INTRODUCTION TO POMP BY EXAMPLE

AARON A. KING

CONTENTS

1. A FIRST EXAMPLE: THE TWO-DIMENSIONAL ORNSTEIN-UHLENBECK PROCESS.

To keep things simple, we will study a discrete-time process. The pomp package is designed with continuous-time processes in mind, but the associated computational effort is typically greater, and the additional complexities are best postponed until the structure and usage of the package is understood. 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.

In order to specify this partially-observed Markov process, we must implement both the process model (i.e., the unobserved process) and the measurement model (the observation process). In particular, we will need to be able to both simulate from and compute the p.d.f. of each of these models. In pomp, one constructs an object of class pomp to hold functions that do all of these things, together with data and other information. The documentation (?pomp) spells out the usage of the pomp constructor, including detailed specifications for all its arguments and a worked example.

1.1. **Building the pomp object.** We build a pomp object by specifying the four basic elements mentioned above. First, we write a function that implements the process model simulator. In this function, we assume that:

(1) xstart will be a matrix, each column of which is a vector of initial values of the state process;
(2) params will be a matrix, the columns of which are parameter vectors;
(3) times will be a vector of times at which realizations of the state process are required. In particular, times[1] is the initial time (corresponding to xstart).

```
ou2.rprocess <- function(xstart, times, params,
    ...) {
    nreps <- ncol(xstart)
    ntimes <- length(times)
    x <- array(0, dim = c(2, nreps, ntimes))
    rownames(x) <- rownames(xstart)
    x[, , 1] <- xstart
    for (k in 2:ntimes) {
        for (j in 1:nreps) {
            eps <- rnorm(2, mean = 0, sd = 1)
            x["x1", j, k] <- params["alpha.1",
                j] * x["x1", j, k - 1] + params["alpha.3",
                j] * x["x2", j, k - 1] + params["sigma.1",
                j] * eps[1]
            x["x2", j, k] <- params["alpha.2",
                j] * x["x1", j, k - 1] + params["alpha.4",
                j] * x["x2", j, k - 1] + params["sigma.2",
                j] * eps[1] + params["sigma.3",
                j] * eps[2]
        }
    }
    x
}
```

Notice that this function returns a rank-3 array (x), which has the realized values of the state process at the requested times. Notice too that x has rownames. When this function is called, in the course of any algorithm that uses it, some basic error checks will be performed.

Next, we write a function that computes the likelihoods of a set of process model state transitions. Again, pay special attention to the structure of the input arguments and return value.

```
ou2.dprocess <- function(x, times, params, log,
    ...) {
    nreps <- ncol(x)
    ntimes <- length(times)
    eps <- numeric(2)
    d <- array(0, dim = c(nreps, ntimes - 1))
    for (k in 2:ntimes) {
        for (j in 1:nreps) {
            eps[1] <- (x["x1", j, k] - params["alpha.1",
                j] * x["x1", j, k - 1] - params["alpha.3",
                j] * x["x2", j, k - 1])/params["sigma.1",
                j]
            eps[2] <- (x["x2", j, k] - params["alpha.2",
                j] * x["x1", j, k - 1] - params["alpha.4",
                j] * x["x2", j, k - 1] - params["sigma.2",
                j] * eps[1])/params["sigma.3",
                j]
            d[j, k - 1] <- sum(dnorm(eps, mean = 0,
                sd = 1, log = TRUE), na.rm = T) -
                sum(log(params[c("sigma.1", "sigma.3"),
                  j]))
        }
```

```
    }
    if (log)
        d
    else exp(d)
}
```

In this function, `times` and `params` are as before, and `x` is a rank-3 array. In practice, you can think of `x` as an array that might have been generated by a call to the `rprocess` function above.

Third, we write a measurement model simulator. In this function, `x`, `t`, and `params` are states, time, and parameters, but they have a different form from those above. In particular, `x` and `params` are vectors. Notice that we give the returned vector, `y`, names to match the names of the data.

```
bvnorm.rmeasure <- function(x, t, params, ...) {
    c(y1 = rnorm(n = 1, mean = x["x1"], sd = params["tau"]),
        y2 = rnorm(n = 1, mean = x["x2"], sd = params["tau"]))
}
```

Finally, we specify how to evaluate the likelihood of an observation given the underlying state. Again, the arguments `x`, `y`, and `params` are vectors.

```
bvnorm.dmeasure <- function(y, t, x, params, log,
    ...) {
    f <- sum(dnorm(x = y[c("y1", "y2")], mean = x[c("x1",
        "x2")], sd = params["tau"], log = TRUE),
        na.rm = TRUE)
    if (log)
        f
    else exp(f)
}
```

With these four functions in hand, we construct 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)
```

In the above, `times` are the times at which the observations (given by `data`) were observed. The scalar `t0` is the time at which the process model is initialized: `t0` should not be any later than the first observation time `times[1]`. In the present case, it was easy to specify all four of the basic functions. That won't always be the case and it's not necessary to specify all of them to construct a `pomp` object. If any statistical method using the `pomp` object wants access to a function that hasn't been provided, however, an error will be generated.

We'll now specify some "true" parameters and initial states in the form of a named numeric vector:

```
true.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)
```

Note that the initial states are specified by parameters that have names ending in '.0'. This is important, since this identifies them as initial-value parameters. By default, the unobserved (state) process will be initialized with these values, and the names of the state variables will be obtained by dropping the '.0'.

In applications, one will frequently want more flexibility in parameterizing the initial state. This is available: one can optionally specify an alternative initializer. See `?pomp` for details.

The pomp object `ou2` we just constructed has no data. If we simulate the model, we'll obtain another `pomp` object just like `ou2`, but with the `data` slot filled with simulated data:

```
ou2 <- simulate(ou2, params = true.p, nsim = 1000,
    seed = 800733088)
ou2 <- ou2[[1]]
```

Here, we actually ran 1000 simulations: the default behavior of `simulate` is to return a list of `pomp` objects.

There are a number of *methods* that perform operations on `pomp` objects. One can read the documentation on all of these by doing `class?pomp`. For example, one can coerce a `pomp` object to a data frame:

```
as(ou2, "data.frame")
```

and if we `print` a `pomp` object, the resulting data frame is what is shown. One can access the data and the observation times using

```
data.array(ou2)
time(ou2)
```

One can also plot a `pomp` object (Fig. 1).

1.2. The low-level interface.
There is a low-level interface to `pomp` objects, primarily designed for package developers. Ordinary users should have little reason to use this interface. Here, I'll just introduce each of the methods that make up this interface.

The `init.state` method is called to initialize the state (unobserved) process. It takes a vector or matrix of parameters and returns a matrix of initial states.

```
x0 <- init.state(ou2, params = true.p)
x0

   [,1]
x1   50
x2  -50
```

The `rprocess` method gives access to the process model simulator. It takes initial conditions (which need not correspond to the zero-time `t0` specified when the `pomp` object was constructed), a set of times, and a set of parameters. The initial states and parameters must be matrices, and they are checked for commensurability. The method returns a rank-3 array containing simulated state trajectories, sampled at the times specified.

```
x <- rprocess(ou2, xstart = as.matrix(x0), times = c(0,
    time(ou2)), params = as.matrix(true.p))
dim(x)

[1]   2   1 101
```

Note that the dimensions of x are `nvars x nreps x ntimes`, where `nvars` is the number of state variables, `nreps` is the number of simulated trajectories (which is the number of columns in the `params` and
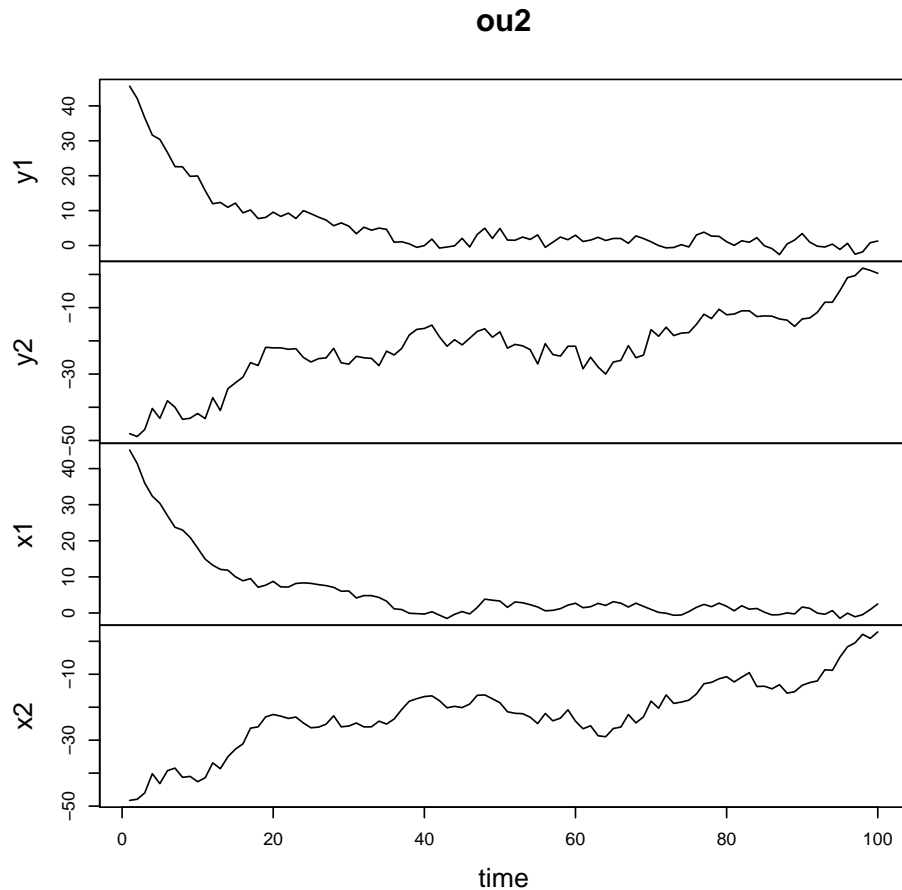
ou2

FIGURE 1. One can plot a pomp object. This shows the result of plot(ou2).

xstart matrices), and ntimes is the length of the times argument. Note also that x[,,1] is identical to xstart.

The rmeasure method gives access to the measurement model simulator:

```
y <- rmeasure(ou2, x = x[, , -1, drop = F], times = time(ou2),
    params = as.matrix(true.p))
dim(y)
```

```
[1]   2   1 100
```

The dmeasure and dprocess methods give access to the measurement and process model densities, respectively.

```
dprocess(ou2, x[, , 36:41, drop = F], times = time(ou2)[35:40],
    params = as.matrix(true.p))
```

```
           [,1]       [,2]       [,3]       [,4]       [,5]
[1,] 0.01655179 0.01907030 0.02469285 0.03349765 0.05539911
```

```
dmeasure(ou2, y = y[, 1, 1:4], x = x[, , 2:5,
    drop = F], times = time(ou2)[1:4], params = as.matrix(true.p))

          [,1]        [,2]        [,3]        [,4]
[1,] 0.02808023 0.03896308 0.05809258 0.09430426
```

All of these are to be preferred to direct access to the slots of the `pomp` object, because they do sanity checks on the inputs and outputs.

Since the codes above that implement the model are written in R, they're not particularly fast. To maximize computational efficiency, you'll frequently want to instead use compiled codes. For more information on this topic, see the vignette, *Using compiled code in pomp*. For the remainder of this vignette, we'll use a version of `ou2` in which each of the four functions — `dprocess`, `rprocess`, `dmeasure`, `rmeasure` — is written in C. This `pomp` object is provided in the package; just do

```
data(ou2)
```

to retrieve it. The C codes that make it up are included in the source distribution of the package (look in the file 'ou2.c'). Another example, an SIR model of disease transmission, is included in the package 'examples' directory.

2. PARTICLE FILTER.

We can run a particle filter as follows:

```
fit1 <- pfilter(ou2, params = true.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)
```

with much the same result, for example.

We can compare the results against those of the Kalman filter, which is exact in the case of a linear, Gaussian model such as the one implemented in `ou2`. 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(true.p[c("alpha.1", "alpha.2", "alpha.3",
    "alpha.4")], 2, 2)
b <- diag(1, 2)
sigma <- matrix(c(true.p["sigma.1"], true.p["sigma.2"],
    0, true.p["sigma.3"]), 2, 2)
tau <- diag(true.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$loglik=-412`, while the particle filter gives us `fit1$loglik=-412`.

3. The maximum likelihood by iterated filtering (MIF) algorithm

The MIF algorithm works by modifying the model slightly. It replaces the model we are interested in fitting — which has time-invariant parameters — with a model that is just the same except that its parameters take a random walk in time. As the intensity of this random walk approaches zero, the modified model approaches the fixed-parameter model. MIF works by iterating a particle filter on this model. The extra variability in the parameters combats the particle depletion that typically plagues simple particle filters.

At the beginning of each iteration, MIF must create an initial distribution of particles in the state-parameter space. For this purpose, MIF uses a function, `particles`, which can be optionally specified by the user. By default, MIF uses a multivariate normal particle distribution. The `particles` function takes an argument, `sd`, that scales the width of the distribution of particles in each of the directions of the state-parameter space. In particular, this distribution must be such that, when `sd=0`, all the particles are identical. In this vignette, we'll use the default (multivariate normal) particle distribution.

Let's jump right in and run MIF to maximize the likelihood over two of the parameters and both 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. Just to make it interesting, we'll start far from the true parameter values.

```
start.p <- true.p
start.p[c("x1.0", "x2.0", "alpha.1", "alpha.4")] <- c(45,
    -60, 0.8, 0.9)
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.4 = 0.1), Np = 1000,
    var.factor = 1, ic.lag = 10, cooling.factor = 0.95,
    max.fail = 100)
fit <- continue(fit, Nmif = 79, max.fail = 100)
fitted.pars <- c("alpha.1", "alpha.4", "x1.0",
    "x2.0")
cbind(start = start.p[fitted.pars], mle = signif(coef(fit,
    fitted.pars), 3), truth = true.p[fitted.pars])
```

```
        start     mle   truth
alpha.1   0.8   0.898    0.90
alpha.4   0.9   0.989    0.99
x1.0     45.0  51.100   50.00
x2.0    -60.0 -49.500  -50.00
```

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)=-410.9`. 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] -411.6
```

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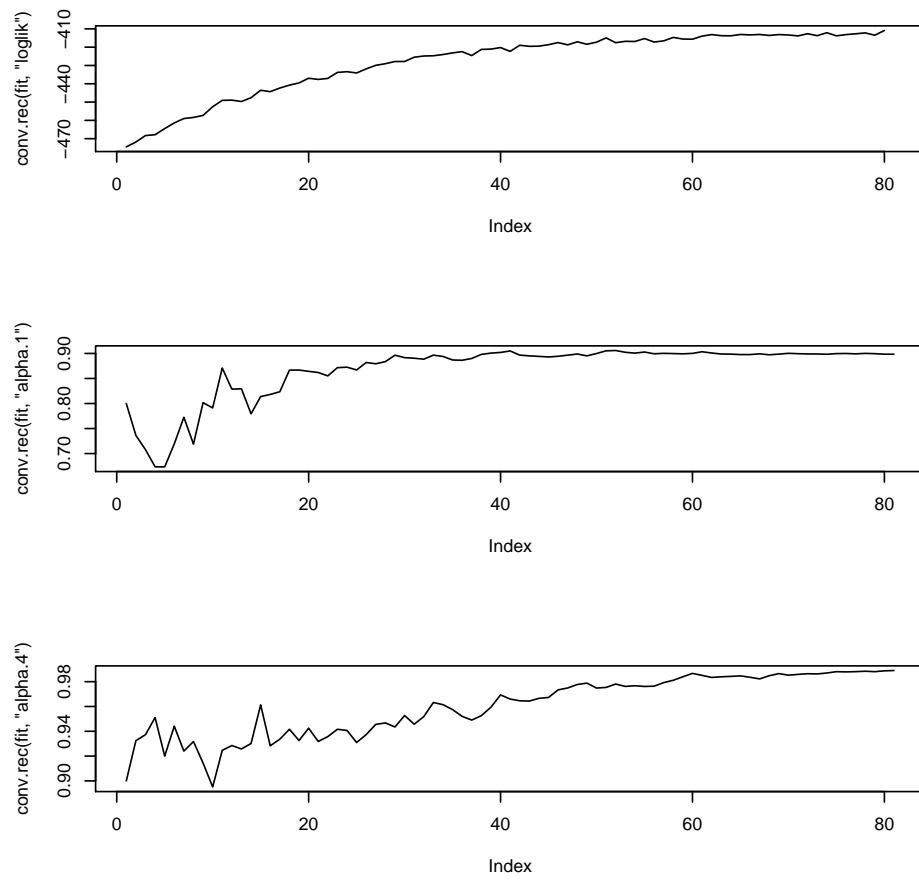
E-mail address: `kingaa at umich dot edu`

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

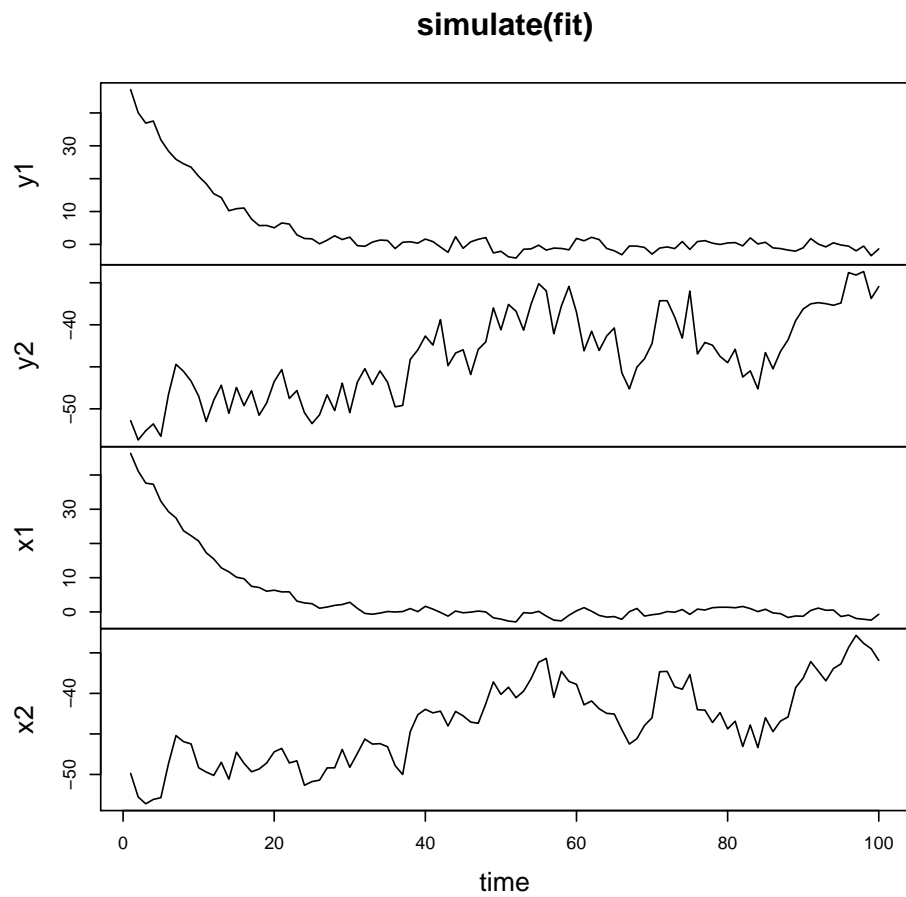FIGURE 2. Convergence plots can be used to help diagnose convergence of the MIF algorithm.

simulate(fit)

FIGURE 3. mif objects can be simulated.