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

Date: March 20, 2014, pomp version 0.49-1.

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 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 $X_{t_{k+1}} \mid X_{t_k}$. 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 its 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*, 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 simple discrete-time population model.

We'll demonstrate the basics of pomp using a very simple discrete-time model. The plug-and-play methods in pomp were designed to work on 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 the discrete-time Gompertz model of population growth. Under this model, the density, $X_{t+\Delta t}$, of a population of plants or animals at time $t+\Delta t$ depends on the density, X_t, at time t according to

$$X_{t+\Delta t} = K^{1-e^{-r\,\Delta t}}\, X_t^{e^{-r\,\Delta t}}\, \varepsilon_t, \tag{3}$$

where K is the so-called "carrying capacity" of the population, r is a positive parameter, and the ε_t are independent and identically-distributed lognormal random variables. In different notation, this model is

$$\log X_{t+\Delta t} \;\sim\; \text{normal}(\log K + \log\left(\frac{X_t}{K}\right) e^{-r\,\Delta t}, \sigma), \tag{4}$$

where $\sigma^2 = \text{Var}[\log \epsilon_t]$. We'll assume that we can measure the population density only with error. In particular, we'll assume that errors in measurement are lognormally distributed:

$$\log Y_t \;\sim\; \text{normal}(\log X_t, \tau). \tag{5}$$

As we noted above, for this particular model, it isn't necessary to use plug-and-play methods: one can obtain exact maximum likelihood estimates of this model's parameters using the Kalman filter. We will demonstrate this below and use it to check the results of plug-and-play inference. In this document, 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 $X_{t+\Delta t} \,|\, X_t = x$ for arbitrary x and t (`rprocess`),
(2) compute the probability density function (pdf) of state transitions, i.e., compute $f(X_{t+\Delta t} = x' \,|\, X_t = x)$ for arbitrary x, x', t, and Δt (`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 $x \mapsto \mathbb{E}[X_{t+\Delta t} \,|\, X_t = x]$ 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 Gompertz model in pomp. Here, we'll take the shortest path to this goal, defining the necessary functions directly in R. 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. **NB: For all but the very simplest models, it is almost always necessary to use compiled native codes to make the computationally intensive algorithms in pomp fast enough to be useful in practice.**

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

```
require(pomp)
gompertz.proc.sim <- function (x, t, params, delta.t, ...) {
  ## unpack the parameters:
  r <- params["r"]
  K <- params["K"]
  sigma <- params["sigma"]
  ## the state at time t:
  X <- x["X"]
  ## generate a log-normal random variable:
  eps <- exp(rnorm(n=1,mean=0,sd=sigma))
  ## compute the state at time t+delta.t:
  S <- exp(-r*delta.t)
  xnew <- c(X=unname(K^(1-S)*X^S*eps))
  return(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 K, r, and σ) are passed in the argument `params`. Notice that `x` and `params` are named numeric vectors and that the output must be also be a 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 (5).

```
gompertz.meas.sim <- function (x, t, params, ...) {
  ## unpack the parameters:
  tau <- params["tau"]
  ## state at time t:
```

```
  X <- x["X"]
  ## generate a simulated observation:
  y <- c(Y=unname(rlnorm(n=1,meanlog=log(X),sd=tau)))
  return(y)
}
```

Again the translation from the model (5) is straightforward. When `gompertz.meas.sim` 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 (5).

Complementing the measurement model simulator is the corresponding measurement model density, which we can implement as follows:

```
gompertz.meas.dens <- function (y, x, t, params, log, ...) {
  ## unpack the parameters:
  tau <- params["tau"]
  ## state at time t:
  X <- x["X"]
  ## observation at time t:
  Y <- y["Y"]
  ## compute the likelihood of Y|X,tau
  f <- dlnorm(x=Y,meanlog=log(X),sdlog=tau,log=log)
  return(f)
}
```

We'll need this later on for likelihood-based inference. Note that `gompertz.meas.dens` is closely related to `gompertz.meas.sim`.

4. Simulating the model: simulate

With the two functions above, we already have all we need to simulate the full 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:

```
gompertz <- pomp(
                 data=data.frame(
                   time=1:100,
                   Y=NA
                   ),
                 times="time",
                 rprocess=discrete.time.sim(
                   step.fun=gompertz.proc.sim,
                   delta.t=1
                   ),
                 rmeasure=gompertz.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. Its second argument, delta.t, specifies the duration of the time step (by default, delta.t=1). 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 gompertz.proc.sim and gompertz.meas.sim. The parameter vector needs to specify the initial conditions $X(t_0) = x_0$ as well.

```
theta <- c(r=0.1,K=1,sigma=0.1,tau=0.1,X.0=1)
```

In addition to the parameters r, K, σ, and τ, note that we've specified the initial condition X_0 in the vector theta. The fact that the initial condition parameter's name ends in ".0" is significant: it tells pomp that this is the initial condition of the state variable X. This use of the ".0" suffix is the default behavior of pomp: one can also parameterize 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:

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

```
newly created pomp object(s):
 gompertz
```

Now gompertz 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 gompertz. We can plot the simulated data via

```
plot(gompertz,variables="Y")
```

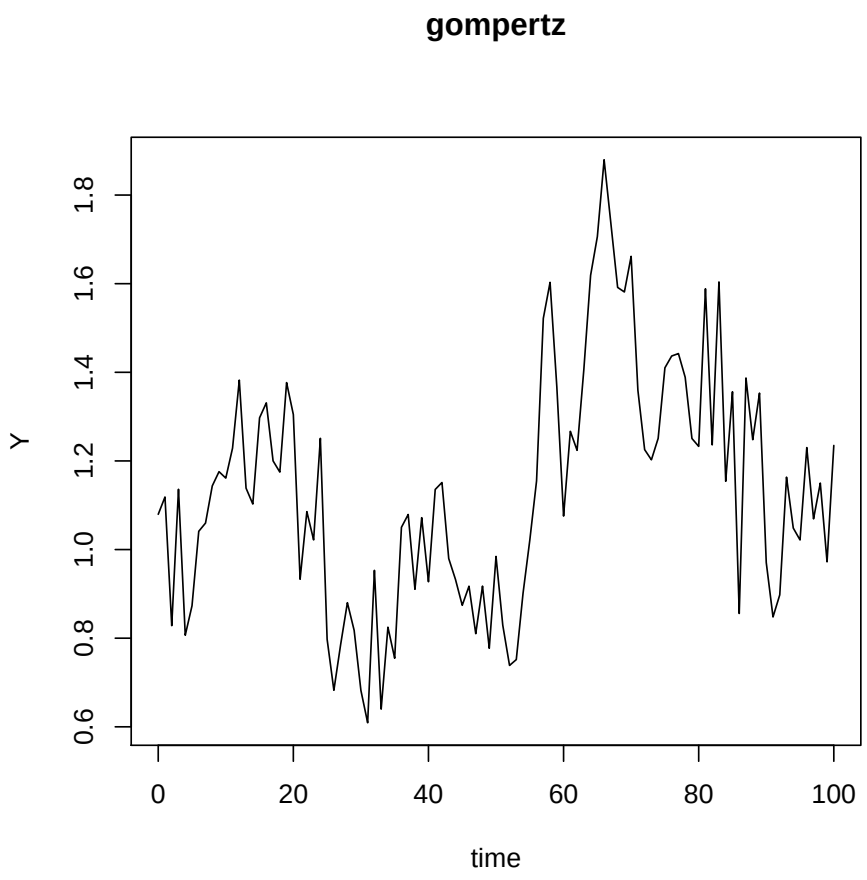

FIGURE 1. Simulated data and unobserved states from the Gompertz model (Eqs. 3–5). This figure shows the output of the command `plot(gompertz,variables="Y")`.

Fig. 1 shows the results of this operation.

5. COMPUTING LIKELIHOOD USING PARTICLE FILTERING: PFILTER

Some parameter estimation algorithms in the pomp package only require `rprocess` and `rmeasure`. These include the nonlinear forecasting algorithm `nlf` and the probe-matching algorithm `probe.match`. 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. Above, we wrote an R function, `gompertz.meas.dens`, to do this. We haven't yet used it all. To do so, we'll need to incorporate it into the `pomp` object. We can do this by specifying the `dmeasure` argument in another call to `pomp`:

```
gompertz <- pomp(
                 gompertz,
                 dmeasure=gompertz.meas.dens
                 )
```

The result of the above is a new `pomp` object `gompertz` in every way identical to the one we had before, but with the measurement-model density function `dmeasure` now specified.

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 and Ionides *et al.* (2006) for a pseudocode description of the algorithm implemented in `pomp`. We must decide how many concurrent 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(gompertz,params=theta,Np=1000)
loglik.truth <- logLik(pf)
loglik.truth
```

```
[1] 31.05209
```

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

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

or even just

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

which would have worked since the parameters are stored in the `pomp` object `gompertz`. Now let's compute the log likelihood at a different point in parameter space, one for which r, K, and σ are 50% higher than their true values.

```
theta.true <- coef(gompertz)
theta.guess <- theta.true
theta.guess[c("r","K","sigma")] <- 1.5*theta.true[c("r","K","sigma")]
pf <- pfilter(gompertz,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(gompertz)
x0 <- init.state(gompertz)
r <- coef(gompertz,"r")
K <- coef(gompertz,"K")
sigma <- coef(gompertz,"sigma")
tau <- coef(gompertz,"tau")
kf <- kalman.filter(y,x0,r,K,sigma,tau)
```

In this case, the Kalman filter gives us a log likelihood of 31.2, while the particle filter with 1000 particles gives 31.05. 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.

Box 1 Implementation of the Kalman filter for the Gompertz model.

```
kalman.filter <- function (Y, X0, r, K, sigma, tau) {
  ntimes <- length(Y)
  sigma.sq <- sigma^2
  tau.sq <- tau^2
  cond.loglik <- numeric(ntimes)
  filter.mean <- numeric(ntimes)
  pred.mean <- numeric(ntimes)
  pred.var <- numeric(ntimes)
  m <- log(X0)
  v <- 0
  S <- exp(-r)
  for (k in seq_len(ntimes)) {
    pred.mean[k] <- M <- (1-S)*log(K) + S*m
    pred.var[k] <- V <- S*v*S+sigma.sq
    q <- V+tau.sq
    r <- log(Y[k])-M
    cond.loglik[k] <- dnorm(x=log(Y[k]),mean=M,sd=sqrt(q),log=TRUE)
    q <- 1/V+1/tau.sq
    filter.mean[k] <- m <- (log(Y[k])/tau.sq+M/V)/q
    v <- 1/q
  }
  list(
       pred.mean=pred.mean,
       pred.var=pred.var,
       filter.mean=filter.mean,
       cond.loglik=cond.loglik,
       loglik=sum(cond.loglik)
       )
}
```

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(gompertz,"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(gompertz)
obs(gompertz,"Y")
time(gompertz)
```

The observation times can be changed using

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

One can respectively view and change the zero-time by

```
timezero(gompertz)
timezero(gompertz) <- -10
```

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

```
time(gompertz,t0=TRUE)
time(gompertz,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(gompertz,start=3,end=20)
```

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

```
coef(gompertz)
coef(gompertz,c("sigma","tau")) <- c(1,0)
```

See below for more information on the coef and coef<- methods for getting and setting parameters. Finally, one has access to the unobserved states via, e.g.,

```
states(gompertz)
states(gompertz,"X")
```

7. Parameter transformations

It frequently happens that it is advantageous to estimate parameters on a scale different from that on which they appear in the model. For example, the parameters in the Gompertz model above are constrained to be positive. When we estimate these parameters using numerical search algorithms, we must find some way to ensure that these constraints will be honored. A straightforward way to accomplish this is to transform the parameters so that they become unconstrained. To introduce some terminology, we want to transform the parameters from the *natural scale* to another, *estimation scale*, on which they will be unconstrained.

pomp provides a facility to make it easier to work with parameter transformations. Specifically, we can specify the optional functions `parameter.transform` and `parameter.inv.transform` when we create the pomp object. The first function will take transform our parameters from the estimation scale to the natural scale. the second one will invert that operation, transforming the parameters from the natural to the estimation scale. In our Gompertz model example, the natural-scale parameters are r, K, σ, τ, and X_0, all of which are constrained to be positive. We'll log-transform all the parameters to enforce this constrant.

```
gompertz <- pomp(
                 gompertz,
                 parameter.transform=function(params,...){
                   exp(params)
                 },
                 parameter.inv.transform=function(params,...){
                   log(params)
                 }
                 )
```

Note that the function given to the `parameter.transform` argument transforms parameters *to* the natural scale while that given to the `parameter.inv.transform` argument transforms them *from* the natural scale. Operationally, the defining property of the natural scale is that this is the scale on which the elementary functions `rprocess`, `rmeasure`, `dprocess`, `dmeasure`, `skeleton`, and `init.process` expect to see the parameters. Note also that we've changed the names of the parameters to make it obvious at a glance what scale we're on, but that this isn't really necessary.

The parameter transformations come into play in the `coef` and `coef<-` methods for getting and setting parameters. If we have a vector of parameters on the natural scale, one can set them as parameters of `gompertz` by doing

```
coef(gompertz) <- c(r=0.1,K=1,tau=0.1,sigma=0.1,X.0=1)
```

and get the parameters by

```
coef(gompertz)
```

```
   r     K   tau sigma   X.0
 0.1   1.0   0.1   0.1   1.0
```

On the other hand, given parameter values on the estimation scale, one can set the parameters by

```
coef(gompertz,transform=TRUE) <- c(r=log(0.1),K=0,tau=log(0.1),
                sigma=log(0.1),X.0=0)
```

and get them by

```
coef(gompertz,transform=TRUE)
```

```
       r         K       tau     sigma       X.0
-2.302585  0.000000 -2.302585 -2.302585  0.000000
```

We can check that the parameters were set correctly on the natural scale:

```
coef(gompertz)
```

```
   r     K   tau sigma   X.0
 0.1   1.0   0.1   0.1   1.0
```

To be clear: the parameter-setting expression `coef(gompertz) <- theta` assumes that `theta` is a named vector of parameters on the natural scale, while `coef(gompertz,transform=TRUE) <- theta` assumes that `theta` is on the estimation scale.

Note that it's the user's responsibility to ensure that the `parameter.transform` and `parameter.inv.transform` are actually mutually inverse. A simple (but not foolproof) test of this is

```
# use parameter.inv.transform:
theta <- coef(gompertz,transform=TRUE)
## theta is on the estimation scale
g2 <- gompertz
## use parameter.transform:
coef(g2,transform=TRUE) <- theta
## compare the internal-scale representations:
identical(coef(gompertz),coef(g2))
```

```
[1] TRUE
```

A `pomp` object corresponding to the one just created (but with the `rprocess`, `rmeasure`, and `dmeasure` bits coded in C for speed) can be loaded by executing `pompExample(gompertz)`. For your convenience, codes creating this `pomp` object are included with the package. To view them, run the demo

```
demo(gompertz)
```

8. Estimating parameters using iterated filtering: mif

Iterated filtering is a technique for maximizing the likelihood obtained by filtering. In pomp, it is the particle filter that is iterated. Iterated filtering is implemented in the `mif` function. For a description of the algorithm and a description of its theoretical basis, see Ionides *et al.* (2006). A more complete set of proofs is provided in Ionides *et al.* (2011).

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 `gompertz`. We'll initialize the algorithm at several starting points around `theta.true` and just estimate the parameters r, τ, and σ:

```
newly created pomp object(s):
 gompertz
```

```
 estpars <- c("r","sigma","tau")
 replicate(
           n=10,
           {
             theta.guess <- theta.true
             theta.guess[estpars] <- rlnorm(
                                          n=length(estpars),
                                          meanlog=log(theta.guess[estpars]),
                                          sdlog=1
                                          )
             mif(
                 gompertz,
                 Nmif=100,
                 start=theta.guess,
                 transform=TRUE,
                 pars=estpars,
                 rw.sd=c(r=0.02,sigma=0.02,tau=0.05),
                 Np=2000,
                 var.factor=4,
                 ic.lag=10,
                 cooling.type="geometric",
                 cooling.fraction=0.95,
                 max.fail=10
                 )
           }
           ) -> mf
```

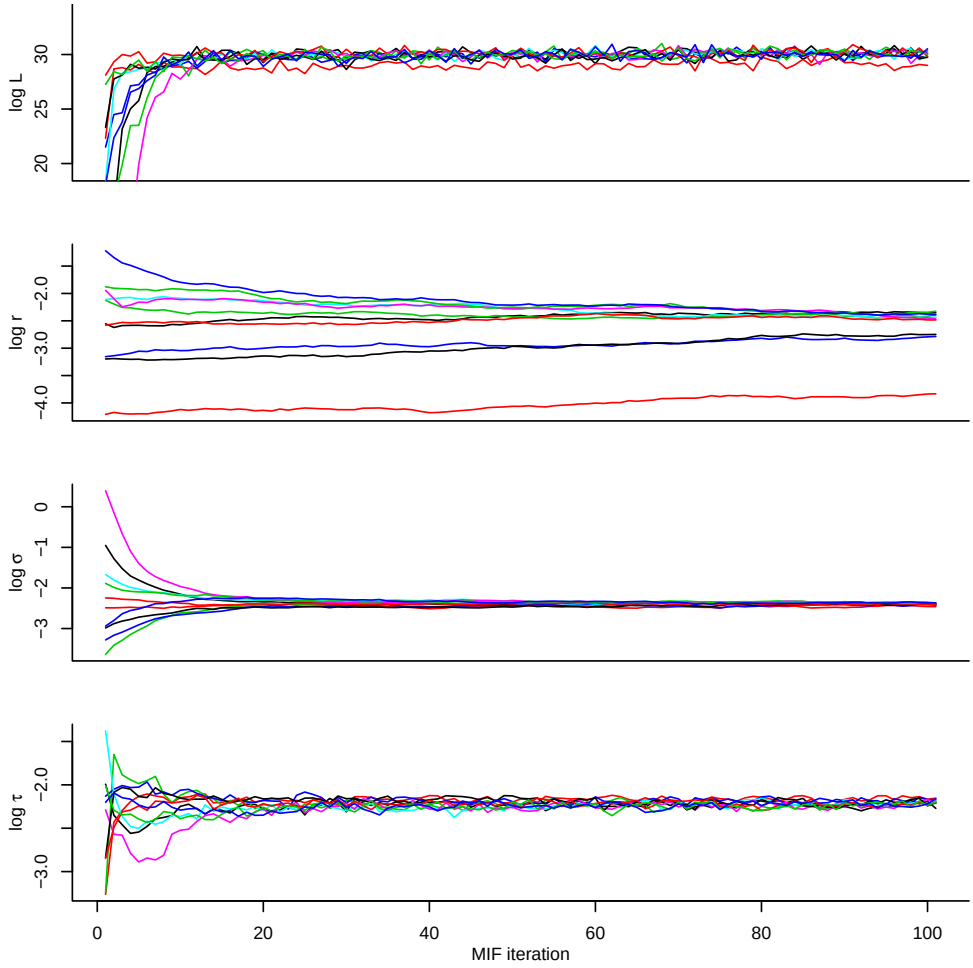

FIGURE 2. Convergence plots can be used to help diagnose convergence of the iterated filtering algorithm. This shows part of the output of `compare.mif(mf)`.

Note that we've set `transform=TRUE` in the above. This means that the likelihood maximization is done on the estimation scale (see the section above on Parameter Transformations).

Each of the 10 `mif` runs ends up at a different place. In this case (but by no means in every case), we can average across these parameter estimates to get an approximate maximum likelihood estimate. We'll evaluate the likelihood several times at this estimate to get an idea of the Monte Carlo error in our likelihood estimate. The particle filter produces an unbiased estimate of the likelihood; therefore, we'll average the likelihoods, not the log likelihoods.

```
theta.true <- coef(gompertz)
theta.mif <- apply(sapply(mf,coef),1,mean)
loglik.mif <- replicate(n=10,logLik(pfilter(mf[[1]],params=theta.mif,Np=10000)))
loglik.mif.est <- logmeanexp(loglik.mif,se=TRUE)
loglik.true <- replicate(n=10,logLik(pfilter(gompertz,params=theta.true,Np=10000)))
loglik.true.est <- logmeanexp(loglik.true,se=TRUE)
```

```
          r  sigma   tau loglik loglik.se.se
mle   0.078 0.0906 0.111   31.7        0.094
truth 0.100 0.1000 0.100   31.5        0.150
```

9. 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

$$x \mapsto \mathbb{E}\left[X_{t+\Delta t} \mid X_t = x, \theta\right]. \tag{6}$$

In the continuous-time case, this is the vectorfield

$$x \mapsto \lim_{\Delta t \to 0} \mathbb{E}\left[\frac{X_{t+\Delta t} - x}{\Delta t} \;\middle|\; X_t = x, \theta\right]. \tag{7}$$

Our discrete-time Gompertz has the deterministic skeleton

$$x \mapsto K^{1-S}\, x^S, \tag{8}$$

where $S = e^{-r\,\Delta t}$ and Δt is the time-step. This can be implemented in the R function

```
gompertz.skel <- function (x, t, params, ...) {
  r <- params["r"]
  K <- params["K"]
  X <- x["X"]
  S <- exp(-r)
  xnew <- c(X=unname(K^(1-S)*X^S))
  return(xnew)
}
```

Note that we have here assumed that $\Delta t = 1$.

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

```
new.gompertz <- pomp(
                     gompertz,
                     skeleton.type="map",
                     skeleton=gompertz.skel
                     )
coef(new.gompertz,"X.0") <- 0.1
coef(new.gompertz,"r") <- 0.1
coef(new.gompertz,"tau") <- 0.05
coef(new.gompertz,"sigma") <- 0.05
new.gompertz <- simulate(new.gompertz,seed=88737400L)
```

We use `skeleton.type="map"` for discrete-time processes (such as the Gompertz model) and `skeleton.type="vectorfield"` for continuous-time processes.

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. The following codes perform trajectory matching. Since we set `transform=TRUE`, the optimization will be performed on the estimation scale.

```
tm <- traj.match(
                 new.gompertz,
                 start=coef(new.gompertz),
                 transform=TRUE,
                 est=c("r","K","tau","X.0"),
```

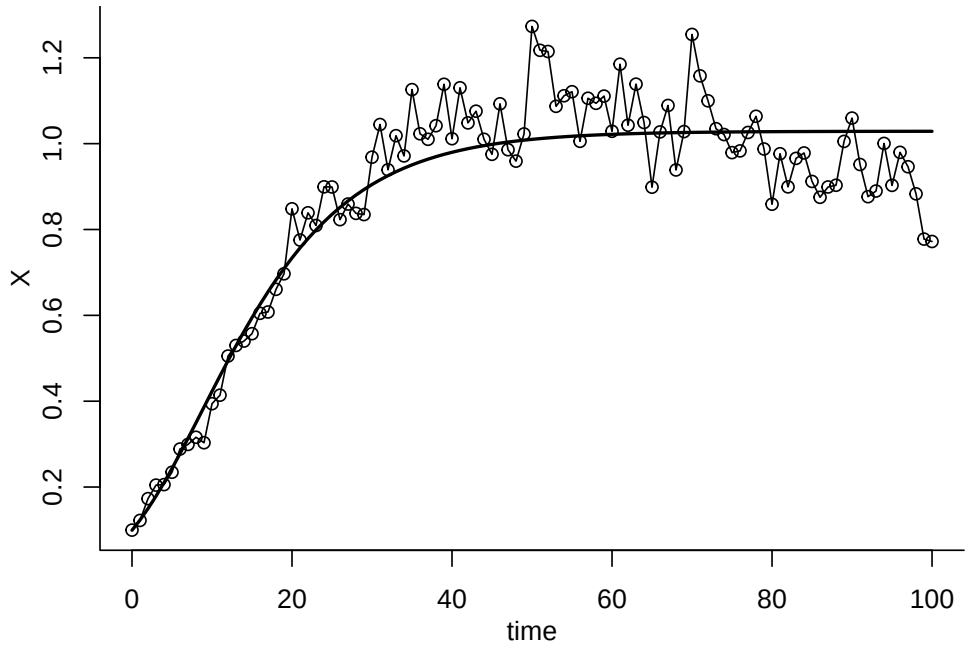

FIGURE 3. Illustration of trajectory matching. The points show data simulated from `new.gompertz`. The solid line shows the trajectory of the best-fitting model, obtained using `traj.match`. Fitting by trajectory matching is tantamount to the assumption that the data-generating process has no process noise but only measurement error.

```
method="Nelder-Mead",
maxit=1000,
reltol=1e-8
)
```

Fig. 3 shows the results of this fit.

10. 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, delta.t, ...) {
  e <- rnorm(n=1,mean=0,sd=params["sigma"])
  setNames(
           c(
             params["r"]*x["N"]*exp(-x["N"]+e),
             e
             ),
           c("N","e")
           )
}
```

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*phi)
               )
coef(ricker) <- c(
                  r=exp(3.8),
                  sigma=0.3,
                  phi=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

```
pompExample(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(r=20,sigma=1,phi=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("r","sigma","phi"),
                  transform=TRUE,
```

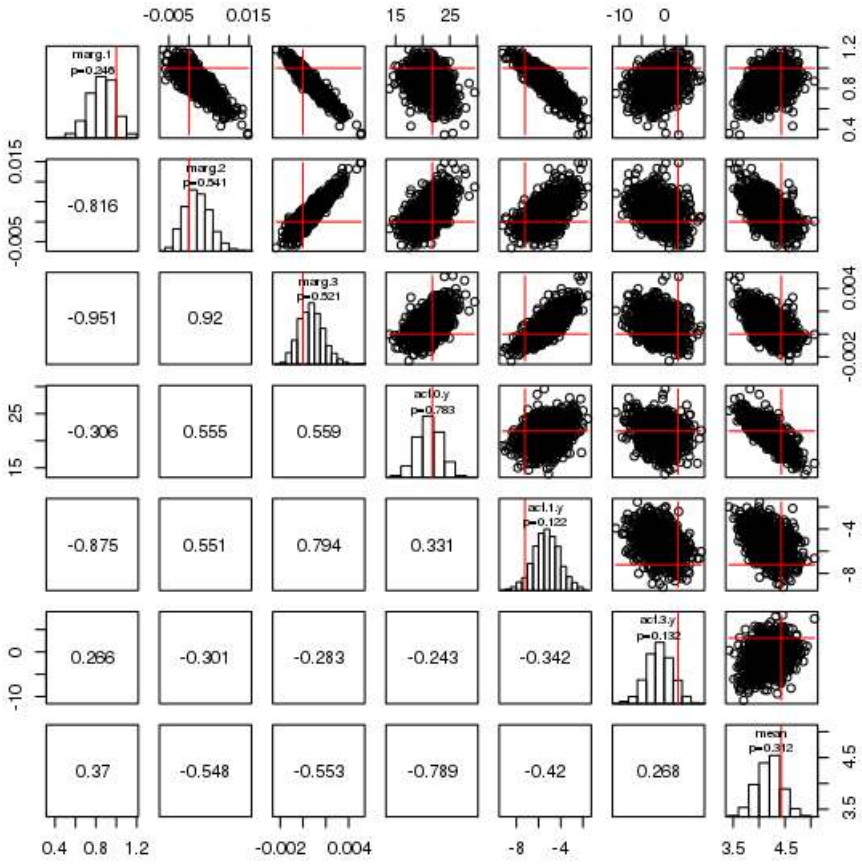

FIGURE 4. Results of `plot` on a `probed.pomp`-class object. Above the diagonal, the pairwise scatterplots show the values of the probes on each of 1000 data sets. The red lines show the values of each of the probes on the data. The panels along the diagonal show the distributions of the probes on the 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.

```
                method="Nelder-Mead",
                maxit=2000,
                seed=1066L,
                reltol=1e-8,
                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,
          transform=TRUE,
          cooling.type="geometric",
          cooling.fraction=0.6,
          var.factor=2,
          ic.lag=3,
          max.fail=50,
          rw.sd=c(r=0.1,sigma=0.1,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(
               logLik(pf.guess),logLik(pf.truth),logLik(pf.mf),logLik(pf.pm)
               ),
             synth.loglik=c(
               logLik(pb.guess),logLik(pb.truth),logLik(pb.mf),logLik(pm)
               )
             )

print(res,digits=3)
```

```
              guess  truth      MLE       PM
r              20.0   44.7   44.341   42.105
sigma           1.0    0.3    0.175    0.337
phi            20.0   10.0   10.231   11.255
N.0             7.0    7.0    7.000    7.000
e.0             0.0    0.0    0.000    0.000
loglik       -230.9 -139.5 -136.733 -145.580
synth.loglik  -12.3   17.5   17.736   20.385
```

11. Nonlinear forecasting: nlf

NLF is an indirect inference approach (Gouriéroux and Monfort, 1996), meaning that an intermediate statistical model is used to quantify the model's goodness of fit to the data. Specifically, NLF is a *Simulated Quasi-Maximum Likelihood* (SQML) method. The quasilikelihood function is defined by fitting a convenient statistical model to a long simulation output from the model of interest, and then evaluating the statistical model's likelihood function on the data. The intermediate statistical model in `nlf` is a multivariate generalized additive autoregressive model, using radial basis functions as the ridge functions and multivariate Gaussian process noise. Smith (1993) first proposed SQML and developed the underlying statistical theory, Tidd *et al.* (1993) independently proposed a similar method, and Kendall *et al.* (2005) describe in detail the methods used by `nlf` and use them to fit and compare models for insect population cycles.

As a simple example we can use `nlf` to estimate the parameters `K` and `r` of the Gompertz model. An example of a minimal call, accepting the defaults for all optional arguments, is

```
pompExample(gompertz)
out <- nlf(
           gompertz,
           start=c(r=1,K=2,sigma=0.5,tau=0.5,X.0=1),
           partrans=TRUE,
           est=c("K","r"),
           lags=c(1,2)
           )
```

where the first argument is the `pomp` object, `start` is a vector containing model parameters at which `nlf`'s search will begin, `est` contains the names of parameters `nlf` will estimate, and `lags` specifies which past values are to be used in the autoregressive model. In the call above `lags=c(1,2)` specifies that the autoregressive model predicts each observation, y_t using y_{t-1} and y_{t-2}, the two most recent past observations. The set of lags need not include the most recent observation, and skips are allowed, so that `lags=c(2,3,6)` is also "legal".

The quasilikelihood is optimized numerically, so the reliability of the optimization should be assessed by doing multiple fits with different starting parameter values. Because of the way `nlf` controls the random number seed, starting values should all be chosen before the calls to `nlf`:

```
# pick 5 random starting parameter values
starts <- replicate(n=5,
                    {
                      p <- coef(gompertz)
                      p[c("K","r")] <- rlnorm(n=2,meanlog=log(p[c("K","r")]),
                                              sdlog=0.1)
                      p
                    },
                    simplify=FALSE
                    )
```

Then to make the results from different starts comparable, use the `seed` argument to initialize the random number generator the same way for each fit:

```
out <- list()
## Do the fitting.
## method, trace, and nasymp are explained below
for (j in 1:5) {
```

```
  out[[j]] <- nlf(
                  gompertz,
                  start=starts[[j]],
                  transform=log,
                  transform.params=TRUE,
                  est=c("K","r"),
                  lags=c(1,2),
                  seed=7639873L,
                  method="Nelder-Mead",
                  trace=4,
                  skip.se=TRUE,
                  nasymp=5000
                  )
}
fits <- t(sapply(out,function(x)c(x$params[c("r","K")],value=x$value)))
```

The results in this case are very encouraging,

```
fits
```

```
             r        K    value
[1,] 0.1121845 1.118521 40.33674
[2,] 0.1121513 1.118505 40.33674
[3,] 0.1121514 1.118460 40.33674
[4,] 0.1121546 1.118493 40.33674
[5,] 0.1121198 1.118417 40.33674
```

so below we will trust that repeated optimization isn't needed.

The call above also used the `method` argument to specify that the Nelder-Mead option in `optim` is used to maximize the quasilikelihood, and the `trace` argument is passed to `optim`; other arguments can be passed to `optim` in the same way. `nasymp` sets the length of the Gompertz model simulation on which the quasilikelihood is based; larger values will give less variable parameter estimates, but will slow down the fitting process. The slowdown is dominated by the time required to generate the model simulations, so efficient coding of `rprocess` is essential. The "Advanced topics in pomp" vignette gives some advice on this. Do `vignette("advanced_topics_in_pomp")` to view it.

The choice of lags affects the accuracy of the intermediate statistical model and therefore the accuracy of parameter estimates, so it is worth putting some effort into choosing good lags. Given enough time, a user could generate many artificial data sets, fit them all with several candidate lags, and evaluate the precision and accuracy of estimates. A quicker approach is to explore the shape and variability of the quasilikelihood function near pilot parameter estimates for several candidate sets of lags, using `nlf` with `eval.only=TRUE` to evaluate the quasilikelihood without performing optimization.

For the Gompertz model the complete state vector is observed, so it is plausible that forecasting based on the one most recent observation is optimal, i.e. `lags=1`. But because of measurement error, prediction based on multiple lags might be more accurate and more sensitive to parameter values, and longer-term forecasting might be beneficial if the effects of parameters are amplified over time. Fig. 5 shows results for several candidate lags suggested by these considerations. To reduce Monte Carlo error in the objective function, we used `simulate` to create a very long "data set":

```
long.gomp <- simulate(gompertz,times=1:1000)
theta <- coef(long.gomp)
```

and then evaluated the quasilikelihood for a range of parameter values:

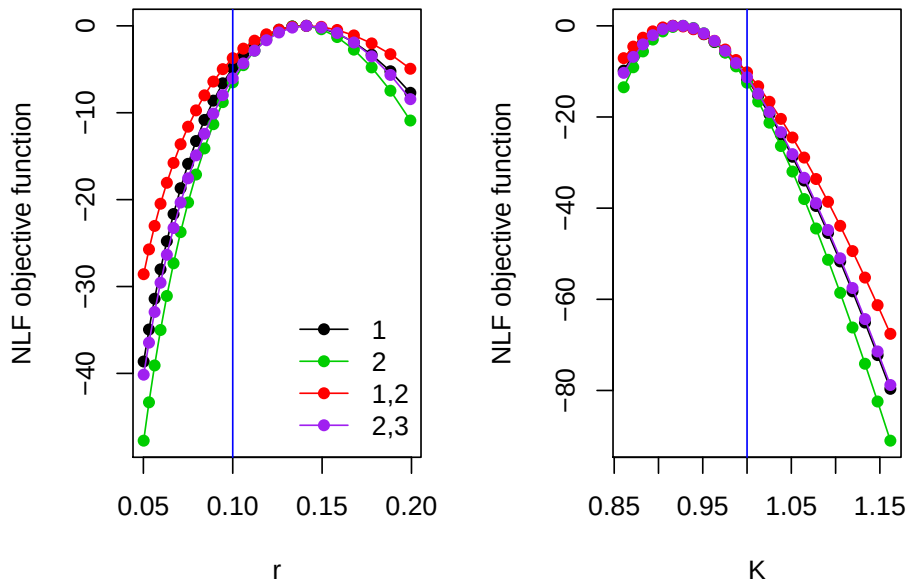

FIGURE 5. Values of the NLF objective function (log of the quasilikelihood) for the Gompertz model as a function of the parameters `r`,`K` for various choices of the `lags` argument. All plotted curves were shifted vertically so as to have maximum value zero. The objective function was evaluated on an artificial data set of length 1000 that was generated assuming `r=0.1`, `K=1`, indicated by the vertical blue lines.

```
lags <- list(1,2,c(1,2),c(2,3))
r.vals <- theta["r"]*exp(seq(-0.69,0.69,length=25))
fvals <- matrix(nrow=25,ncol=4)
for (j in 1:25) {
  pars <- theta
  pars["r"] <- r.vals[j]
  for(k in 1:4) {
    fvals[j,k] <- nlf(
                      long.gomp,
                      start=pars,
                      nasymp=5000,
                      est=NULL,
                      lags=lags[[k]],
                      eval.only=TRUE
                      )
  }
}
```

Based on Fig. 5, `lags=2` seems like a good choice. Another consideration is the variability of parameter estimates on multiple short data sets:

```
nreps <- 100
ndata <- 60
fvals <- matrix(nrow=nreps,ncol=length(lags))
new.pomp <- simulate(gompertz,times=1:ndata,nsim=nreps,seed=NULL) # nreps simulated data sets
for (j in 1:nreps) {
```

```
  for (k in seq_along(lags)) {
    fvals[j,k] <- nlf(
                      new.pomp[[j]],
                      start=coef(gompertz),
                      nasymp=5000,
                      est=NULL,
                      lags=lags[[k]],
                      eval.only=TRUE
                      )
  }
}
fvals <- exp(fvals/ndata)
```

The last line above expresses the objective function as the geometric mean (quasi)likelihood per data point. The variability across data sets was nearly the same for all lags:

```
apply(fvals,2,function(x)sd(x)/mean(x))
```

```
[1] 0.1462303 0.1476348 0.1397683 0.1474704
```

so we proceed to fit with `lags=2`.

```
true.fit <- nlf(
                gompertz,
                transform.params=TRUE,
                est=c("K","r"),
                lags=2,
                seed=7639873,
                method="Nelder-Mead",
                trace=4,
                nasymp=5000
                )
```

From `true.fit$params` and `true.fit$se` we get the estimates ($\pm$ 1 standard error) $r = 0.11 \pm 0.062$ and $K = 1.1 \pm 0.084$.

The standard errors provided by `nlf` are based on a Newey-West estimate of the variance-covariance matrix that is generally somewhat biased downward. More importantly, these rough-and-ready standard error estimates can be unstable. This is because they are obtained from finite differences of the NLF objective function. This function, in turn, is approximated using simulated time series of finite length, which typically gives rise to fine-scale wrinkles. Therefore, when time permits, bootstrap standard errors are preferable. When `nlf` is called with `bootstrap=TRUE`, the quasilikelihood function is evaluated on the bootstrap sample of the time series specified in `bootsamp`. The first `max(lags)` observations cannot be forecast by the autoregressive model, so the size of the bootstrap sample is the length of the data series minus `max(lags)`:

```
lags <- 2
ndata <- length(obs(gompertz))
nboot <- ndata-max(lags)
nreps <- 100
pars <- matrix(0,nreps,2)
bootsamp <- replicate(n=nreps,sample(nboot,replace=TRUE))
for (j in seq_len(nreps)) {
  fit <- nlf(
```

```
              gompertz,
              start=coef(gompertz),
              transform.params=TRUE,
              est=c("K","r"),
              lags=lags,
              seed=7639873,
              bootstrap=TRUE,
              bootsamp=bootsamp[,j],
              skip.se=TRUE,
              method="Nelder-Mead",
              trace=4,
              nasymp=5000
              )
   pars[j,] <- fit$params[c("r","K")]
}
colnames(pars) <- c("r","K")

apply(pars,2,sd)
```

```
         r          K
0.03789826 0.08255798
```

In this case, the bootstrap standard errors don't differ much from the Newey-West estimates.

The code above implements a "resampling cases" approach to bootstrapping the data set to which the intermediate autoregressive model is fitted. This is valid when observations are conditionally independent given the past observations, which is only true for a Markov process if the complete state is observed. Otherwise there may be correlations, and we need to use methods for bootstrapping time series. In `nlf` it is relatively easy to implement the "blocks of blocks" resampling method (Davison and Hinkley, 1997, p. 398). For example, with block length $l = 3$ we resample (with replacement) observations in groups of length 3:

```
bootsamp <- replicate(n=nreps,sample(nboot,size=floor(nboot/3),replace=TRUE))
bootsamp <- rbind(bootsamp,bootsamp+1,bootsamp+2)
```

and otherwise proceed exactly as above.

12. Bayesian sequential Monte Carlo: bsmc

The approximate Bayesian sequential Monte Carlo method of Liu and West (2001) is implemented in pomp. The following demonstrates its use on the Ricker model.

First, we'll specify a prior distribution. Bayesian sequential Monte Carlo requires only that we be able to simulate from the prior, so we'll need to write a function that draws samples. We then give this function to the `rprior` argument of `pomp`. Let's use uniform priors on $\log r$ and σ, leaving the others fixed at their true values.

```
pompExample(ricker)
ricker <- pomp(
               ricker,
               rprior=function (params, ...) {
                 params["r"] <- exp(runif(n=1,min=2,max=5))
                 params["sigma"] <- runif(n=1,min=0.1,max=1)
                 params
               }
               )
```

We'll use 10,000 particles, so we'll need that many samples from the prior distribution. The following runs the Bayesian sequential Monte Carlo algorithm with 10,000 particles. Note that, by specifying `transform=TRUE`, we cause the estimation to proceed on the transformed scale.

```
  fit1 <- bsmc(ricker,Np=10000,transform=TRUE,
               est=c("r","sigma"),smooth=0.2,
               seed=1050180387L)
```

Fig. 6 shows the results of this computation. Obtain the posterior medians of the parameters by doing

```
signif(coef(fit1),digits=2)
```

```
   r sigma   phi   N.0   e.0
49.00  0.11 10.00  7.00  0.00
```

Note that these are reported on the natural (i.e., untransformed) scale.

To repeat the procedure with log-normal priors on r and σ, one might do the following.

```
ricker <- pomp(ricker,
               rprior=function (params, ...) {
                 x <- rlnorm(n=2,meanlog=c(4,log(0.5)),sdlog=c(3,5))
                 params[c("r","sigma")] <- x
                 params
               }
               )
fit2 <- bsmc(ricker,transform=TRUE,Np=10000,
             est=c("r","sigma"),smooth=0.2,
             seed=90348704L)
```

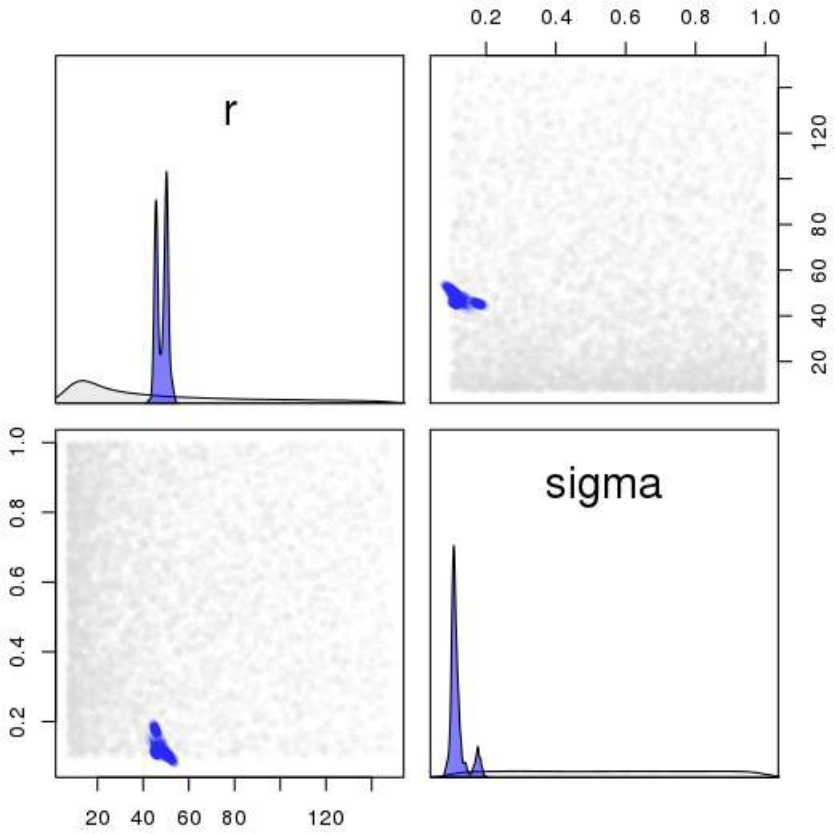

FIGURE 6. Results of `bsmc` on the Ricker model. The off-diagonal panels show 5000 samples from the prior (grey) and posterior (blue) distributions. The diagonal panels show kernel density estimates of the marginal prior and posterior distributions for each of the parameters. Note that these are shown on the natural scale. This plot was produced by executing `plot(fit1,pars=c("r","sigma"),thin=5000)`.

```
signif(coef(fit2),digits=2)
```

```
    r sigma   phi   N.0   e.0
49.00  0.24 10.00  7.00  0.00
```

13. Particle Markov chain Monte Carlo: `pmcmc`

Andrieu *et al.* (2010). To be added.

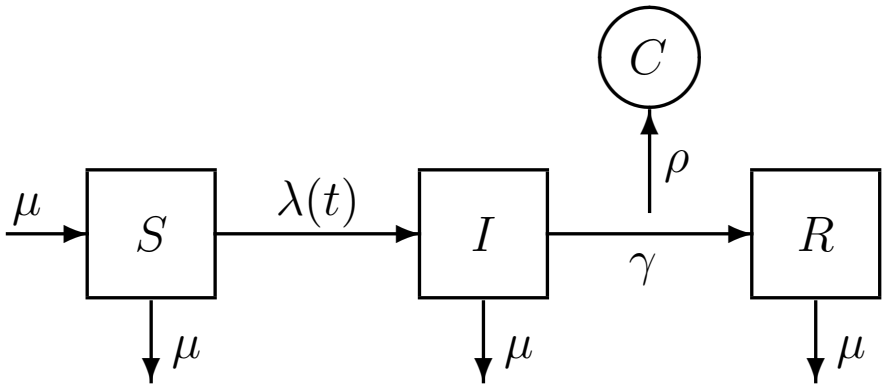

FIGURE 7. Diagram of the SIR model. The host population is divided into three classes according to their infection status: S, susceptible hosts; I, infected (and infectious) hosts; R, recovered and immune hosts. Births result in new susceptibles and all individuals have a common death rate μ. Since the birth rate equals the death rate, the expected population size, $N = S + I + R$, remains constant. The S→I rate, λ, called the *force of infection*, depends on the number of infectious individuals according to $\lambda(t) = \beta\, I/N$. The I→R, or recovery, rate is γ. The case reports, C, result from a process by which infections are recorded with probability ρ. Since diagnosed cases are treated with bed-rest and hence removed, infections are counted upon transition to R.

14. A more complex example: a seasonal epidemic model

The stochastic SIR model. A mainstay of theoretical epidemiology, the SIR model describes the progress of a contagious infection through a population of hosts. The hosts are divided into three classes, according to their status vis-a-vis the infection (Fig. 7). The S class contains those that have not yet been infected and are thereby still susceptible to it; the I class comprises those who are currently infected and, by assumption, infectious; the R class includes those who have recovered from the infection. The latter are assumed to be immune against reinfection. We let $S(t)$, $I(t)$, and $R(t)$ represent the numbers of individuals within the respective classes at time t.

It is natural to formulate this model as a continuous-time Markov process. In this process, the numbers of individuals within each class change through time in whole-number increments. In particular, individuals move between classes (entering S at birth, moving thence to I, and on to R unless death arrives first) at random times. Thus, the numbers of births and class-transitions that occur in any interval of time are random variables. The birth rate, death rates, and the rate of transition, γ, from I to R are frequently assumed to be constants, specific to the infection and the host population. Crucially, the S to I transition rate, the so-called *force of infection*, is not constant, but depends on the current number of infectious individuals. The assumption that transmission is *frequency dependent*, as for example when each individual realizes a fixed number of contacts per unit time, corresponds to the assumption $\lambda(t) = \beta\, I(t)/N$, where β is known as the contact rate and $N = S + I + R$ is the population size. This assumption introduces the model's only nonlinearity. It is useful sometimes to further assume that birth and death rates are equal and independent of infection status—call the common rate μ—which has the consequence that the expected population size then remains constant.

It is typically impossible to monitor S, I, and R, directly. Relatively commonly, however, records of *cases*, i.e., individual infections, are kept by public health authorities. The number of cases, $C(t_1, t_2)$, recorded within a given reporting interval $[t_1, t_2)$ might perhaps be modeled by a binomial process

$$C(t_1, t_2) \;\sim\; \text{binomial}(\Delta_{I\to R}(t_1, t_2), \rho)$$

where $\Delta_{I\to R}(t_1,t_2)$ is the accumulated number of recoveries that have occured over the interval in question and ρ is the *reporting rate*, i.e., the probability that a given infection is observed and recorded. The fact that the observed data are linked to an accumulation, as opposed to an instantaneous value, introduces a little bit of complication; see the section on "Accumulator variables" in the "Advanced Topics in pomp" vignette for a thorough discussion of this issue.

The model's deterministic skeleton is a system of nonlinear ordinary differential equations—a vectorfield—on the space of positive values of S, I, and R (cf. Eq. 7). Specifically, the SIR deterministic skeleton is

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

Implementing the SIR model in pomp. As before, we'll need to write functions to implement some or all of the SIR model's `rprocess`, `dprocess`, `rmeasure`, `dmeasure`, and `skeleton` components. It turns out to be relatively straightforward to implement all of these but `dprocess`.

For the `rprocess` portion, we can use `gillespie.sim` to implement the continuous-time Markov process exactly using the stochastic simulation algorithm of Gillespie (1977). For many practical purposes, however, this will prove quite slow and inefficient. If we are willing to live with an approximate simulation scheme, we can use the the so-called "tau-leap" algorithm, one version of which is implemented in pomp as the `euler.sim` plug-in. This algorithm holds the transition rates λ, μ, γ constant over a small interval of time δt and simulates the numbers of births, deaths, and transitions that occur over that interval. It then updates the state variables S, I, R accordingly, increments the time variable by δt, recomputes the transition rates, and repeats. Naturally, as $\delta t \to 0$, this approximation to the true continuous-time process becomes better and better. The critical feature from the inference point of view, however, is that no relationship need be assumed between the Euler simulation interval δt and the reporting interval, which itself need not even be the same from one observation to the next.

Under this assumption, the number of individuals leaving any of the classes by all available routes over a particular time interval becomes a multinomial process. In particular, the probability that an S individual, for example, becomes infected is $p_{S\to I} = \frac{\lambda(t)}{\lambda(t)+\mu}\,(1-e^{-(\lambda(t)+\mu)\,\delta t})$; the probability that an S individual dies before becoming infected is $p_{S\to} = \frac{\mu}{\lambda(t)+\mu}\,(1-e^{-(\lambda(t)+\mu)\,\delta t})$; and the probability that neither happens is $1-p_{S\to I}-p_{S\to} = e^{-(\lambda(t)+\mu)\,\delta t}$. Thus, if $\Delta_{S\to I}$ and $\Delta_{S\to}$ are the numbers of S individuals acquiring infection and dying, respectively, in the Euler simulation interval $(t,t+\delta t)$, then

$$(\Delta_{S\to I},\Delta_{S\to},S-\Delta_{S\to I}-\Delta_{S\to}) \;\sim\; \text{multinomial}\,(S(t);p_{S\to I},p_{S\to},1-p_{S\to I}-p_{S\to})\,,$$

Now, the expression on the right arises with sufficient frequency in compartmental models like the SIR that pomp has special functions for it. In pomp, the random variable $(\Delta_{S\to I},\Delta_{S\to})$ above is said to have an *Euler-multinomial* distribution. The pomp functions `reulermultinom` and `deulermultinom` provide facilities for drawing random deviates from, and computing the p.d.f. of, such distributions. As the help pages relate, `reulermultinom` and `deulermultinom` parameterize the Euler-multinomial distributions by the size ($S(t)$ in the example above), rates ($\lambda(t)$ and μ), and time interval δt. Obviously, the Euler-multinomial distributions generalize to an arbitrary number of exit routes.

The help (`?euler.sim`) informs us that to use `euler.sim`, we need to specify a function that advances the states from t to $t+\delta t$. The function `sir.proc.sim`, defined here, does this.

```
sir.proc.sim <- function (x, t, params, delta.t, ...) {
  ## unpack the parameters
  N <- params["N"]                # population size
```

```
  gamma <- params["gamma"]     # recovery rate
  mu <- params["mu"]           # birth rate = death rate
  beta <- params["beta"]       # contact rate
  foi <- beta*x["I"]/N                      # the force of infection
  trans <- c(
            rpois(n=1,lambda=mu*N*delta.t), # births are assumed to be Poisson
            reulermultinom(n=1,size=x["S"],rate=c(foi,mu),dt=delta.t), # exits from S
            reulermultinom(n=1,size=x["I"],rate=c(gamma,mu),dt=delta.t), # exits from I
            reulermultinom(n=1,size=x["R"],rate=c(mu),dt=delta.t)        # exits from R
            )
  ## now connect the compartments
  x[c("S","I","R","cases")]+c(
      trans[1]-trans[2]-trans[3],
      trans[2]-trans[4]-trans[5],
      trans[4]-trans[6],
      trans[4]                          # accumulate the recoveries
    )
}
```

Two significant wrinkles remains to be explained. First, notice that in `sir.proc.sim`, the state variable `cases` accumulates the total number of recoveries. Thus, `cases` will be a counting process and, in particular, will be nondecreasing with time. In fact, the number of recoveries within an interval, $\Delta_{I\to R}(t_1,t_2) = \texttt{cases}(t_2) - \texttt{cases}(t_1)$. Clearly, including `cases` as a state variable violates the Markov assumption.

However, this is not an essential violation. Because none of the rates λ, μ, or γ depend on `cases`, the process remains essentially Markovian. We still have a difficulty with the measurement process, however, in that our data are assumed to be records of infections resolving within a given interval while the process model keeps track of the accumulated number of infections that have resolved since the record-keeping began. We can get around this difficulty by re-setting `cases` to zero immediately after each observation. We tell `pomp` to do this using the `pomp`'s `zeronames` argument, as we will see in a moment. The section on "accumulator variables" in the "Advanced topics in `pomp`" vignette discusses this in more detail.

The second wrinkle has to do with the initial conditions, i.e., the states $S(t_0)$, $I(t_0)$, $R(t_0)$. By default, `pomp` will allow us to specify these initial states arbitrarily. For the model to be consistent, they should be positive integers that sum to the population size N. We can enforce this constraint by customizing the parameterization of our initial conditions. We do this in by specializing a custom `initializer` in the call to `pomp` that constructs the `pomp` object. Let's construct it now and fill it with simulated data.

```
simulate(
         pomp(
              data=data.frame(
                time=seq(1/52,15,by=1/52),
                reports=NA
                ),
              times="time",
              t0=0,
              rprocess=euler.sim(
                step.fun=sir.proc.sim,
                delta.t=1/52/20
                ),
              measurement.model=reports~binom(size=cases,prob=rho),
              initializer=function(params, t0, ic.pars, comp.names, ...){
                x0 <- c(S=0,I=0,R=0,cases=0)
```

```
    N <- params["N"]
    fracs <- params[ic.pars]
    x0[comp.names] <- round(N*fracs/sum(fracs))
    x0
  },
  zeronames=c("cases"), # 'cases' is an accumulator variable
  ic.pars=c("S0","I0","R0"), # initial condition parameters
  comp.names=c("S","I","R") # names of the compartments
  ),
params=c(
    N=50000,
    beta=60,gamma=8,mu=1/50,
    rho=0.6,
    S0=8/60,I0=0.002,R0=1-8/60-0.001
  ),
seed=677573454L
) -> sir
```

Notice that we are assuming here that the data are collected weekly and use an Euler step-size of 1/20 wk. Here, we've assumed an infection with an infectious period of $1/\gamma = 1/8$ yr and a basic reproductive number, R_0 of $\beta/(\gamma+\mu) \approx 7.5$. We've assumed a host population size of 50,000 and 60% reporting efficiency. Fig. 8 shows one realization of this process.

Complications: seasonality, imported infections, extra-demographic stochasticity. Let's add a bit of real-world complexity to the simple SIR model. We'll modify the model to take three facts into account: (i) For many infections, the contact rate is *seasonal*: $\beta = \beta(t)$ is a periodic function of time. (ii) No host population is truly closed: *imported infections* arise when infected individuals visit the host population and transmit. (iii) Stochastic fluctuation in the rates λ, μ, and γ can give rise to *extrademographic stochasticity*, i.e., random process variability beyond the purely demographic stochasticity we've included so far.

One way to incorporate seasonality into the model is to assume some functional form for $\beta(t)$. Alternatively, we can use flexible functions to allow β to take a variety of shapes. B-splines are useful in this regard and pomp provides some simple facilities for using these. If $s_i(t)$, $i = 1,\dots,k$ is a periodic B-spline basis, as in Fig. 9, then we can for example define

$$\beta(t) = \sum_i b_i\, s_i(t)$$

and, by varying the coefficients b_i, we can obtain a wide variety of shapes for $\beta(t)$. In pomp, we can define a set of periodic B-spline basis functions by doing:

```
tbasis <- seq(0,20,by=1/52)
basis <- periodic.bspline.basis(tbasis,nbasis=3,degree=2,period=1,names="seas%d")
```

This results in a data-frame with 3 columns; each column is a quadratic periodic B-spline over the 20 yr domain, with period 1 yr. Fig. 9 shows these basis functions. Effectively, `tbasis` and `basis` function as a look-up table that can be used by the `rprocess` simulator to obtain a seasonal contact rate, $\beta(t)$. We accomplish this using the `covar` and `tcovar` arguments to pomp, as we will see below.

There are a number of ways to take account of imported infections. Here, we'll simply assume that there is some background force of infection, ι, not due to I-class individuals. Putting this together with the seasonal contact rate results in a force of infection $\lambda(t) = \beta(t)\, I(t)/N + \iota$.

Finally, we can allow for extrademographic stochasticity by allowing the force of infection to be itself a random variable. We'll accomplish this by assuming a multiplicative white noise on the force of infection,

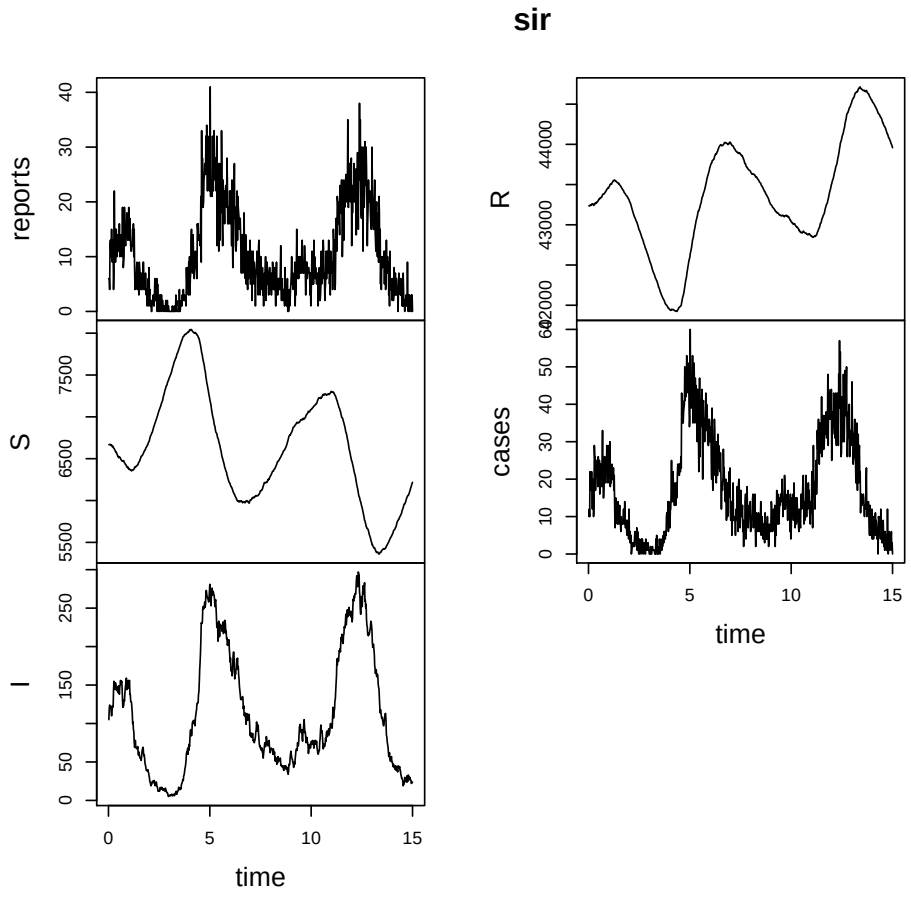

FIGURE 8. Results of `plot(sir)`.

i.e.,

$$\lambda(t) = \left(\beta(t)\,\frac{I(t)}{N} + \iota\right)\,\frac{dW(t)}{dt},$$

where dW/dt is a white noise, specifically the "derivative" of an integrated Gamma white noise process. He *et al.* (2010) discuss such processes and apply them in an inferential context; Bretó and Ionides (2011) develop the theory of infinitesimally overdispersed processes.

Let's modify the process-model simulator to incorporate these three complexities.

```
complex.sir.proc.sim <- function (x, t, params, delta.t, covars, ...) {
  ## unpack the parameters
  N <- params["N"]                  # population size
  gamma <- params["gamma"]          # recovery rate
  mu <- params["mu"]                # birth rate = death rate
  iota <- params["iota"]            # import rate
  b <- params[c("b1","b2","b3")]    # contact-rate coefficients
  beta <- b%*%covars                # flexible seasonality
  beta.sd <- params["beta.sd"]      # extrademographic noise intensity
  dW <- rgammawn(n=1,sigma=beta.sd,dt=delta.t) # Gamma white noise
```

```
  foi <- (beta*x["I"]/N+iota)*dW/delta.t # the force of infection
  trans <- c(
            rpois(n=1,lambda=mu*N*delta.t), # births are assumed to be Poisson
            reulermultinom(n=1,size=x["S"],rate=c(foi,mu),dt=delta.t), # exits from S
            reulermultinom(n=1,size=x["I"],rate=c(gamma,mu),dt=delta.t), # exits from I
            reulermultinom(n=1,size=x["R"],rate=c(mu),dt=delta.t)        # exits from R
            )
  ## now connect the compartments
  x[c("S","I","R","cases","W")]+
    c(
      trans[1]-trans[2]-trans[3],
      trans[2]-trans[4]-trans[5],
      trans[4]-trans[6],
      trans[4],                   # accumulate the recoveries
      (dW-delta.t)/beta.sd        # mean = 0, var = delta.t
    )
}
simulate(
         pomp(
              sir,
              tcovar=tbasis,
              covar=basis,
              rprocess=euler.sim(
                complex.sir.proc.sim,
                delta.t=1/52/20
                ),
              initializer=function(params, t0, ic.pars, comp.names, ...){
                x0 <- c(S=0,I=0,R=0,cases=0,W=0)
                N <- params["N"]
                fracs <- params[ic.pars]
                x0[comp.names] <- round(N*fracs/sum(fracs))
                x0
              }
              ),
         params=c(
             N=50000,
             b1=60,b2=10,b3=110,
             gamma=8,mu=1/50,
             rho=0.6,
             iota=0.01,beta.sd=0.1,
             S0=8/60,I0=0.002,R0=1-8/60-0.001
           ),
         seed=8274355L
         ) -> complex.sir
```

Note that the seasonal basis functions are passed to `complex.sir.proc.sim` via the `covars` argument. Whenever `complex.sir.proc.sim` is called, this argument will contain values of the covariates obtained from the look-up table.

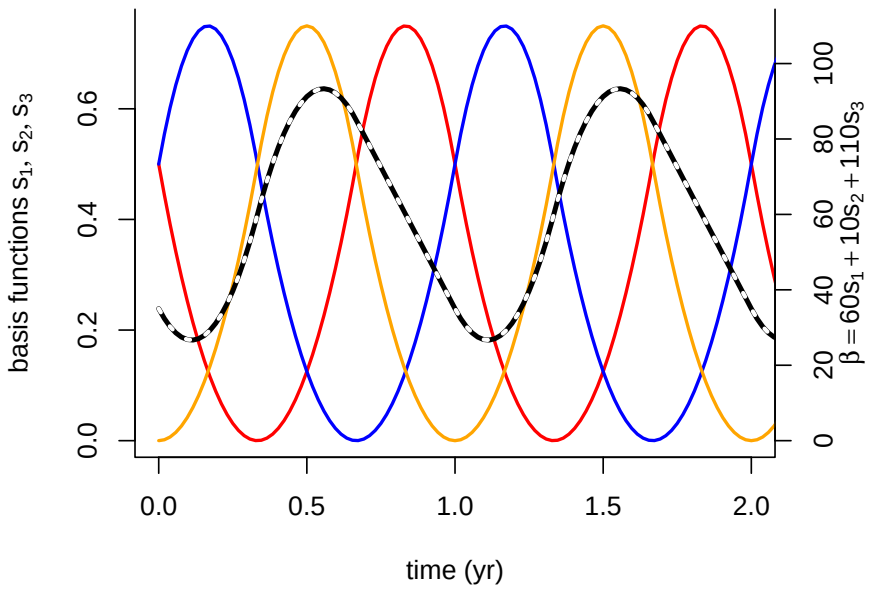

FIGURE 9. Periodic B-spline basis functions can be used to construct flexible periodic functions. The colored lines show the three basis functions, s_1, s_2, s_3. The dashed black line shows the seasonal transmission $\beta(t)$ assumed in `complex.sir`.

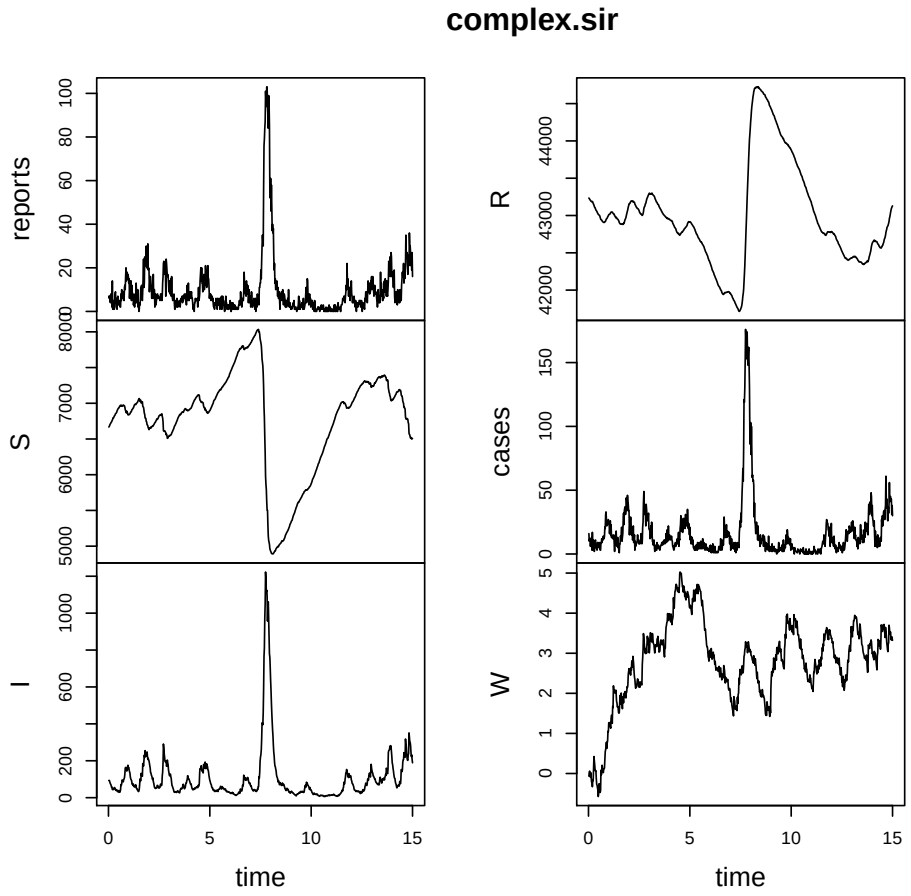

FIGURE 10. One realization of the SIR model with seasonal contact rate, imported infections, and extrademographic stochasticity in the force of infection.

References

C. Andrieu, A. Doucet, and R. Holenstein. Particle Markov chain Monte Carlo methods. *Journal of the Royal Statistical Society, Series B* **72**:269–342 (2010).

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 and E. L. Ionides. Compound markov counting processes and their applications to modeling infinitesimally over-dispersed systems. *Stochastic Processes and their Applications* **121**:2571–2591 (2011).

A. Davison and D. Hinkley. *Bootstrap Methods and their Application* (Cambridge University Press, New York, 1997).

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

C. Gouriéroux and A. Monfort. *Simulation-based Econometric Methods* (Oxford University Press, 1996).

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).

E. L. Ionides, A. Bhadra, Y. Atchadé, and A. A. King. Iterated filtering. *Annals of Statistics* **39**:1776–1802 (2011).

E. L. Ionides, C. Bretó, and A. A. King. Inference for nonlinear dynamical systems. *Proceedings of the National Academy of Sciences of the U.S.A.* **103**:18438–18443 (2006).

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).

B. E. Kendall, S. P. Ellner, E. McCauley, S. N. Wood, C. J. Briggs, W. M. Murdoch, and P. Turchin. Population cycles in the pine looper moth: Dynamical tests of mechanistic hypotheses. *Ecological Monographs* **75**:259–276 (2005).

J. Liu and M. West. Combining Parameter and State Estimation in Simulation-Based Filtering. In A. Doucet, N. de Freitas, and N. J. Gordon, eds., *Sequential Monte Carlo Methods in Practice*, pp. 197–224 (Springer, New York, 2001).

A. A. Smith. Estimating nonlinear time-series models using simulated vector autoregression. *Journal of Applied Econometrics* **8**:S63–S84 (1993).

C. W. Tidd, L. F. Olsen, and W. M. Schaffer. The Case for Chaos in Childhood Epidemics. II. Predicting Historical Epidemics from Mathematical Models. *Proceedings of the Royal Society of London, Series B* **254**:257–273 (1993).

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