INTRODUCTION TO POMP: INFERENCE FOR PARTIALLY-OBSERVED MARKOV PROCESSES

AARON A. KING, EDWARD L. IONIDES, CARLES BRETÓ, STEPHEN P. ELLNER, BRUCE E. KENDALL, MATTHEW FERRARI, MICHAEL L. LAVINE, AND DANIEL C. REUMAN

Contents

1. Partially-observed Markov processes

Partially-observed Markov process models are also known as state-space models or stochastic dynamical systems. The R package `pomp` provides facilities for fitting such models to uni- or multi-variate time series, for simulating them, for assessing model adequacy, and for comparing among models. The methods implemented in `pomp` are all "plug-and-play" in the sense that they require only that one be able to simulate the process portion of the model. This property is desirable because it will typically be the case that a mechanistic model will not be otherwise amenable to standard statistical analyses, but will be relatively easy to simulate. Even when one is interested in a model for which one can write down an explicit likelihood, for example, there are probably models that are "nearby" and equally interesting for which the likelihood cannot explicitly be written. The price one pays for this flexibility is primarily in terms of computational expense.

A partially-observed Markov process has two parts. First, there is the true underlying process which is generating the data. This is typically the thing we are most interested in: our goal is usually to better

Date: February 15, 2011, `pomp` version 0.36-2.

understand this process. Specifically, we may have various alternate hypotheses about how this system functions and we want to see whether time series data can tell us which hypotheses explain the data better. The challenge, of course, is that the data shed light on the system only indirectly.

`pomp` assumes that we can translate our hypotheses about the underlying, unobserved process into a Markov process model: That is, we are willing to assume that the system has a true *state* process, X_t that is Markovian. In particular, given any sequence of times t_0, t_1, ..., t_n, the Markov property allows us to write

$$X_{t_{k+1}} \sim f(X_{t_k}, \theta), \tag{1}$$

for each $k = 1, \ldots, n$, where f is some density. [In this document, we will be fairly cavalier about abusing notation, using the letter f to denote a probability distribution function generically, assuming that the reader will be able to unambiguously tell which probability distribution we're talking about from the arguments to f and the context.] That is, we assume that the state at time t_{k+1} depends only on the state at time t_k and on some parameters θ.

In addition to the state process X_t, there is some measurement or observation process Y_t which models the process by which the data themselves are generated and links the data therefore to the state process. In particular, we assume that

$$Y_t \sim f(X_t, \theta) \tag{2}$$

for all times t. That is, that the observations Y_t are random variables that depend only on the state *at that time* as well as on some parameters.

So, to specify a partially-observed Markov process model, one has to specify a process (unobserved or state) model and a measurement (observation) model. This seems straightforward enough, but from the computational point of view, there are actually two aspects to each model that may be important. On the one hand, one may need to *evaluate* the probability density of the state-transition $X_{t_k} \to X_{t_{k+1}}$, i.e., to compute $f(X_{t_{k+1}} \,|\, X_{t_k}, \theta)$. On the other hand, one may need to *simulate* this distribution, i.e., to draw random samples from the distribution of $X_{t_{k+1}} \,|\, X_{t_k}$. Depending on the model and on what one wants specifically to do, it may be technically easier or harder to do one of these or the other. Likewise, one may want to simulate, or evaluate the likelihood of, observations Y_t. At it's most basic level `pomp` is an infrastructure that allows you to encode your model by specifying some or all of these four basic components:

`rprocess:` a simulator of the process model,
`dprocess:` an evaluator of the process model probability density function,
`rmeasure:` a simulator of the measurement model, and
`dmeasure:` an evaluator of the measurement model probability density function.

Once you've encoded your model, `pomp` provides a number of algorithms you can use to work with it. In particular, within `pomp`, you can:

(1) simulate your model easily, using `simulate`,
(2) integrate your model's deterministic skeleton, using `trajectory`,
(3) estimate the likelihood for any given set of parameters using sequential Monte Carlo, implemented in `pfilter`,
(4) find maximum likelihood estimates for parameters using iterated filtering, implemented in `mif`,
(5) estimate parameters using a simulated quasi maximum likelihood approach called *nonlinear forecasting* and implemented in `nlf`,
(6) estimate parameters using trajectory matching, as implemented in `traj.match`,
(7) estimate parameters using probe matching, as implemented in `probe.match`,
(8) print and plot data, simulations, and diagnostics for the foregoing algorithms,
(9) build new algorithms for partially observed Markov processes upon the foundations `pomp` provides, using the package's applications programming interface (API).

In this document, we'll see how all this works using relatively simple examples.

2. A first example: a discrete-time bivariate autoregressive process.

For simplicity, we'll begin with a very simple discrete-time model. The plug-and-play methods in `pomp` were designed to work on much more complicated models, and for our first example, they'll be extreme overkill, but starting with a simple model will help make the implementation of more general models clear. Moreover, our first example will be one for which plug-and-play methods are not even necessary. This will allow us to compare the results from generalizable plug-and-play methods with exact results from specialized methods appropriate to this particular model. Later we'll look at a continuous-time model for which no such special tricks are available.

Consider a two-dimensional AR(1) process with noisy observations. The state process $X_t \in \mathbb{R}^2$ satisfies

$$X_t = \alpha\, X_{t-1} + \sigma\, \xi_t. \tag{3}$$

The measurement process is

$$Y_t = \beta\, X_t + \tau\, \varepsilon_t. \tag{4}$$

In these equations, α and and β are 2×2 constant matrices. ξ_t and ε_t are mutually-independent families of i.i.d. bivariate standard normal random variables. σ is a lower-triangular 2×2 matrix such that $\sigma\sigma^T$ is the variance-covariance matrix of $X_{t+1}|X_t$. We'll assume that each component of X is measured independently and with the same error, τ, so that the variance-covariance matrix of $Y_t|X_t$ is just τ^2 times the identity matrix.

Given a data set, one can obtain exact maximum likelihood estimates of this model's parameters using the Kalman filter. We will demonstrate this below. Here, however, for pedagogical reasons, we'll approach this model as we would a more complex model for which no such exact estimation is available.

3. Defining a partially observed Markov process in `pomp`.

In order to fully 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). As we saw before, we would like to be able to:

1. simulate from the process model, i.e., make a random draw from $X_{t+1} \,|\, X_t = x$ for arbitrary x and t (`rprocess`),
2. compute the probability density function (pdf) of state transitions, i.e., compute $f(X_{t+1} = x' \,|\, X_t = x)$ for arbitrary x, x', and t (`dprocess`),
3. simulate from the measurement model, i.e., make a random draw from $Y_t \,|\, X_t = x$ for arbitrary x and t (`rmeasure`),
4. compute the measurement model pdf, i.e., $f(Y_t = y \,|\, X_t = x)$ for arbitrary x, y, and t (`dmeasure`), and
5. compute the *deterministic skeleton*. In discrete-time, this is the map $x \mapsto \mathbb{E}[X_{t+1} \,|\, X_t = x]$ for arbitrary x.

For this simple model, all this is easy enough. More generally, it will be difficult to do some of these things. Depending on what we wish to accomplish, however, we may not need all of these capabilities and in particular, **to use any particular one of the algorithms in `pomp`, we need never specify *all* of 1–5.** For example, to simulate data, all we need is 1 and 3. To run a particle filter (and hence to use iterated filtering, `mif`), one needs 1 and 4. To do MCMC, one needs 2 and 4. Nonlinear forecasting (`nlf`) and probe matching (`probe.match`) require 1 and 3. Trajectory matching (`traj.match`) requires 4 and 5.

Using `pomp`, the first step is always to construct an R object that encodes the model and the data. Naturally enough, this object will be of class `pomp`. The key step in this is to specify functions to do some or all of 1–5, along with data and (optionally) other information. The package provides a number algorithms for fitting the models to the data, for simulating the models, studying deterministic skeletons,

and so on. The documentation (`?pomp`) spells out the usage of the `pomp` constructor, including detailed specifications for all its arguments and a worked example.

Let's see how to implement the AR(1) model in `pomp`. Here, we'll take the shortest path to this goal. In the "advanced topics in `pomp`" vignette, we show how one can make the codes much more efficient using compiled native (C or FORTRAN) code.

First, we write a function that implements the process model simulator. This is a function that will simulate a single step ($t \to t+1$) of the unobserved process (3).

```
require(pomp)
ou2.proc.sim <- function (x, t, params, delta.t, ...) {
  xi <- rnorm(n=2,mean=0,sd=sqrt(delta.t)) # noise terms
  xnew <- c(
            params["alpha.1"]*x["x1"]+params["alpha.3"]*x["x2"]+
              params["sigma.1"]*xi[1],
            params["alpha.2"]*x["x1"]+params["alpha.4"]*x["x2"]+
              params["sigma.2"]*xi[1]+params["sigma.3"]*xi[2]
            )
  names(xnew) <- c("x1","x2")
  xnew
}
```

The translation from the mathematical description (3) to the simulator is straightforward. When this function is called, the argument `x` contains the state at time `t`. The parameters (including the matrix α and the process noise s.d. matrix σ) are passed in the argument `params`. Notice that these arguments are named numeric vectors and that the output must be named numeric vector. In fact, the names of the output vector (here `xnew`) must be the same as those of the input vector `x`. The algorithms in `pomp` all make heavy use of the `names` attributes of vectors and matrices. The argument `delta.t` tells how big the time-step is. In this case, our time-step will be 1 unit; we'll see below how that gets specified.

Next, we'll implement a simulator for the observation process (4).

```
ou2.meas.sim <- function (x, t, params, ...) {
  y <- rnorm(n=2,mean=x[c("x1","x2")],sd=params["tau"])
  names(y) <- c("y1","y2")
  y
}
```

Again the translation is straightforward. When this function is called, the unobserved states at time `t` will be in the named numeric vector `x` and the parameters in `params` as before. The function returns a named numeric vector that represents a single draw from the observation process (4).

4. Simulating the model

With the two functions above, we already have all we need to simulate the entire model. The first step is to construct an R object of class `pomp` which will serve as a container to hold the model and data. This is done with a call to `pomp`:

```
ou2 <- pomp(
            data=data.frame(
              time=1:100,
              y1=NA,
              y2=NA
              ),
            times="time",
            rprocess=discrete.time.sim(
              step.fun=ou2.proc.sim
              ),
            rmeasure=ou2.meas.sim,
            t0=0
            )
```

The first argument (`data`) specifies a data-frame that holds the data and the times at which the data were observed. Since this is a toy problem, we have no data. In a moment, however, we'll simulate some data so we can explore `pomp`'s various fitting methods. The second argument (`times`) specifies which of the columns of `data` is the time variable. The third argument (`rprocess`) specifies that the process model simulator will be in discrete-time, one step at a time. The function `discrete.time.sim` belongs to the `pomp` package. It takes the argument `step.fun`, which specifies the particular function that actually takes the step. The step is assumed to be a unit interval of time. The argument `rmeasure` specifies the measurement model simulator function. `t0` fixes t_0 for this model; here we have chosen this to be one time unit before the first observation.

Before we can simulate the model, we need to settle on some parameter values. We do this by specifying a named numeric vector that contains at least all the parameters needed by the functions `ou.proc.sim` and `ou.meas.sim`. The parameter vector needs to specify the initial conditions $X(t_0) = x_0$ as well.

```
theta <- c(
           alpha.1=0.8, alpha.2=-0.5, alpha.3=0.3, alpha.4=0.9,
           sigma.1=3, sigma.2=-0.5, sigma.3=2,
           tau=1,
           x1.0=-3, x2.0=4
           )
```

In terms of the mathematical formulation of the model (Eqs. 3–4), we have

$$\alpha = \begin{pmatrix} \alpha_1 & \alpha_3 \\ \alpha_2 & \alpha_4 \end{pmatrix} \qquad \sigma = \begin{pmatrix} \sigma_1 & 0 \\ \sigma_2 & \sigma_3 \end{pmatrix} \qquad \beta = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \qquad X(0) = \begin{pmatrix} -3 \\ 4 \end{pmatrix}.$$

The initial conditions are specified by the `x1.0` and `x2.0` elements. Here, the fact that the names end in "`.0`" is significant. This is the default operation of `pomp`: it is possible to parameterize the initial conditions in an arbitrary way using the optional `initializer` argument to `pomp`: see the documentation (`?pomp`) for details.

Now we can simulate the model:

```
ou2 <- simulate(ou2,params=theta)
```

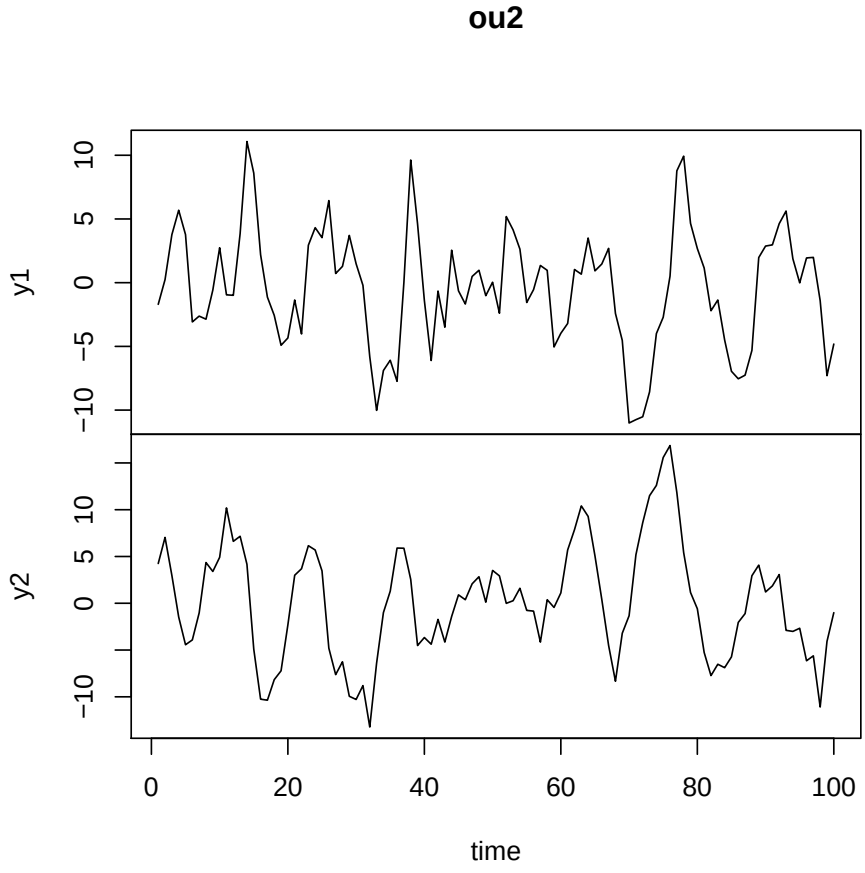

FIGURE 1. Simulated data and unobserved states from the bivariate AR(1) process (Eqs. 3–4). This figure shows the output of the command `plot(ou2,variables=c("y1","y2"))`.

Now `ou2` is identical to what it was before, but the data that were there before have been replaced by simulated data. The parameters (`theta`) at which the simulations were performed have also been saved internally to `ou2`. We can plot the simulated data via

```
plot(ou2,variables=c("y1","y2"))
```

Fig. 1 shows the results of this operation.

5. Computing likelihood using particle filtering

Since some parameter estimation algorithms in the `pomp` package only require simulations of the full process, we are already in a position to use them. At present, the only such algorithm is nonlinear forecasting (`nlf`). In the near future, probe-matching algorithms that require only `rprocess` and `rmeasure` will be included in the package. If we want to work with likelihood-based methods, however, we will need to be able to compute the likelihood of the data Y_t given the states X_t. To implement this in `pomp`, we write another function:

```
ou2.meas.dens <- function (y, x, t, 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)
}
```

This function computes the likelihood of the observation `y` at time `t` given the states `x` and the parameters `params`. Here, the named vector `y` contains data at a single timepoint (`t`) and `x` contains a single realization of the unobserved process at the same time. The extra argument `log` specifies whether likelihood or log-likelihood is desired.

To incorporate this into the `pomp` object, we'll make another call to `pomp`, giving our new function to the `dmeasure` argument:

```
dat <- as(ou2,"data.frame")
theta <- coef(ou2)
ou2 <- pomp(
            data=dat[c("time","y1","y2")],
            times="time",
            rprocess=discrete.time.sim(ou2.proc.sim),
            rmeasure=ou2.meas.sim,
            dmeasure=ou2.meas.dens,
            t0=0
            )
coef(ou2) <- theta
```

Note that we've first extracted the data from our old `ou2` and set up the new one with the same data and parameters. The calls to `coef` and `coef<-` in the lines above make sure the parameters have the same values they had before.

To compute the likelihood of the data, we can use the function `pfilter`. This runs a plain vanilla particle filter (AKA sequential Monte Carlo) algorithm and results in an unbiased estimate of the likelihood. See Arulampalam *et al.* (2002) for an excellent tutorial on particle filtering. To do this, we must decide how many concurrrent realizations (*particles*) to use: the larger the number of particles, the smaller the Monte Carlo error but the greater the computational effort. Let's run `pfilter` with 1000 particles to estimate the likelihood at the true parameters:

```
pf <- pfilter(ou2,params=theta,Np=1000)
loglik.truth <- logLik(pf)
loglik.truth
```

```
[1] -479.0558
```

Since the true parameters (i.e., the parameters that generated the data) are stored within the `pomp` object `ou2` and can be extracted by the `coef` function, we could have done

```
pf <- pfilter(ou2,params=coef(ou2),Np=1000)
```

Box 1 Implementation of the Kalman filter for the AR(1) process.

```
require(mvtnorm)
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.loglik <- numeric(ntimes)
  filter.mean <- matrix(0,n,ntimes)
  pred.mean <- matrix(0,n,ntimes)
  pred.var <- array(0,dim=c(n,n,ntimes))
  m <- x0
  v <- diag(0,n)
  for (k in seq_len(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.loglik[k] <- dmvnorm(x=y[,k],mean=as.numeric(b%*%M),sigma=q,log=TRUE)
    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=cond.loglik,
       loglik=sum(cond.loglik)
       )
}
```

or even just

```
pf <- pfilter(ou2,Np=1000)
```

since the parameters are stored in the `pomp` object `ou2`. Now let's compute the log likelihood at a different point in parameter space:

```
theta.true <- coef(ou2)
theta.guess <- theta.true
theta.guess[c("alpha.2","alpha.3","tau")] <- 1.5*theta.true[c("alpha.2","alpha.3","tau")]
pf <- pfilter(ou2,params=theta.guess,Np=1000)
loglik.guess <- logLik(pf)
```

As we mentioned before, for this particular example, we can compute the likelihood exactly using the Kalman filter, using this as a check on the validity of the particle filtering algorithm. An implementation of the Kalman filter is given in Box 1. Let's run the Kalman filter on the example data we generated above:

```
y <- obs(ou2)
a <- matrix(theta.guess[c('alpha.1','alpha.2','alpha.3','alpha.4')],nrow=2,ncol=2)
```

```
b <- diag(1,2)   ## b is the identity matrix
sigma <- matrix(
                c(
                  theta.guess['sigma.1'],0,
                  theta.guess['sigma.2'],theta.guess['sigma.3']
                  ),
                nrow=2,ncol=2,byrow=T
                )
tau <- diag(theta.guess['tau'],2,2)
x0 <- init.state(ou2)
kf <- kalman.filter(y,x0,a,b,sigma,tau)
```

In this case, the Kalman filter gives us a log likelihood of -497.25, while the particle filter with 1000 particles gives -499.24. Since the particle filter gives an unbiased estimate of the likelihood, the difference is due to Monte Carlo error in the particle filter. One can reduce this error by using a larger number of particles and/or by re-running `pfilter` multiple times and averaging the resulting estimated likelihoods. The latter approach has the advantage of allowing one to estimate the Monte Carlo error itself.

6. Interlude: utility functions for extracting and changing pieces of a pomp object

The `pomp` package provides a number of functions to extract or change pieces of a `pomp`-class object. One can read the documentation on all of these by doing `class?pomp` and `methods?pomp`. For example, as we've already seen, 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, together with the call that created the `pomp` object. One has access to the data and the observation times using

```
obs(ou2)
obs(ou2,"y1")
time(ou2)
```

The observation times can be changed using

```
time(ou2) <- 1:10
```

One can respectively view and change the zero-time by

```
timezero(ou2)
timezero(ou2) <- -10
```

and can respectively view and change the zero-time together with the observation times by doing, for example

```
time(ou2,t0=TRUE)
time(ou2,t0=T) <- seq(from=0,to=10,by=1)
```

Alternatively, one can construct a new pomp object with the same model but with data restricted to a specified window:

```
window(ou2,start=3,end=20)
```

Note that `window` does not change the zero-time. One can display and modify model parameters using, e.g.,

```
coef(ou2)
coef(ou2,c("sigma.1","sigma.2")) <- c(1,0)
```

Finally, one has access to the unobserved states via, e.g.,

```
states(ou2)
states(ou2,"x1")
```

7. Estimating parameters using iterated filtering: mif

Iterated filtering is a technique for maximizing the likelihood obtained by filtering. In `pomp`, the particle filter is used as a basis for iterated filtering. Iterated filtering is implemented in the `mif` function.

The key idea of iterated filtering is to replace 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 original model. Adding additional variability in this way has three positive effects: (i) it smooths the likelihood surface, which makes optimization easier, (ii) it combats *particle depletion*, the fundamental difficulty associated with the particle filter, and (iii) the additional variability can be exploited to estimate of the gradient of the (smoothed) likelihood surface *with no more computation than is required to estimate of the value of the likelihood*. Iterated filtering exploits these effects to optimize the likelihood in a computationally efficient manner. As the filtering is iterated, the additional variability is decreased according to a *cooling schedule*. The cooling schedule can be adjusted in `mif`, as can the intensity of the parameter-space random walk and the other algorithm parameters. See the documentation (`?mif`) for details.

Let's use iterated filtering to obtain an approximate MLE for the data in `ou2`. We'll initiate the algorithm at `theta.guess` and just estimate the parameters α_1, α_4, and τ along with the initial conditions:

```
mf <- replicate(
                n=3,
                mif(
                    ou2,
                    Nmif=120,
                    start=theta.guess,
                    pars=c('alpha.2','alpha.3','tau'),
                    ivps=c('x1.0','x2.0'),
                    rw.sd=c(
                      x1.0=5,x2.0=5,
                      alpha.2=0.02,alpha.3=0.02,tau=0.05
                      ),
                    Np=1000,
                    var.factor=4,
                    ic.lag=10,
                    cooling.factor=0.97,
                    max.fail=10
                    )
                )
fitted.pars <- c("alpha.2","alpha.3","tau","x1.0","x2.0")
pf <- lapply(mf,pfilter)
loglik.mle <- log(mean(exp(480+sapply(pf,logLik))))-480
loglik.mle.sd <- sd(sapply(pf,logLik))
theta.mle <- apply(sapply(mf,coef),1,mean)
```

```
          guess      mle  truth
alpha.2   -0.75   -0.555   -0.5
alpha.3    0.45    0.259    0.3
tau        1.50    0.958    1.0
x1.0      -3.00   -3.030   -3.0
x2.0       4.00    3.040    4.0
loglik  -499.20 -477.900 -479.1
```

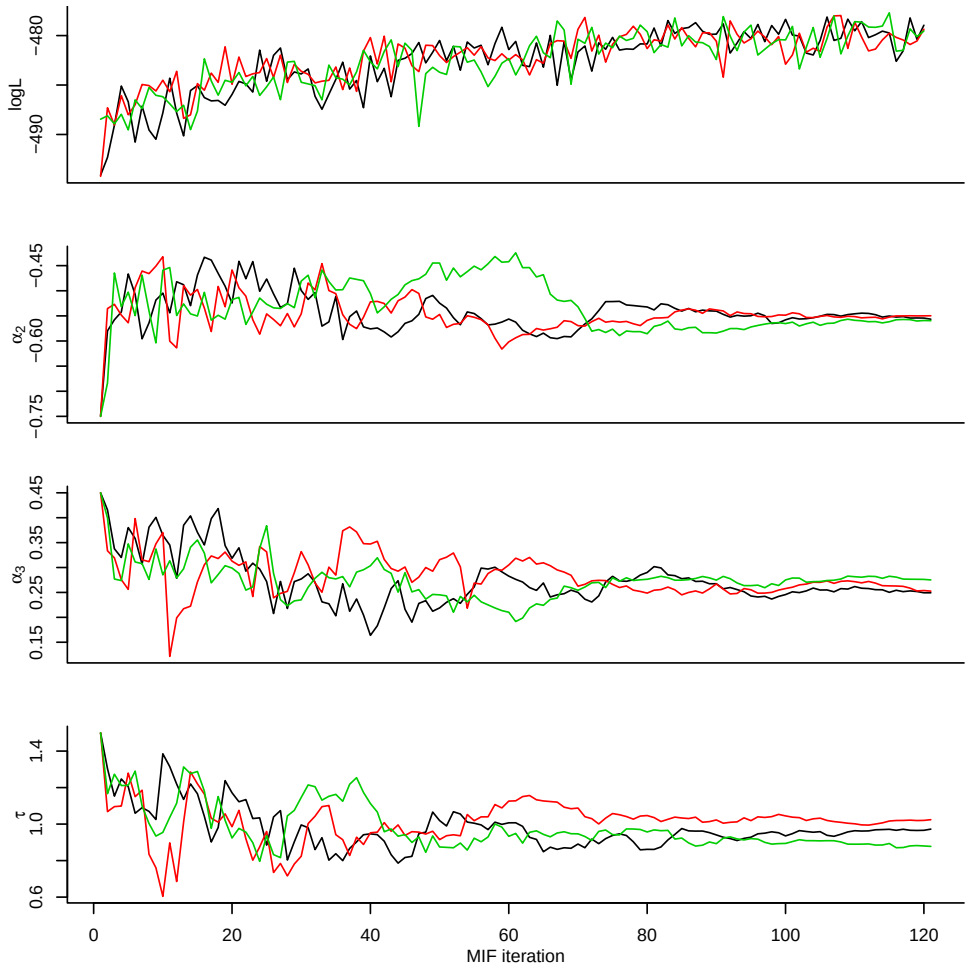

FIGURE 2. Convergence plots can be used to help diagnose convergence of the iterated filtering algorithm. This shows part of the output of `compare.mif(mf)`.

8. Trajectory matching: traj.match

The idea behind trajectory matching is a simple one. One attempts to fit a deterministic dynamical trajectory to the data. This is tantamount to assuming that all the stochasticity in the system is in the measurement process. In pomp, the trajectory is computed using the trajectory function, which in turn uses the skeleton slot of the pomp object. The skeleton slot should be filled with the deterministic skeleton of the process model. In the discrete-time case, this is the map

$$x \mapsto \mathbb{E}\left[X_{t+1} \mid X_t = x, \theta\right].$$

In the continuous-time case, this is the vectorfield

$$x \mapsto \lim_{\Delta t \to 0} \mathbb{E}\left[\frac{X_{t+\Delta t} - x}{\Delta t} \;\middle|\; X_t = x, \theta\right].$$

Our discrete-time bivariate autoregressive process has the deterministic skeleton

$$x \mapsto \alpha\, x, \tag{5}$$

which can be implemented in the R function

```
ou2.skel <- function (x, t, params, ...) {
  xnew <- c(
            params["alpha.1"]*x["x1"]+params["alpha.3"]*x["x2"],
            params["alpha.2"]*x["x1"]+params["alpha.4"]*x["x2"]
            )
  names(xnew) <- c("x1","x2")
  xnew
}
```

We can incorporate the deterministic skeleton into a new pomp object via the skeleton.map argument:

```
dat <- subset(as(ou2,"data.frame"),time<=60)
theta <- coef(ou2)
new.ou2 <- pomp(
                data=dat[c("time","y1","y2")],
                times="time",
                rprocess=discrete.time.sim(ou2.proc.sim),
                rmeasure=ou2.meas.sim,
                dmeasure=ou2.meas.dens,
                skeleton.map=ou2.skel,
                t0=0
                )
coef(new.ou2) <- theta
coef(new.ou2,"tau") <- 0.5
coef(new.ou2,c("sigma.1","sigma.2","sigma.3")) <- 0
new.ou2 <- simulate(new.ou2,seed=88737400L)
```

We use the skeleton.map argument for discrete-time processes and skeleton.vectorfield for continuous-time processes. Note that we have turned off the process noise in new.ou2 (next to last line) so that trajectory matching is actually formally appropriate for this model.

The pomp function traj.match calls the optimizer optim to minimize the discrepancy between the trajectory and the data. The discrepancy is measured using the dmeasure function from the pomp object. Fig. 3 shows the results of this fit.

```
tm <- traj.match(
                 new.ou2,
```

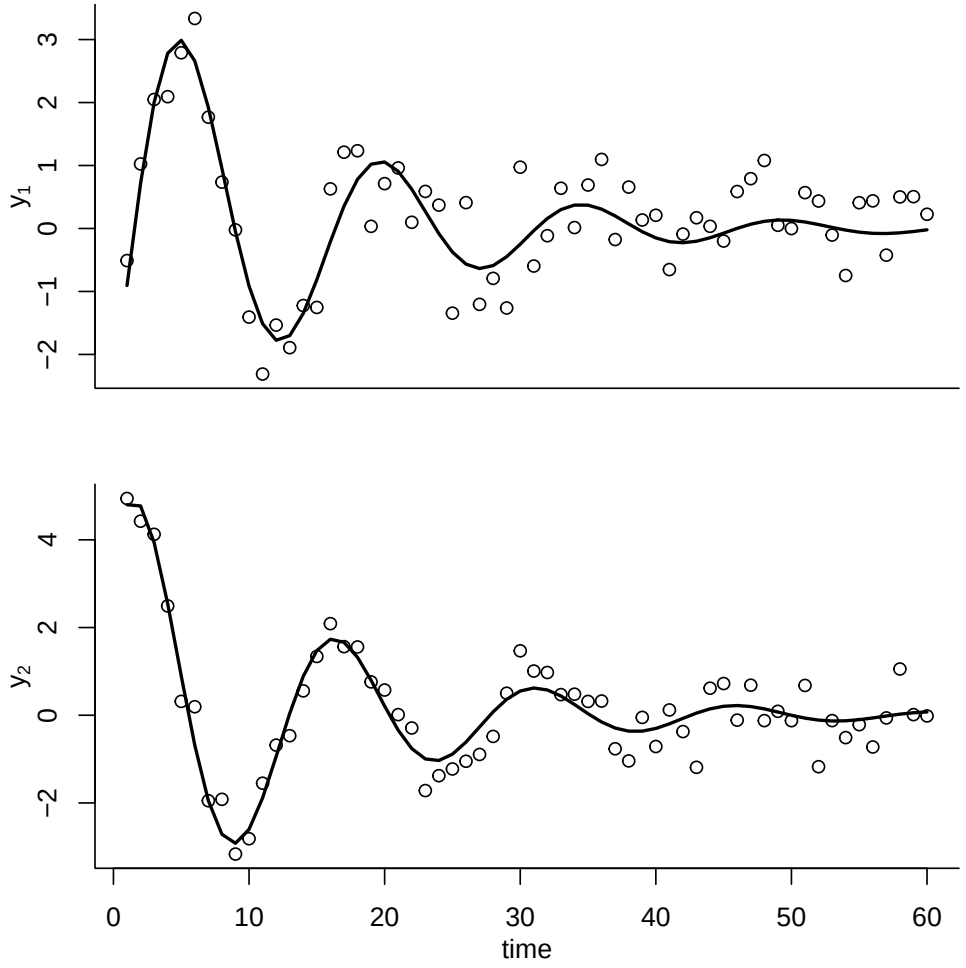

FIGURE 3. Illustration of trajectory matching. The points show data simulated from `new.ou2`, which has no process noise but only measurement error. The solid line shows the trajectory of the best-fitting model, obtained using `traj.match`.

```
start=coef(new.ou2),
est=c("alpha.2","alpha.3","tau","x1.0","x2.0"),
method="Nelder-Mead",
maxit=1000,
reltol=1e-8
)
```

9. Probe matching: `probe.match`

In probe matching, we fit a model to data using a set of summary statistics. We evaluate these statistics on the data and compare them to the distribution of values they take on simulations, then adjust model parameters to maximize agreement between model and data according to some criterion. Following Kendall *et al.* (1999), we refer to the summary statistics as *probes*. In probe-matching, one has unrestricted choice of probes, and there are a great many probes that one might sensibly choose. This introduces a degree of subjectivity into the inference procedure but has the advantage of allowing the investigator to identify *a priori* those features of a data set he or she believes to be informative.

In this section, we'll illustrate probe matching using a stochastic version of the Ricker map. In this discrete-time model, N_t represents the (true) size of a population at time t and obeys

$$N_{t+1} = r\,N_t\,\exp(-N_t + e_t), \qquad e_t \sim \text{normal}(0,\sigma).$$

In addition, we assume that measurements y_t of N_t are themselves noisy, according to

$$y_t \sim \text{Poisson}(\phi\,N_t).$$

As before, we'll begin by writing an R function that implements a simulator (`rprocess`) for the Ricker model. It will be convenient to work with log-transformed parameters $\log r$, $\log\sigma$, $\log\phi$. Thus

```
ricker.sim <- function (x, t, params, ...) {
  e <- rnorm(n=1,mean=0,sd=exp(params["log.sigma"]))
  xnew <- c(
            exp(params["log.r"]+log(x["N"])-x["N"]+e),
            e
            )
  names(xnew) <- c("N","e")
  xnew
}
```

Note that, in this implementation, e is taken to be a state variable. This is not strictly necessary, but it might prove useful, for example, in *a posteriori* diagnostic checking of model residuals. Now we can construct a `pomp` object; in this case, we use the `discrete.time.sim` plug-in. Note how we specify the measurement model.

```
ricker <- pomp(
               data=data.frame(time=seq(0,50,by=1),y=NA),
               times="time",
               t0=0,
               rprocess=discrete.time.sim(
                 step.fun=ricker.sim
                 ),
               measurement.model=y~pois(lambda=N*exp(log.phi))
               )
coef(ricker) <- c(
                  log.r=3.8,
                  log.sigma=log(0.3),
                  log.phi=log(10),
                  N.0=7,
                  e.0=0
                  )
ricker <- simulate(ricker,seed=73691676L)
```

A pre-built `pomp` object implementing this model is included with the package. Its `rprocess`, `rmeasure`, and `dmeasure` components are written in C and are thus a bit faster than the R implementation above. Do

```
data(ricker)
```

to load this `pomp` object.

In `pomp`, probes are simply functions that can be applied to an array of real or simulated data to yield a scalar or vector quantity. Several functions that create commonly-useful probes are included with the package. Do `?basic.probes` to read the documentation for these probes. In this illustration, we will make use of several probes recommended by Wood (2010): `probe.marginal`, `probe.acf`, and `probe.nlar`. `probe.marginal` regresses the data against a sample from a reference distribution; the probe's values are those of the regression coefficients. `probe.acf` computes the auto-correlation or auto-covariance of the data at specified lags. `probe.nlar` fits a simple nonlinear (polynomial) autoregressive model to the data; again, the coefficients of the fitted model are the probe's values. We construct our set of probes by specifying a list

```
plist <- list(
              probe.marginal("y",ref=obs(ricker),transform=sqrt),
              probe.acf("y",lags=c(0,1,2,3,4),transform=sqrt),
              probe.nlar("y",lags=c(1,1,1,2),powers=c(1,2,3,1),transform=sqrt)
              )
```

An examination of the structure of `plist` reveals that it is a list of functions of a single argument. Each of these functions can be applied to the `ricker`'s data or to simulated data sets. A call to `pomp`'s function `probe` results in the application of these functions to the data, their application to each of some large number, `nsim`, of simulated data sets, and finally to a comparison of the two. To see this, we'll apply probe to the Ricker model at the true parameters and at a wild guess.

```
pb.truth <- probe(ricker,probes=plist,nsim=1000,seed=1066L)
guess <- c(log.r=log(20),log.sigma=log(1),log.phi=log(20),N.0=7,e.0=0)
pb.guess <- probe(ricker,params=guess,probes=plist,nsim=1000,seed=1066L)
```

Results summaries and diagnostic plots showing the model-data agreement and correlations among the probes can be obtained by

```
summary(pb.truth)
summary(pb.guess)
plot(pb.truth)
plot(pb.guess)
```

An example of a diagnostic plot (using a simplified set of probes) is shown in Fig. 4. Among the quantities returned by `summary` is the synthetic likelihood (Wood, 2010). It is this synthetic likelihood that `pomp` attempts to maximize in probe matching.

Let us now attempt to fit the Ricker model to the data using probe-matching.

```
pm <- probe.match(
                  pb.guess,
                  est=c("log.r","log.sigma","log.phi"),
                  method="Nelder-Mead",
                  maxit=2000,
                  seed=1066L,
                  reltol=1e-8,
```

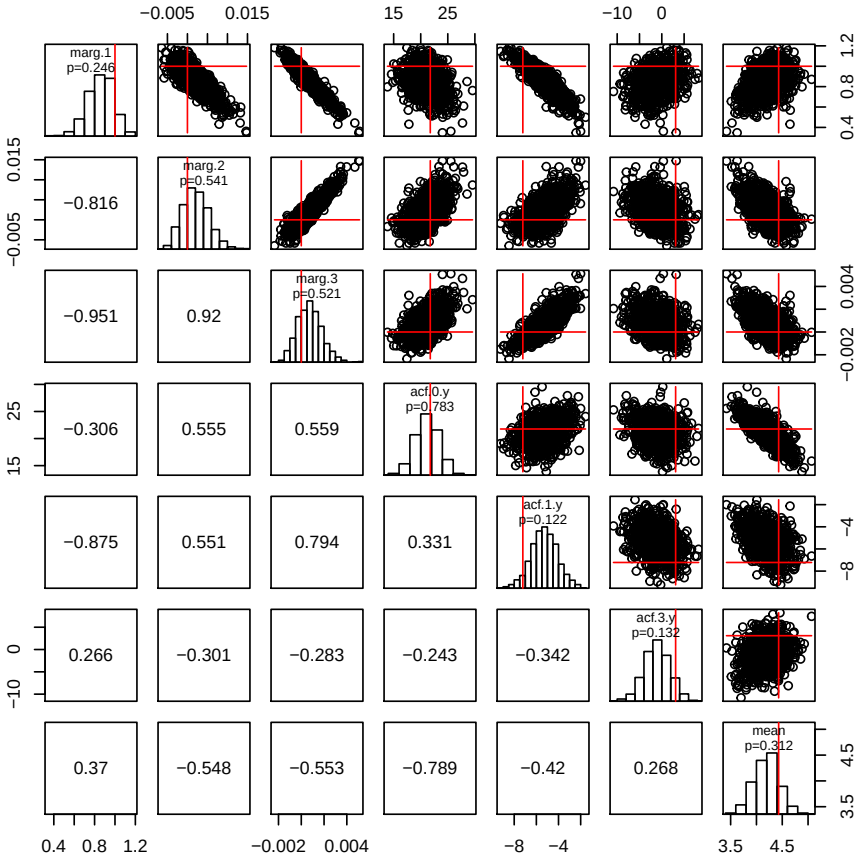

FIGURE 4. Results of `plot` on a `probed.pomp`-class object. Above the diagonal, the pairwise scatterplots show the values of the probes on each of 1000 data sets. The red lines show the values of each of the probes on the data. The panels along the diagonal show the distributions of the probes on the simulated data, together with their values on the data and a two-sided p-value. The numbers below the diagonal indicate the Pearson correlations between the corresponding probes.

```
                trace=3
                )
summary(pm)
```

This code runs a Nelder-Mead optimizer from the starting parameters `guess` in an attempt to maximize the synthetic likelihood based on the probes in `plist`. Both the starting parameters and the probes are stored internally in `pb.guess`, which is why we don't specify them explicitly here; if we wanted to change these, we could do so by specifying the `params` and/or `probes` arguments to `probe.match`. See `?probe.match` for full documentation.

By way of putting the synthetic likelihood in context, let's compare the results of estimating the Ricker model parameters using probe-matching and using iterated filtering, which is based on likelihood. The following code runs 600 MIF iterations starting at `guess`:

```
mf <- mif(
          ricker,
          start=guess,
          Nmif=100,
          Np=1000,
          cooling.factor=0.99,
          var.factor=2,
          ic.lag=3,
          max.fail=50,
          rw.sd=c(log.r=0.1,log.sigma=0.1,log.phi=0.1)
          )
mf <- continue(mf,Nmif=500,max.fail=20)
```

The following code compares parameters, likelihoods, and synthetic likelihoods (based on the probes in `plist`) at each of (1) the wild guess, (2) the truth, (3) the MLE from `mif`, and (4) the maximum synthetic likelihood estimate from `probe.match`.

```
pf.truth <- pfilter(ricker,Np=1000,max.fail=50,seed=1066L)
pf.guess <- pfilter(ricker,params=guess,Np=1000,max.fail=50,seed=1066L)
pf.mf <- pfilter(mf,Np=1000,seed=1066L)
pf.pm <- pfilter(pm,Np=1000,max.fail=10,seed=1066L)
pb.mf <- probe(mf,nsim=1000,probes=plist,seed=1066L)
res <- rbind(
             cbind(guess=guess,truth=coef(ricker),MLE=coef(mf),PM=coef(pm)),
             loglik=c(
               pf.guess$loglik,
               pf.truth$loglik,
               pf.mf$loglik,
               pf.pm$loglik
               ),
             synth.loglik=c(
               summary(pb.guess)$synth.loglik,
               summary(pb.truth)$synth.loglik,
               summary(pb.mf)$synth.loglik,
               summary(pm)$synth.loglik
               )
             )
print(res,digits=3)
```

```
               guess   truth     MLE      PM
log.r           3.00    3.80    3.78    3.74
log.sigma       0.00   -1.20   -1.74   -1.12
log.phi         3.00    2.30    2.33    2.41
N.0             7.00    7.00    7.00    7.00
e.0             0.00    0.00    0.00    0.00
loglik       -230.86 -139.55 -136.83 -143.33
synth.loglik  -12.28   17.47   17.36   20.34
```

10. Nonlinear forecasting: `nlf`

To be added.

11. A more complex example: a seasonal epidemic model

The SIR model is a mainstay of theoretical epidemiology. It has the deterministic skeleton

$$\frac{dS}{dt} = \mu\,(N - S) - \beta(t)\,\frac{I}{N}\,S$$
$$\frac{dI}{dt} = \beta(t)\,\frac{I}{N}\,S - \gamma\,I - \mu\,I$$
$$\frac{dR}{dt} = \gamma\,I - \mu\,R$$

Here $N = S + I + R$ is the (constant) population size and β is a time-dependent contact rate. We'll assume that the contact rate is periodic and implement it as a covariate. As an additional wrinkle, we'll assume that the rate of the infection process $\beta\,I/N$ is perturbed by white noise.

As in the earlier example, we need to write a function that will simulate the process. We can use `gillespie.sim` to implement this using the exact stochastic simulation algorithm of Gillespie (1977). This will be quite slow and inefficient, however, so we'll use the so-called "tau-leap" algorithm, one version of which is implemented in `pomp` using Euler-multinomial processes. Before we do this, we'll first define the basis functions that will be used for the seasonality. It is convenient to use periodic B-splines for this purpose. The following codes set up this basis.

```
tbasis <- seq(0,25,by=1/52)
basis <- periodic.bspline.basis(tbasis,nbasis=3)
colnames(basis) <- paste("seas",1:3,sep='')
```

Now we'll define the process model simulator. Since we have covariates now, our function will have one additional argument, `covars`, which will contain the value of each covariate at time `t`, established using linear interpolation if necessary.

```
sir.proc.sim <- function (x, t, params, covars, delta.t, ...) {
  params <- exp(params)
  with(
       as.list(c(x,params,covars)),
       {
         beta <- exp(sum(log(c(beta1,beta2,beta3))*c(seas1,seas2,seas3)))
         beta.var <- beta.sd^2
         dW <- if (beta.var>0)
           rgamma(n=1,shape=delta.t/beta.var,scale=beta.var)
         else
           delta.t
         foi <- (iota+beta*I*dW/delta.t)/pop
         trans <- c(
                    rpois(n=1,lambda=mu*pop*delta.t),
                    reulermultinom(n=1,size=S,rate=c(foi,mu),dt=delta.t),
                    reulermultinom(n=1,size=I,rate=c(gamma,mu),dt=delta.t),
                    reulermultinom(n=1,size=R,rate=c(mu),dt=delta.t)
                    )
         c(
           S=S+trans[1]-trans[2]-trans[3],
           I=I+trans[2]-trans[4]-trans[5],
           R=R+trans[4]-trans[6],
           cases=cases+trans[4],
           W=if (beta.sd>0) W+(dW-delta.t)/beta.sd else W
           )
```

```
    }
    )
}
```

Let's look at this definition in a bit of detail. We will be log-transforming the parameters: the first line untransforms them. Here, we use `with` to make the codes a bit easier to read. The variable `beta` will be the transmission rate: the time-dependence of this rate is parameterized using the basis functions, the current values have been passed via the `covars` argument. The next lines make a draw, `dW`, from a Gamma random variable which will model environmental stochasticity as white noise in the transmission process (Bretó *et al.*, 2009; He *et al.*, 2010). The next line computes the force of infection, `foi`. `trans` is next filled with random draws of all the transitions between the S, I, and R compartments. Births are modeled using a Poisson distribution, the number of births is stored in `trans[1]`. Individuals leave the S class through either death or infection: `trans[2]` will contain the number infected, `trans[3]` the number dead. The number leaving the I and R classes are handled similarly. See the documentation on `reulermultinom` for a more thorough explanation of how this function works. Finally, a named vector is returned that contains the new values of the state variables, each of which is the old value, adjusted by the transitions. Note that the state variable `cases` accumulates the number of I→R transitions and `W` accumulates the (standardized) white noise.

Now we're ready to construct the `pomp` object.

```
pomp(
     data=data.frame(
       time=seq(1/52,4,by=1/52),
       reports=NA
       ),
     times="time",
     t0=0,
     tcovar=tbasis,
     covar=basis,
     rprocess=euler.sim(
       step.fun=sir.proc.sim,
       delta.t=1/52/20
       ),
     measurement.model=reports~binom(size=cases,prob=exp(rho)),
     zeronames=c("cases"),
     initializer=function(params, t0, comp.names, ...){
       p <- exp(params)
       snames <- c("S","I","R","cases","W")
       fracs <- p[paste(comp.names,"0",sep=".")]
       x0 <- numeric(length(snames))
       names(x0) <- snames
       x0[comp.names] <- round(p['pop']*fracs/sum(fracs))
       x0
     },
     comp.names=c("S","I","R")
     ) -> sir
```

The specification of `data`, `times`, and `t0` should be familiar. The covariates are specified using the arguments `tcovar` and `covar`. We use `euler.sim` to specify the process simulator (`rprocess`). We are approximating the continuous-time process using an Euler simulator with a time-step of 1/20 of a week. Both `rmeasure` and `dmeasure` can be specified at once using the `measurement.model` argument.

Here, we model the observation process using a binomial process, where the *reporting rate*, `rho`, is the probability that a case is reported.

As we noted before, the state variable `cases` accumulates the number of cases (i.e., the number of I→R transitions). However, the data are not the cumulative number of cases, but the number of cases that have occurred since the last report. Specifying `cases` in the `zeronames` argument has the effect of re-setting the state variable `cases` to zero after each observation.

Finally, in this example, we do not use the default parameterization of the initial states. Instead, we specify a custom `initializer` argument. We want instead to parameterize the initial states in terms of the fractions of the total population contained in each compartment. In particular, as we see in the `initializer` argument to `pomp`, we normalize so that the sum of `S.0`, `I.0`, and `R.0` is 1, then multiply by the initial population size, and then round to the nearest whole number. Note that the initializer we have specified needs an argument `comp.names` (the names of the S, I, and R state variables). This is set in the last line. More generally, one can give any number or kind of additional arguments to `pomp`: they will be passed to the `initializer`, `rprocess`, `dprocess`, `rmeasure`, `dmeasure`, and `skeleton` functions, if these exist. This feature aids in the writing of customized `pomp` objects.

Now we'll simulate data using the parameters

```
theta <- c(
           gamma=26,mu=1/50,iota=10,
           beta1=1200,beta2=1500,beta3=900,
           beta.sd=1e-2,
           pop=5e5,
           rho=0.6,
           S.0=26/1200,I.0=0.001,R.0=1-0.001-26/1200
           )

sir <- simulate(sir,nsim=1,params=log(theta),seed=329348545L)
```

Figure 5 shows the simulated data and state variable trajectories.

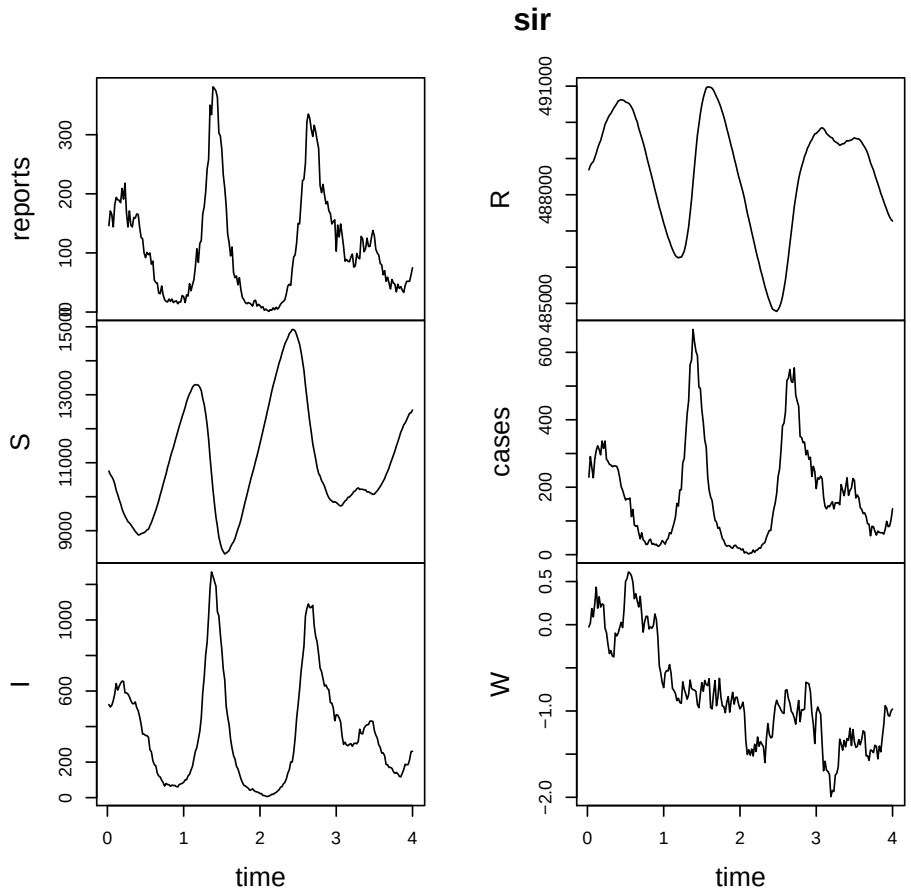

FIGURE 5. Results of `plot(sir)`.

References

M. S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp. A Tutorial on Particle Filters for Online Nonlinear, Non-Gaussian Bayesian Tracking. *IEEE Transactions on Signal Processing* **50**:174 – 188 (2002).

C. Bretó, D. He, E. L. Ionides, and A. A. King. Time series analysis via mechanistic models. *Annals of Applied Statistics* **3**:319–348 (2009).

D. T. Gillespie. Exact Stochastic Simulation of Coupled Chemical Reactions. *Journal of Physical Chemistry* **81**:2340–2361 (1977).

D. He, E. L. Ionides, and A. A. King. Plug-and-play inference for disease dynamics: measles in large and small populations as a case study. *Journal of the Royal Society Interface* **7**:271–283 (2010).

B. E. Kendall, C. J. Briggs, W. W. Murdoch, P. Turchin, S. P. Ellner, E. McCauley, R. M. Nisbet, and S. N. Wood. Why do populations cycle? A synthesis of statistical and mechanistic modeling approaches. *Ecology* **80**:1789–1805 (1999).

S. N. Wood. Statistical inference for noisy nonlinear ecological dynamic systems. *Nature* **466**:1102–1104 (2010).

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://pomp.r-forge.r-project.org