

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	1
2. A first example: a simple discrete-time population model.	3
3. Defining a partially observed Markov process in <code>pomp</code> .	3
4. Simulating the model: <code>simulate</code>	6
5. Computing likelihood using particle filtering: <code>pfilter</code>	7
6. Interlude: utility functions for extracting and changing pieces of a <code>pomp</code> object	10
7. Transforming parameters	11
8. Estimating parameters using iterated filtering: <code>mif</code>	13
9. Trajectory matching: <code>traj.match</code>	15
10. Probe matching: <code>probe.match</code>	17
11. Nonlinear forecasting: <code>nlf</code>	21
12. Bayesian sequential Monte Carlo: <code>bsmc</code>	21
13. Particle Markov chain Monte Carlo: <code>pmcmc</code>	21
14. A more complex example: a seasonal epidemic model	22
References	26

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 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, \dots, t_n , the Markov property allows us to write

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

for each $k = 1, \dots, n$, 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

$$Y_t \sim f(X_t, \theta) \quad (2)$$

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 $X_{t_k} \rightarrow X_{t_{k+1}}$, 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 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

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

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

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

where $\sigma^2 = \text{Var}[\log \varepsilon_t]$. 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:

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

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. For now, let’s 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. 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 ($t \rightarrow t + \Delta t$) 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)
```

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

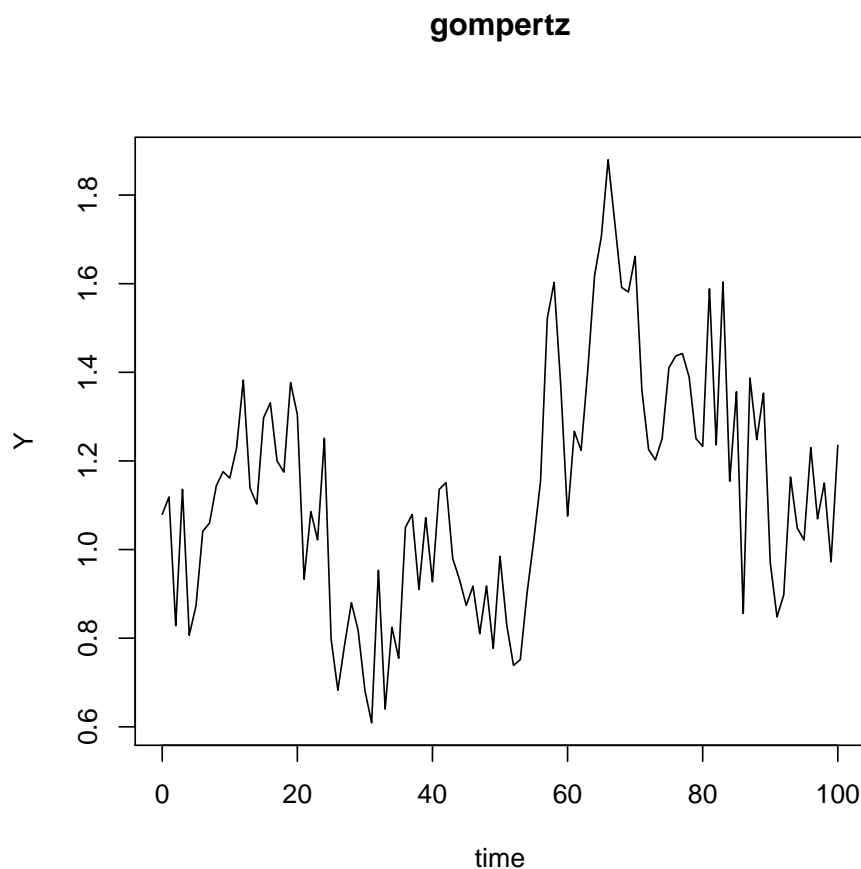


FIGURE 1. Simulated data and unobserved states from the Gompertz model (Eqs. 3–5). This figure shows the output of the command `plot(gompertz,variables="Y")`.

```
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
)
coef(gompertz) <- theta
```

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)
```

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

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

7. TRANSFORMING PARAMETERS

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. The following codes re-implement the Gompertz model using transformed parameters.

```
gompertz.proc.sim <- function (x, t, params, delta.t, ...) {
  ## unpack and untransform the parameters:
  r <- exp(params["log.r"])
  K <- exp(params["log.K"])
  sigma <- exp(params["log.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)
}

gompertz.meas.sim <- function (x, t, params, ...) {
  ## unpack and untransform the parameters:
  tau <- exp(params["log.tau"])
  ## state at time t:
  X <- x["X"]
  ## generate a simulated observation:
  y <- c(Y=unname(rlnorm(n=1,meanlog=log(X),sdlog=tau)))
  return(y)
}

gompertz.meas.dens <- function (y, x, t, params, log, ...) {
  ## unpack and untransform the parameters:
  tau <- exp(params["log.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)
}
```

Note that, in each of the above functions, we *untransform* the parameters before we do any computations.

```
dat <- as(gompertz,"data.frame")
theta <- c(
  log(coef(gompertz,c("r","K","tau","sigma"))),
  coef(gompertz,"X.0")
)
names(theta) <- c("log.r","log.K","log.tau","log.sigma","X.0")
```

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

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 `data(gompertz)`. For your convenience, codes creating this `pomp` object are included with the package. To view them, do:

```
file.show(system.file("examples/gompertz.R", package="pomp"))  
file.show(system.file("examples/gompertz.c", package="pomp"))
```

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.* (in press).

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 :

```
estpars <- c("log.r","log.sigma","log.tau")
mf <- replicate(
  n=10,
  {
    theta.guess <- theta.true
    theta.guess[estpars] <- rnorm(
      n=length(estpars),
      mean=theta.guess[estpars],
      sd=1
    )
    mif(
      gompertz,
      Nmif=100,
      start=theta.guess,
      pars=estpars,
      rw.sd=c(
        log.r=0.02,log.sigma=0.02,log.tau=0.05
      ),
      Np=2000,
      var.factor=4,
      ic.lag=10,
      cooling.factor=0.999,
      max.fail=10
    )
  }
)
##mf <- lapply(mf,continue,Nmif=50)
```

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.

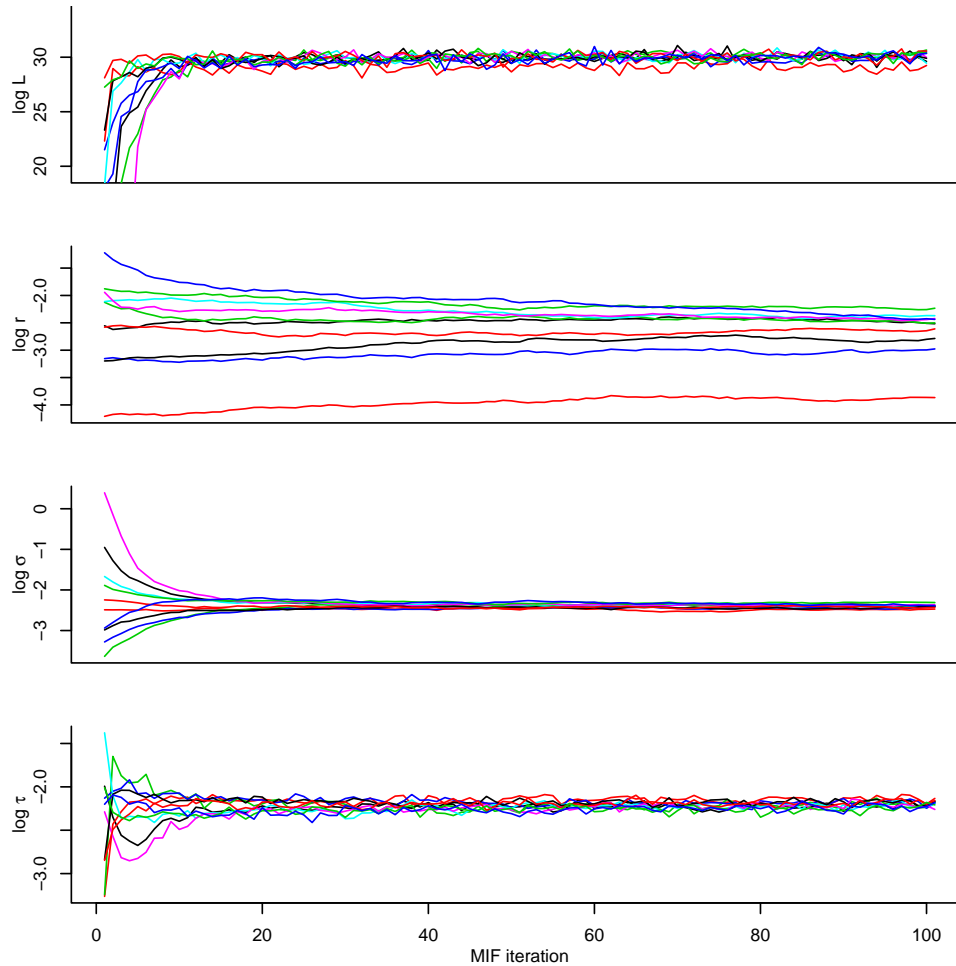


FIGURE 2. Convergence plots can be used to help diagnose convergence of the iterated filtering algorithm. This shows part of the output of `compare.mif(mf)`.

We'll evaluate the likelihood several times at this estimate to get an idea of the Monte Carlo error in our likelihood estimate.

```
theta.mif <- apply(sapply(mf,coef),1,mean)
loglik.mif <- replicate(n=10,logLik(pfilter(mf[[1]],params=theta.mif,Np=10000)))
bl <- mean(loglik.mif)
loglik.mif.est <- bl+log(mean(exp(loglik.mif-bl)))
loglik.mif.se <- sd(exp(loglik.mif-bl))/sqrt(length(loglik.mif))/exp(loglik.mif.est-bl)
loglik.true <- replicate(n=10,logLik(pfilter(gompertz,params=theta.true,Np=10000)))
loglik.true.est <- bl+log(mean(exp(loglik.true-bl)))
loglik.true.se <- sd(exp(loglik.true-bl))/sqrt(length(loglik.true))/exp(loglik.true.est-bl)
```

	mle	truth
log.r	-2.670	-2.300
log.sigma	-2.400	-2.300
log.tau	-2.180	-2.300
loglik	31.700	31.500
loglik.se	0.027	0.048

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

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

In the continuous-time case, this is the vectorfield

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

Our discrete-time Gompertz has the deterministic skeleton

$$x \mapsto K^{1-S} x^S, \tag{6}$$

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

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

```
new.gompertz <- pomp(
  data=data.frame(time=1:200,Y=NA),
  times="time",
  rprocess=discrete.time.sim(gompertz.proc.sim),
  rmeasure=gompertz.meas.sim,
  dmeasure=gompertz.meas.dens,
  skeleton.type="map",
  skeleton=gompertz.skel,
  t0=0
)
coef(new.gompertz) <- theta
coef(new.gompertz,"X.0") <- 0.1
coef(new.gompertz,"log.r") <- log(0.1)
coef(new.gompertz,"log.tau") <- log(0.05)
coef(new.gompertz,"log.sigma") <- log(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. Fig. 3 shows the results of this fit.

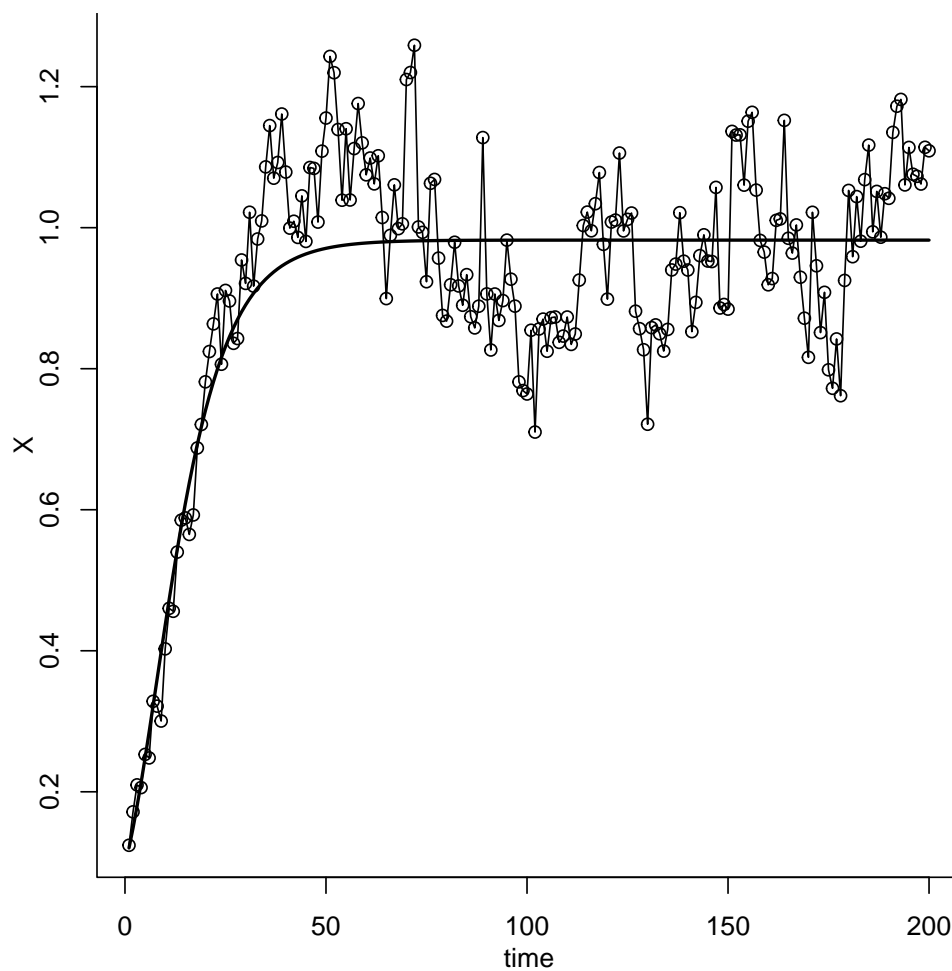


FIGURE 3. Illustration of trajectory matching. The points show data simulated from `new.gompertz`. The solid line shows the trajectory of the best-fitting model, obtained using `traj.match`. Fitting by trajectory matching is tantamount to the assumption that the data-generating process has no process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

```
tm <- traj.match(
  new.gompertz,
  start=coef(new.gompertz),
  est=c("log.r", "log.K", "log.tau", "X.0"),
  method="Nelder-Mead",
  maxit=1000,
  reltol=1e-8
)
```


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

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

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=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,
```

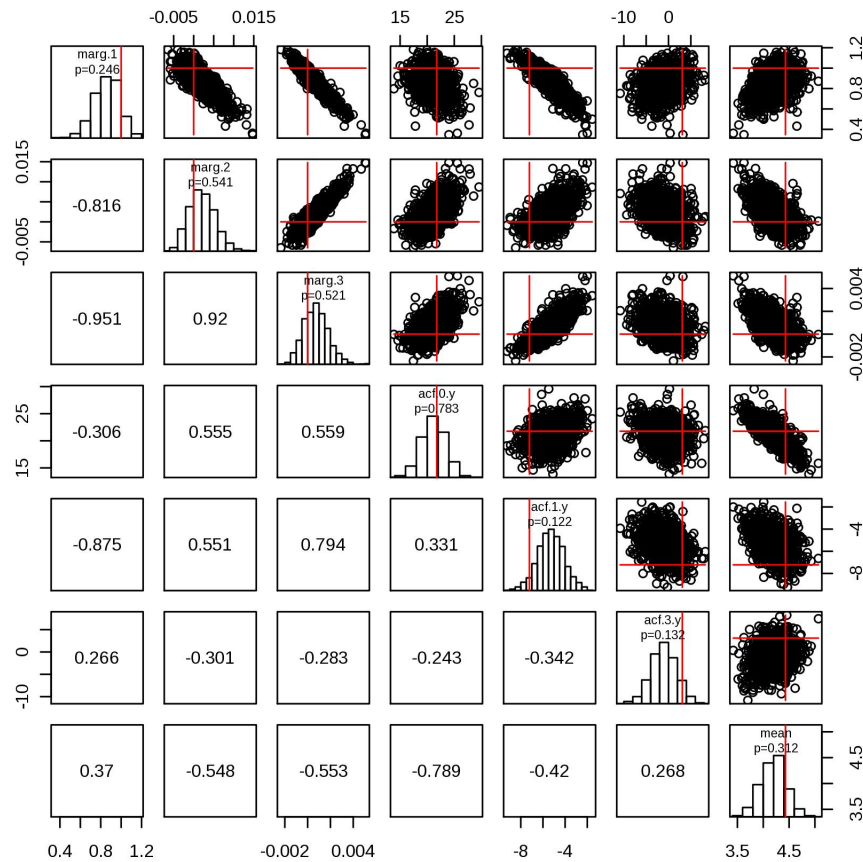


FIGURE 4. Results of `plot` on a `probed.pomp`-class object. Above the diagonal, the pairwise scatterplots show the values of the probes on each of 1000 data sets. The red lines show the values of each of the probes on the data. The panels along the diagonal show the distributions of the probes on the 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	-1.66	-1.12
log.phi	3.00	2.30	2.34	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	-137.12	-143.33
synth.loglik	-12.28	17.47	18.33	20.34

11. NONLINEAR FORECASTING: **NLF**

Ellner *et al.* (1998); Kendall *et al.* (1999, 2005). To be added.

12. BAYESIAN SEQUENTIAL MONTE CARLO: **BSMC**

Liu and West (2001). To be added.

13. PARTICLE MARKOV CHAIN MONTE CARLO: **PMCMC**

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

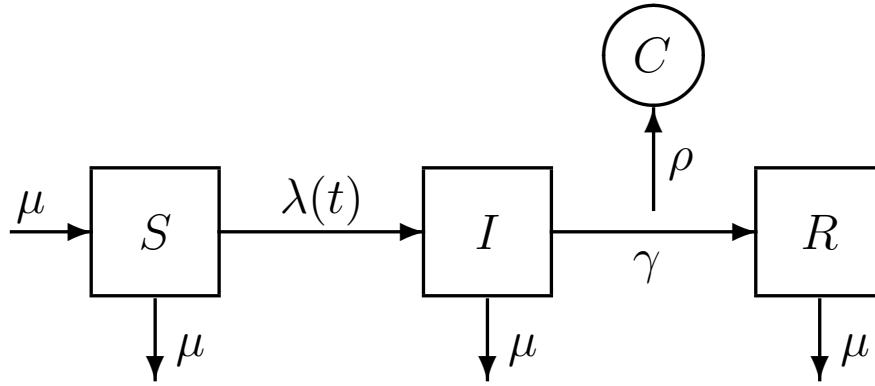


FIGURE 5. Diagram of the SIR model. The host population is divided into three classes according to their infection status: S, susceptible hosts; I, infected (and infectious) hosts; R, recovered and immune hosts. Births result in new susceptibles and all individuals have a common death rate μ . Since 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 $\lambda(t) = \beta(t) \xi(t) I/N$, where $\beta(t)$ is the seasonal (periodic) contact rate and $\xi(t)$ is a white noise. The I→R, or recovery, rate is γ . The case reports, C , count infected individuals 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 SIR model is a mainstay of theoretical epidemiology. It has the deterministic skeleton

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

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(frac))
  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*, ρ , 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 $I \rightarrow R$ 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)

```

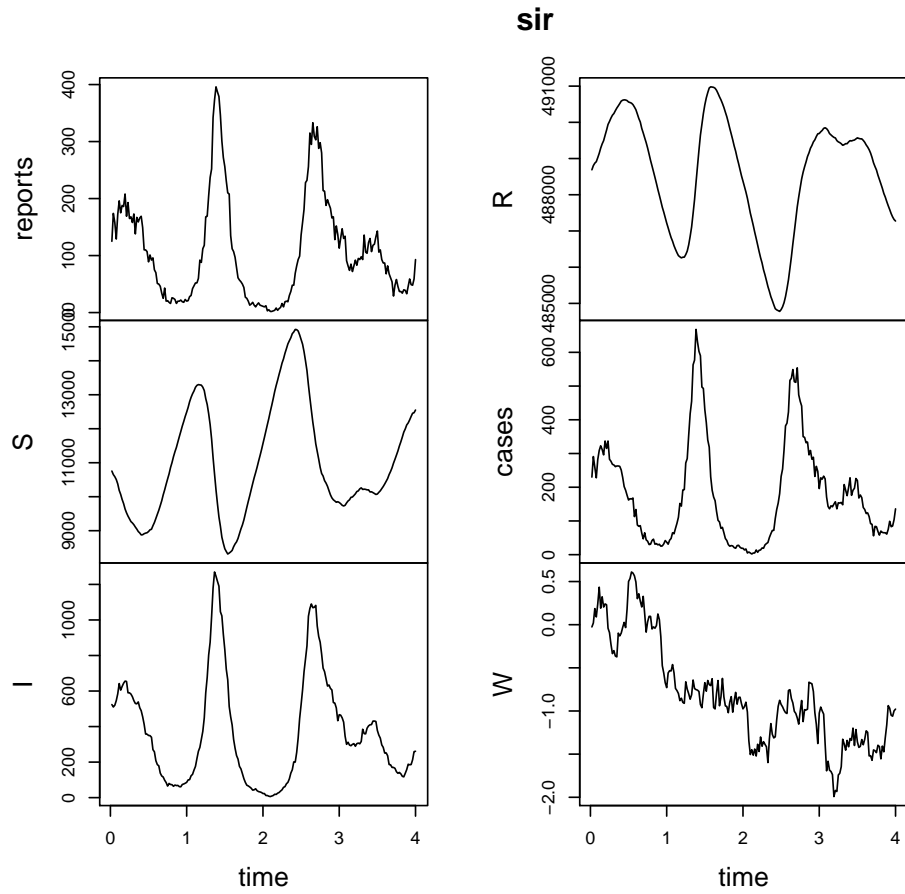

FIGURE 6. Results of `plot(sir)`.

Figure 6 shows the simulated data and state variable trajectories.

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, D. He, E. L. Ionides, and A. A. King. Time series analysis via mechanistic models. *Annals of Applied Statistics* **3**:319–348 (2009).
- S. P. Ellner, B. A. Bailey, G. V. Bobashev, A. R. Gallant, B. T. Grenfell, and D. W. Nychka. Noise and nonlinearity in measles epidemics: Combining mechanistic and statistical approaches to population modeling. *American Naturalist* **151**:425–440 (1998).
- 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).
- E. L. Ionides, A. Bhadra, Y. Atchadé, and A. A. King. Iterated filtering. *Annals of Statistics* (in press).
- 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).
- 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>