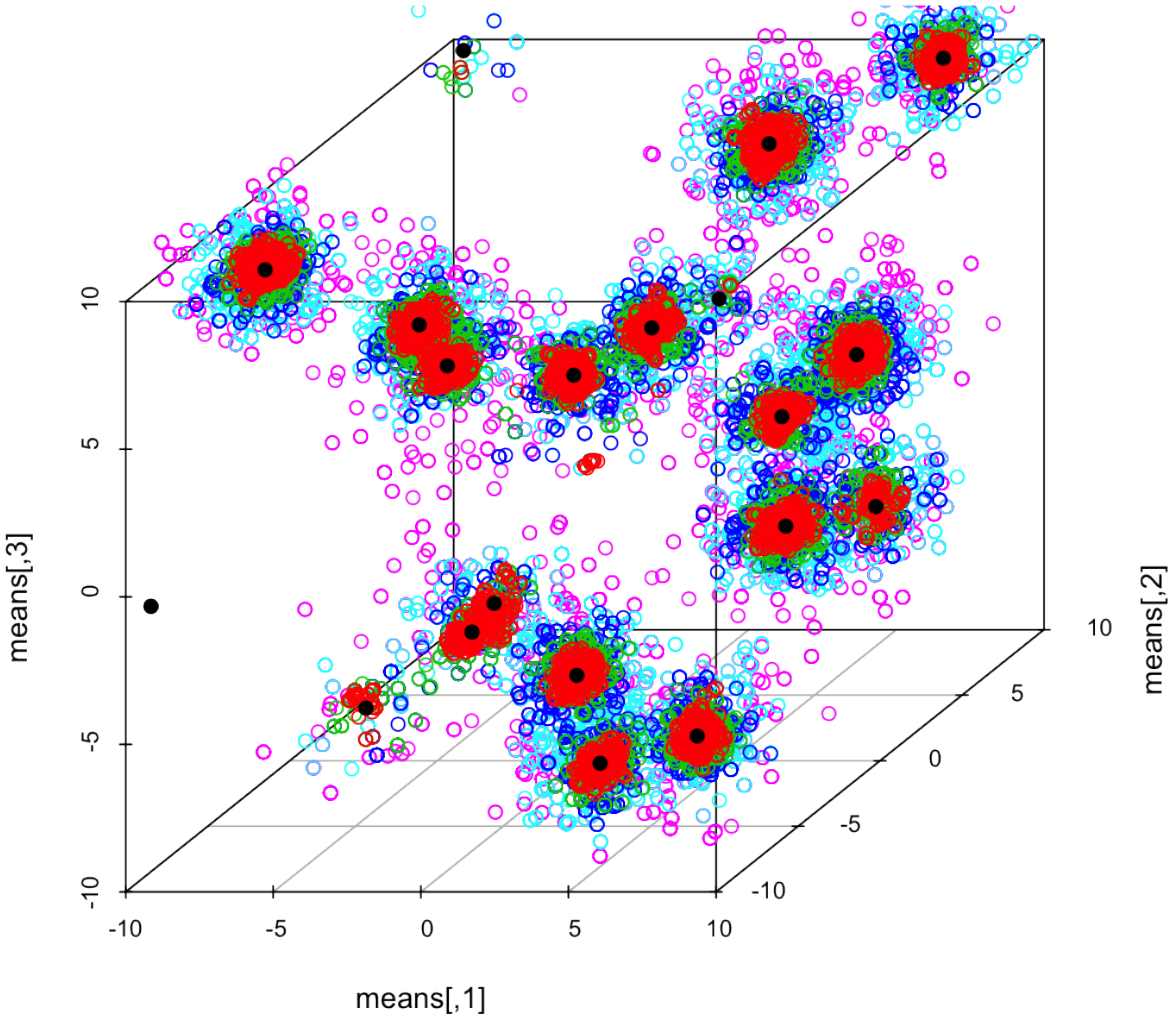
```
##  [92] 0.394657 0.939312 0.285683 0.593501 0.557517 0.875150 0.578077
##  [99] 0.815279 0.351948

lineage <- iterate(10000, evolve, init)
```

```
plot(lineage)
```

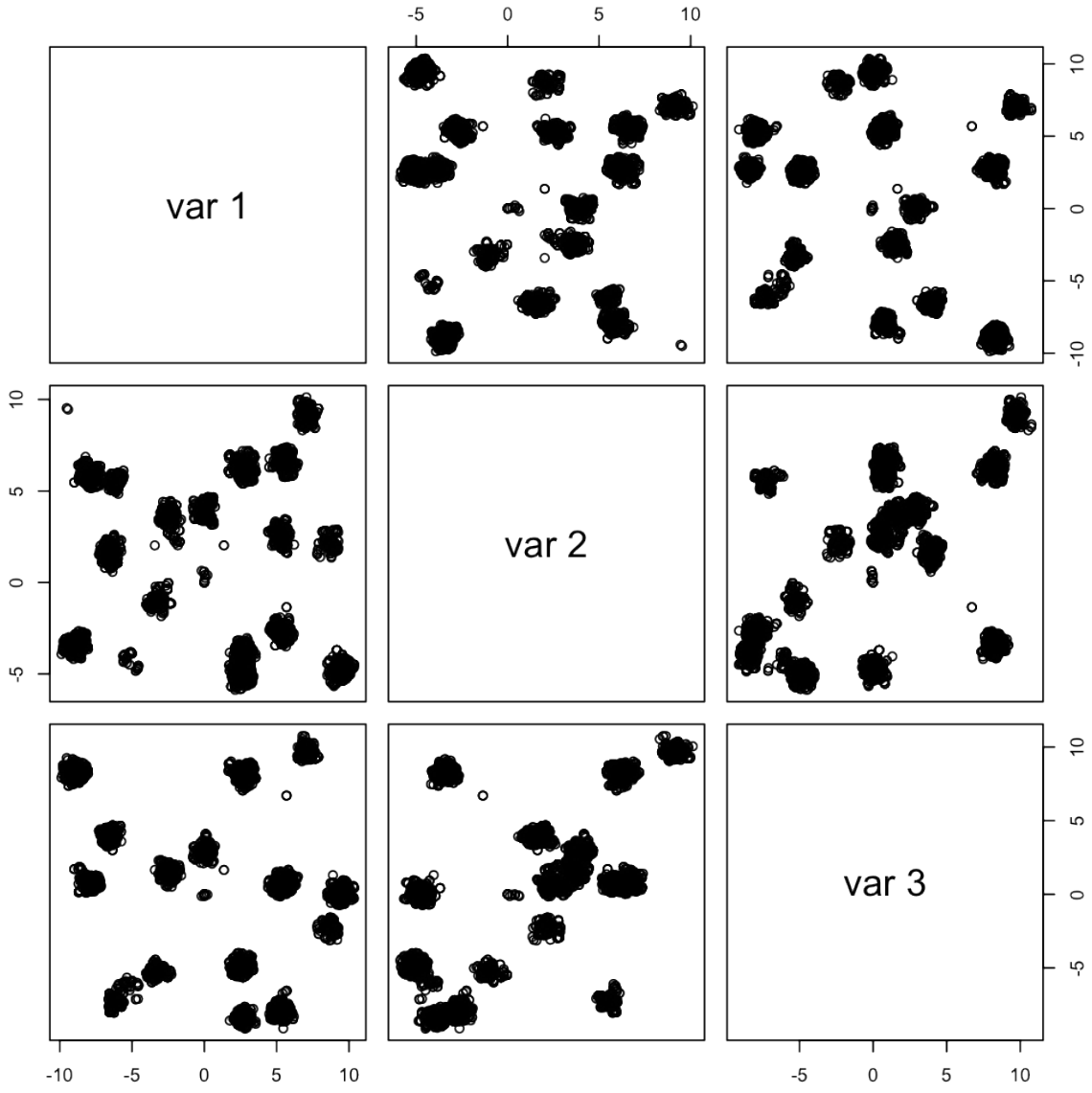



```
m <- aperm(simplify2array(lineage))
library(scatterplot3d)
pp <- scatterplot3d(means)
```

```
invisible(sapply(5:1, function(i) pp$points3d(m[, , i], col = i + 1)))
pp$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))
```