INTRODUCTION TO POMP BY EXAMPLE

AARON A. KING

1. A first example: a two-dimensional random walk.

In order to specify a partially-observed Markov process, we must define the process model and the measurement model. In particular, we will need to be able to simulate from and compute the p.d.f. of both models. The following function will simulate the process model. The documentation (`?pomp`) spells out the specifications for this function.

```
> rw.rprocess <- function(xstart, times, params,
+     ...) {
+     nsims <- ncol(params)
+     ntimes <- length(times)
+     dt <- diff(times)
+     x <- array(0, dim = c(2, nsims, ntimes))
+     rownames(x) <- rownames(xstart)
+     noise.sds <- params[c("s1", "s2"), ]
+     x[, , 1] <- xstart
+     for (j in 2:ntimes) {
+         x[, , j] <- rnorm(n = 2 * nsims, mean = x[,
+             , j - 1], sd = noise.sds * dt[j -
+             1])
+     }
+     x
+ }
```

Some methods will require the probability density of a given state transition. The function `dprocess` will evaluate this for a sequences of state transitions.

```
> rw.dprocess <- function(x, times, params, log = FALSE,
+     ...) {
+     nsims <- ncol(params)
+     ntimes <- length(times)
+     dt <- diff(times)
+     d <- array(0, dim = c(2, nsims, ntimes - 1))
+     noise.sds <- params[c("s1", "s2"), ]
+     for (j in 2:ntimes) d[, , j - 1] <- dnorm(x[,
+         , j] - x[, , j - 1], mean = 0, sd = noise.sds *
+         dt[j - 1], log = TRUE)
+     if (log) {
+         apply(d, c(2, 3), sum)
+     }
+     else {
+         exp(apply(d, c(2, 3), sum))
+     }
+ }
```

Date: May 14, 2008.

Now we specify the function that will simulate the measurement process. Again, the documentation spells out how.

```
> bvnorm.rmeasure <- function(x, times, params,
+     ...) {
+     nsims <- dim(x)[2]
+     ntimes <- dim(x)[3]
+     y <- array(0, dim = c(2, nsims, ntimes))
+     rownames(y) <- c("y1", "y2")
+     for (j in 1:nsims) {
+         for (k in 1:ntimes) {
+             y[, j, k] <- rnorm(2, mean = x[, j,
+                 k], sd = params["tau", j])
+         }
+     }
+     y
+ }
```

Finally, we have to specify how to evaluate the likelihood of an observation given the underlying state.

```
> bvnorm.dmeasure <- function(y, x, times, params,
+     log = FALSE, ...) {
+     d1 <- dnorm(x = y["y1", ], mean = x["x1",
+         , ], sd = params["tau", ], log = TRUE)
+     d2 <- dnorm(x = y["y2", ], mean = x["x2",
+         , ], sd = params["tau", ], log = TRUE)
+     if (log) {
+         d1 + d2
+     }
+     else {
+         exp(d1 + d2)
+     }
+ }
```

The following builds a `pomp` object called `rw2`.

```
> rw2 <- pomp(rprocess = rw.rprocess, dprocess = rw.dprocess,
+     rmeasure = bvnorm.rmeasure, dmeasure = bvnorm.dmeasure,
+     times = 1:100, data = rbind(y1 = rep(0, 100),
+         y2 = rep(0, 100)), t0 = 0, useless = 23)
```

Now we'll specify some parameters and initial states.

```
> p <- rbind(s1 = c(2, 2, 3), s2 = c(0.1, 1, 2),
+     tau = c(1, 5, 0), x1.0 = c(0, 0, 5), x2.0 = c(0,
+         0, 0))
> p
     [,1] [,2] [,3]
s1    2.0    2    3
s2    0.1    1    2
tau   1.0    5    0
x1.0  0.0    0    5
x2.0  0.0    0    0
```

Each column is a different initial state or parameter vector. Note that we must use rownames! Notice also that we parameterize the initial states by means of parameters with names ending in ".0".

When we defined `rw2`, the data were all missing. We can generate simulated data by:

```
> examples <- simulate(rw2, params = p)
> rw2 <- examples[[1]]
```

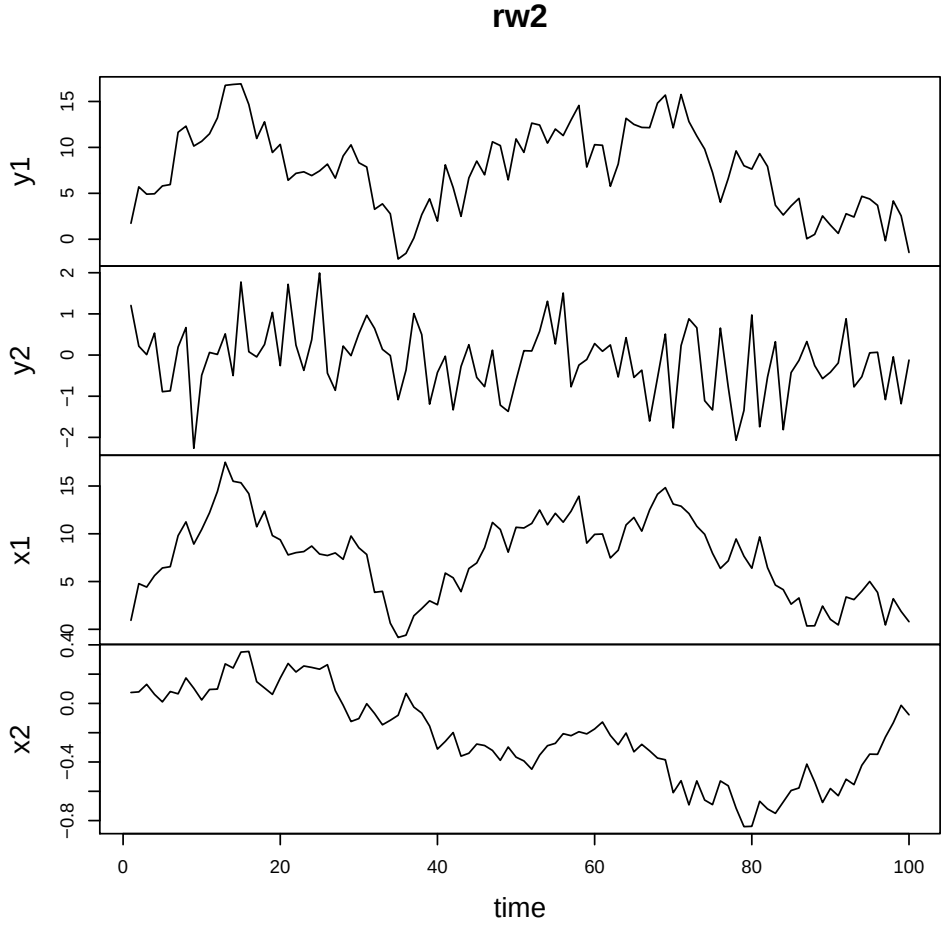

FIGURE 1. A `plot` method exists for `pomp` objects.

By default `simulate` will generate a list of new `pomp` objects. It can also be used to obtain the simulated state and/or measurement trajectories:

```
> x <- simulate(rw2, params = p, states = T)
> y <- simulate(rw2, params = p, obs = T)
> y <- simulate(rw2, params = p, obs = T, states = T,
+     nsim = 10)
```

A `plot` method exists for `pomp` objects (Fig. 1).

Access to the individual components of the `pomp` object is available by means of a few *methods*. To extract the data and the observation times, use `data.array` and `time`, respectively:

```
> x <- data.array(rw2)
> t <- time(rw2)
```

It's also possible to coerce a `pomp` object to a `data.frame`:

```
> z <- as(rw2, "data.frame")
> names(z)
[1] "time" "y1"   "y2"   "x1"   "x2"
```

To run the process model, users should use `simulate` with the `states=T` option. Lower-level access to the process model is available via the `rprocess` method:

```
> x0 <- init.state(rw2, params = p)
> x <- rprocess(rw2, xstart = x0, times = 0:100,
+     params = p)
```

Note that we use the low-level `init.state` method to initialize the unobserved state. Similarly, low-level access to the measurement-model simulator can be had through `rmeasure`:

```
> y <- rmeasure(rw2, x = x, times = 0:100, params = p)
```

Access to the process-model p.d.f. is available via `dprocess`:

```
> log(dprocess(rw2, x[, , 6:11], times = 5:10, params = p))
          [,1]       [,2]       [,3]       [,4]       [,5]
[1,] -1.437801 -0.9863767 -0.7196126 -0.8879575 -0.5433848
[2,] -2.776012 -2.6539181 -3.4014757 -2.9845915 -2.5575773
[3,] -4.398925 -4.0934394 -4.1382440 -4.8650012 -4.1257058
> dprocess(rw2, x[, , 6:11], times = 5:10, params = p,
+     log = T)
          [,1]       [,2]       [,3]       [,4]       [,5]
[1,] -1.437801 -0.9863767 -0.7196126 -0.8879575 -0.5433848
[2,] -2.776012 -2.6539181 -3.4014757 -2.9845915 -2.5575773
[3,] -4.398925 -4.0934394 -4.1382440 -4.8650012 -4.1257058
```

Note that `dprocess` returns a matrix: the rows correspond to independent simulations, the columns to distinct state transitions. The measurement-model p.d.f. is accessed via the `dmeasure` method, which like `dprocess`, returns a matrix. The rows correspond to independent simulations, the columns to distinct times.

```
> dmeasure(rw2, y = y[, 1, 1:4], x = x[, , 1:4,
+     drop = F], times = time(rw2)[1:4], p)
            [,1]        [,2]        [,3]        [,4]
[1,] 0.068269456 0.028181544 0.132531016 0.005889303
[2,] 0.005796973 0.004926193 0.003140046 0.003672915
[3,] 0.000000000 0.000000000 0.000000000 0.000000000
> dmeasure(rw2, y = y[, 2, 1:4], x = x[, , 1:4,
+     drop = F], times = time(rw2)[1:4], p)
             [,1]         [,2]         [,3]         [,4]
[1,] 8.546886e-11 0.0008184712 3.464838e-38 2.049698e-18
[2,] 9.185328e-04 0.0009025457 1.632805e-03 8.888404e-04
[3,] 0.000000e+00 0.0000000000 0.000000e+00 0.000000e+00
> log(dmeasure(rw2, y = y[, 3, 1:4], x = x[, , 1:4,
+     drop = F], times = time(rw2)[1:4], p))
           [,1]      [,2]      [,3]      [,4]
[1,] -14.337877 -64.82906 -78.34774 -57.69180
[2,]  -6.797482  -6.72047 -12.25143 -10.91024
[3,]       -Inf       Inf      -Inf       Inf
> dmeasure(rw2, y = y[, 3, 1:4], x = x[, , 1:4,
+     drop = F], times = time(rw2)[1:4], p, log = T)
           [,1]      [,2]      [,3]      [,4]
[1,] -14.337877 -64.82906 -78.34774 -57.69180
[2,]  -6.797482  -6.72047 -12.25143 -10.91024
[3,]       -Inf       Inf      -Inf       Inf
```

2. A two-dimensional Ornstein-Uhlenbeck process.

To keep things simple, we will study a discrete-time process. The tricks below will continue to be useful even in the case of a continuous-time process, but the computational effort will be greater. The unobserved

Ornstein-Uhlenbeck (OU) process $X_t \in \mathbb{R}^2$ satisfies

$$X_t = A\,X_{t-1} + \xi_t.$$

The observation process is

$$Y_t = B\,X_t + \varepsilon_t.$$

In these equations, A and and B are 2×2 constant matrices; ξ_t and ε_t are mutually-independent families of i.i.d. bivariate normal random variables. We let $\sigma\sigma^T$ be the variance-covariance matrix of ξ_t, where σ is lower-triangular; likewise, we let $\tau\tau^T$ be that of ε_t.

We build the `pomp` object by specifying the three basic elements. The process model simulator and density functions:

```
> ou2.rprocess <- function(xstart, times, params,
+     ...) {
+     nsims <- ncol(xstart)
+     ntimes <- length(times)
+     alpha <- array(params[c("alpha.1", "alpha.2",
+         "alpha.3", "alpha.4"), ], dim = c(2, 2,
+         nsims))
+     sigma <- array(params[c("sigma.1", "sigma.2",
+         "sigma.2", "sigma.3"), ], dim = c(2, 2,
+         nsims))
+     sigma[1, 2, ] <- 0
+     x <- array(0, dim = c(2, nsims, ntimes))
+     rownames(x) <- rownames(xstart)
+     x[, , 1] <- xstart
+     for (k in 1:nsims) {
+         for (j in 2:ntimes) {
+             x[, k, j] <- alpha[, , k] %*% x[,
+                 k, j - 1] + sigma[, , k] %*% rnorm(2)
+         }
+     }
+     x
+ }
> ou2.dprocess <- function(x, times, params, log = FALSE,
+     ...) {
+     nsims <- ncol(x)
+     ntimes <- length(times)
+     alpha <- array(params[c("alpha.1", "alpha.2",
+         "alpha.3", "alpha.4"), ], dim = c(2, 2,
+         nsims))
+     sigma <- array(params[c("sigma.1", "sigma.2",
+         "sigma.2", "sigma.3"), ], dim = c(2, 2,
+         nsims))
+     sigma[1, 2, ] <- 0
+     d <- array(0, dim = c(nsims, ntimes - 1))
+     for (k in 1:nsims) {
+         for (j in 2:ntimes) {
+             z <- forwardsolve(sigma[, , k], x[,
+                 k, j] - alpha[, , k] %*% x[, k,
+                 j - 1])
+             if (log) {
+                 d[k, j - 1] <- sum(dnorm(z, mean = 0,
+                   sd = 1, log = TRUE))
+             }
```

```
+             else {
+                 d[k, j - 1] <- exp(sum(dnorm(z,
+                   mean = 0, sd = 1, log = TRUE)))
+             }
+         }
+     }
+     d
+ }
```

The measurement model is the same as that for the random walk example above. We build the pomp object:

```
> ou2 <- pomp(times = seq(1, 100), data = rbind(y1 = rep(0,
+     100), y2 = rep(0, 100)), t0 = 0, rprocess = ou2.rprocess,
+     dprocess = ou2.dprocess, rmeasure = bvnorm.rmeasure,
+     dmeasure = bvnorm.dmeasure)
```

Now we'll specify the "true" parameters and initial states.

```
> p <- c(alpha.1 = 0.9, alpha.2 = 0, alpha.3 = 0,
+     alpha.4 = 0.99, sigma.1 = 1, sigma.2 = 0,
+     sigma.3 = 2, tau = 1, x1.0 = 50, x2.0 = -50)
alpha.1 alpha.2 alpha.3 alpha.4 sigma.1 sigma.2 sigma.3
   0.90    0.00    0.00    0.99    1.00    0.00    2.00
    tau    x1.0    x2.0
   1.00   50.00  -50.00
```

As before, we'll fill in the missing values with simulated data.

```
> tic <- Sys.time()
> ou2 <- simulate(ou2, params = p, nsim = 1000)
> toc <- Sys.time()
> print(toc - tic)
Time difference of 9.805371 secs
> ou2 <- ou2[[1]]
```

Let's make sure everything works.

```
> x0 <- init.state(ou2, params = p)
> x <- rprocess(ou2, xstart = as.matrix(x0), times = c(0,
+     time(ou2)), params = as.matrix(p))
> y <- rmeasure(ou2, x = x[, , -1, drop = F], times = time(ou2),
+     params = as.matrix(p))
> dprocess(ou2, x[, , 36:41, drop = F], times = time(ou2)[35:40],
+     params = as.matrix(p))
           [,1]      [,2]       [,3]      [,4]       [,5]
[1,] 0.03862098 0.1329115 0.01107108 0.0235313 0.07949738
> dmeasure(ou2, y = y[, 1, 1:4], x = x[, , 2:5,
+     drop = F], times = time(ou2)[1:4], params = as.matrix(p))
[1] 0.08517573 0.06644739 0.14777204 0.03340110
```

The codes above show how the pomp object is constructed. Being written in R, however, they're not particularly fast. To maximize computational efficiency, we'll instead use compiled codes. For more information on this topic, see the vignette, "Using compiled code in pomp".

```
> data(ou2)
```

3. Particle filter.

We can run a particle filter as follows:

```
> fit1 <- pfilter(ou2, params = p, Np = 1000, filter.mean = T,
+     pred.mean = T, pred.var = T)
```

Since `ou2` already contained the parameters `p`, it wasn't necessary to specify them; we could have done

```
> fit1 <- pfilter(ou2, Np = 1000)
```

for example. We can compare the results against those of the Kalman filter, which is exact in this case. First, we need to implement the Kalman filter.

```
> kalman.filter <- function(y, x0, a, b, sigma,
+     tau) {
+     n <- nrow(y)
+     ntimes <- ncol(y)
+     sigma.sq <- sigma %*% t(sigma)
+     tau.sq <- tau %*% t(tau)
+     inv.tau.sq <- solve(tau.sq)
+     cond.dev <- numeric(ntimes)
+     filter.mean <- matrix(0, n, ntimes)
+     pred.mean <- matrix(0, n, ntimes)
+     pred.var <- array(0, dim = c(n, n, ntimes))
+     dev <- 0
+     m <- x0
+     v <- diag(0, n)
+     for (k in seq(length = ntimes)) {
+         pred.mean[, k] <- M <- a %*% m
+         pred.var[, , k] <- V <- a %*% v %*% t(a) +
+             sigma.sq
+         q <- b %*% V %*% t(b) + tau.sq
+         r <- y[, k] - b %*% M
+         cond.dev[k] <- n * log(2 * pi) + log(det(q)) +
+             t(r) %*% solve(q, r)
+         dev <- dev + cond.dev[k]
+         q <- t(b) %*% inv.tau.sq %*% b + solve(V)
+         v <- solve(q)
+         filter.mean[, k] <- m <- v %*% (t(b) %*%
+             inv.tau.sq %*% y[, k] + solve(V, M))
+     }
+     list(pred.mean = pred.mean, pred.var = pred.var,
+         filter.mean = filter.mean, cond.loglik = -0.5 *
+             cond.dev, loglik = -0.5 * dev)
+ }
```

Now we can run it on the example data we generated above.

```
> y <- data.array(ou2)
> a <- matrix(p[c("alpha.1", "alpha.2", "alpha.3",
+     "alpha.4")], 2, 2)
> b <- diag(1, 2)
> sigma <- matrix(c(p["sigma.1"], p["sigma.2"],
+     0, p["sigma.3"]), 2, 2)
> tau <- diag(p["tau"], 2, 2)
> fit2 <- kalman.filter(y, x0, a, b, sigma, tau)
```

In this case, the Kalman filter gives us a log likelihood of `fit2$loglik=-422.4`, while the particle filter gives us `fit1$loglik=-421.8`.

4. The MIF algorithm

In order to use MIF, we need to specify the distribution of particles in the state-parameter space. This distribution must be such that, when `sd=0`, all the particles are identical. For this example, we'll just use the default particle distribution, which draws particles from a multivariate normal distribution.

We'll run MIF to maximize the likelihood over two of the parameters and the initial conditions. We'll use 1000 particles, an exponential cooling factor of 0.95, and a fixed-lag smoother with lag 10 for the initial conditions:

```
> alg.pars <- list(Np = 1000, var.factor = 1, ic.lag = 10,
+     cooling.factor = 0.95)
```

Just to make it interesting, we'll start far from the true parameter values:

```
> start.p <- p
> start.p[c("x1.0", "x2.0", "alpha.1", "alpha.4")] <- c(45,
+     -60, 0.8, 0.9)
> tic <- Sys.time()
> fit <- mif(ou2, Nmif = 1, start = start.p, pars = c("alpha.1",
+     "alpha.4"), ivps = c("x1.0", "x2.0"), rw.sd = c(x1.0 = 5,
+     x2.0 = 5, alpha.1 = 0.1, alpha.2 = 0, alpha.3 = 0,
+     alpha.4 = 0.1, sigma.1 = 0, sigma.2 = 0, sigma.3 = 0,
+     tau = 0), alg.pars = alg.pars, max.fail = 100)
> fit <- continue(fit, Nmif = 79, max.fail = 100)
> toc <- Sys.time()
> print(toc - tic)
Time difference of 1.898409 mins
> coef(fit)
    alpha.1     alpha.2     alpha.3     alpha.4     sigma.1
  0.8994751   0.0000000   0.0000000   0.9715443   1.0000000
    sigma.2     sigma.3         tau        x1.0        x2.0
  0.0000000   2.0000000   1.0000000  51.5505368 -50.5808232
```

One can plot various diagnostics for the fitted `mif` object using

```
> plot(fit)
```

Here, we'll just plot the convergence records for the log likelihood and the two α parameters (Fig. 2). In applications, a good strategy is to start several MIFs from different starting points. A good diagnostic for convergence is obtained by plotting the *convergence records* (see the documentation for `conv.rec`) and verifying that all the MIF iterations converge to the same parameters. One plots these—and other—diagnostics using `compare.mif` applied to a list of `mif` objects.

The log likelihood of the random-parameter model at the end of the mif iterations—which should be a rough approximation of that of the fixed-parameter model—is `logLik(fit)=-423.6`. To get the log likelihood of the fixed-parameter model (up to Monte Carlo error) we can use `pfilter`:

```
> round(pfilter(fit)$loglik, 1)
[1] -422
```

Like `pomp` objects, one can simulate from a fitted `mif` object (Fig. 3). In this case, the `pomp` is simulated at the MLE.

A. A. King, Departments of Ecology & Evolutionary Biology and Mathematics, University of Michigan, Ann Arbor, Michigan 48109-1048 USA

E-mail address: `kingaa at umich dot edu`

URL: `http://www.umich.edu/~kingaa`

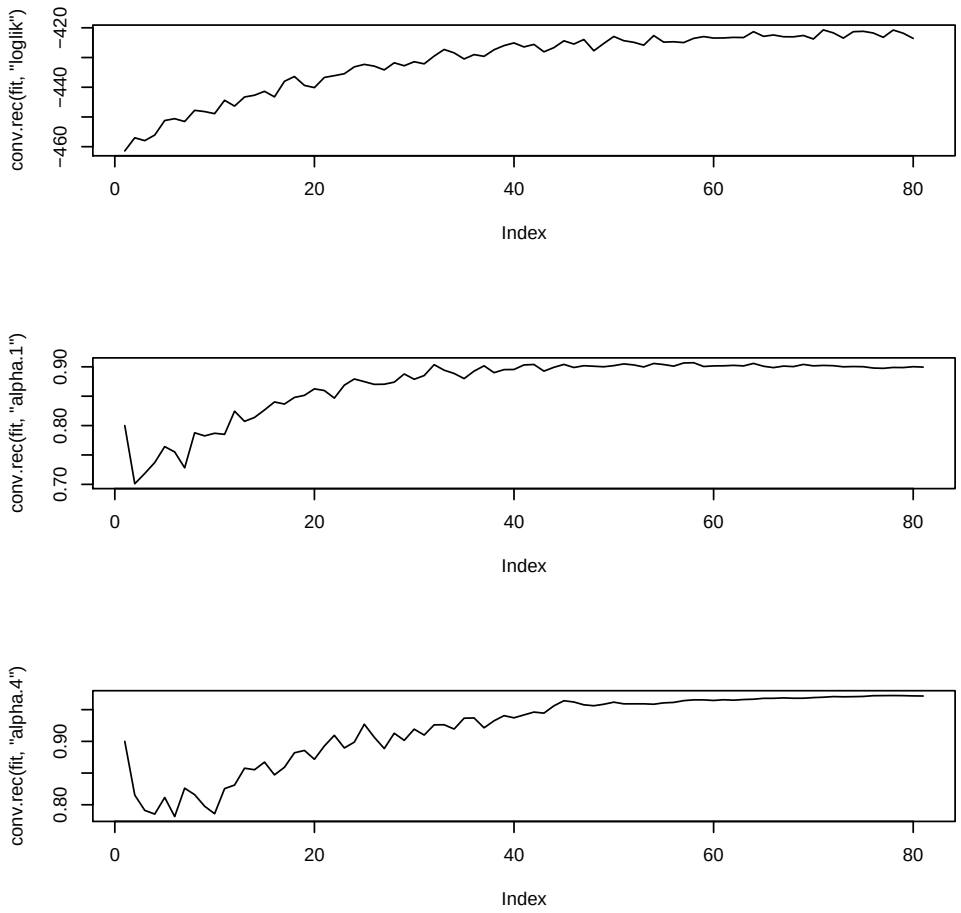

FIGURE 2. Convergence plots can be used to help diagnose convergence of the MIF algorithm.

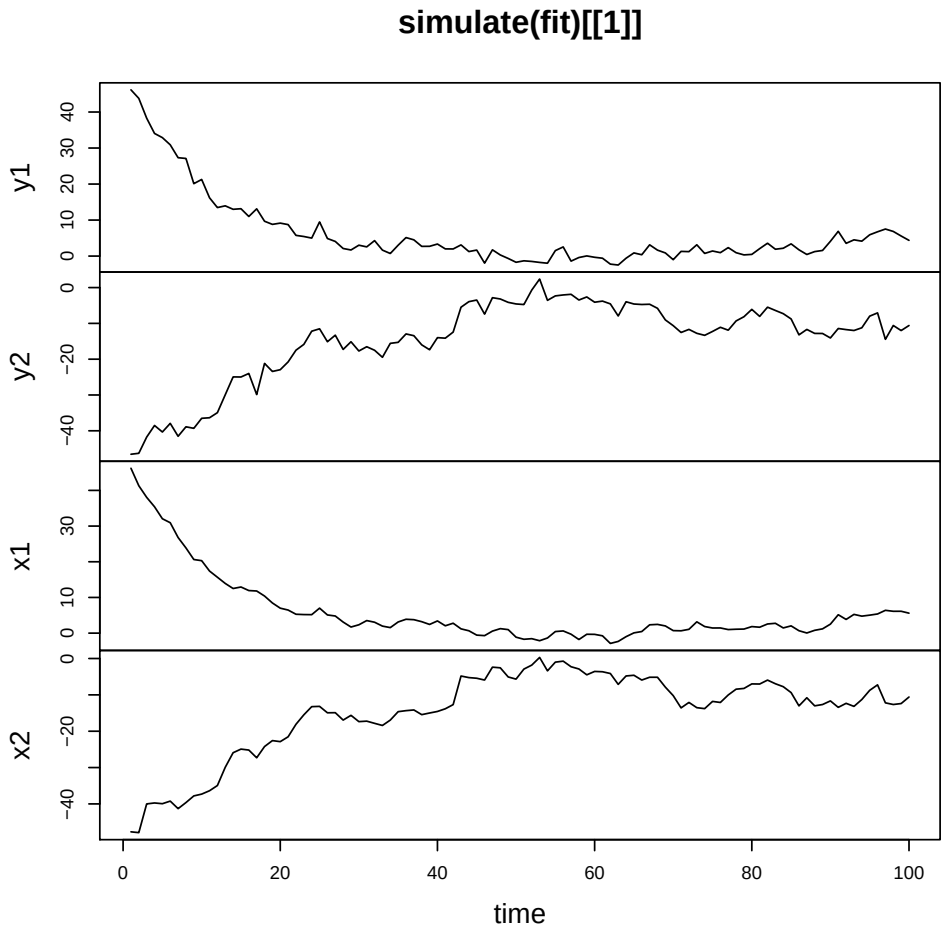

FIGURE 3. `mif` objects can be simulated.