Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *http://www.jstatsoft.org/*

Statistical Inference for Partially Observed Markov Processes via the R Package pomp

Aaron A. King, Dao Nguyen, Edward L. Ionides
University of Michigan
December 13, 2014

Abstract

Partially observed Markov process (POMP) models, also known as hidden Markov models or state space models, are ubiquitous tools for time series analysis. The R package **pomp** provides a very flexible framework for Monte Carlo statistical investigations using nonlinear, non-Gaussian POMP models. A range of modern statistical methods for POMP models have been implemented in this framework including sequential Monte Carlo, iterated filtering, particle Markov chain Monte Carlo, approximate Bayesian computation, maximum synthetic likelihood estimation, nonlinear forecasting, and trajectory matching. In this paper, we demonstrate the application of these methodologies using some simple toy problems. We also illustrate the specification of more complex POMP models, using a nonlinear epidemiological model with a discrete population, seasonality, and extra-demographic stochasticity. We discuss the specification of user-defined models and the development of additional methods within the programming environment provided by **pomp**.

Keywords: Markov processes, hidden Markov model, state space model, stochastic dynamical system, maximum likelihood, plug-and-play, time series, mechanistic model, sequential Monte Carlo, R.

1. Introduction

A partially observed Markov process (POMP) model consists of incomplete and noisy measurements of a latent, unobserved Markov process. The far-reaching applicability of this class of models has motivated much software development (Commandeur *et al.* 2011). With the exception of models with a linear, Gaussian structure, or models for which the latent process takes values in a small discrete set, statistical inference typically involves Monte Carlo computations. It has been a challenge to provide a software environment that can effectively

handle broad classes of nonlinear POMP models and take advantage of the wide range of proposed statistical methodologies. The **pomp** software package (King *et al.* 2014) differs from previous approaches by providing a general and abstract representation of a POMP model. Algorithms implemented within **pomp** are necessarily applicable to arbitrary POMP models. Models formulated with **pomp** can be analyzed using multiple methodologies in search of the most effective method, or combination of methods, for the problem at hand.

A POMP model may be characterized by the transition density for the Markov process and the measurement density. However, some methods require only simulation from the transition density whereas others require evaluation of this density. Still other methods may not work with the model itself but with an approximation, such as a linearization. Algorithms for which the dynamic model is specified only via a simulator are said to be *plug-and-play* (Bretó *et al.* 2009; He *et al.* 2010). Plug-and-play methods can be implemented by "plugging" a model simulator into the inference machinery. Many scientific POMP models are relatively easy to simulate, and so the plug-and-play property facilitates data analysis. Even if one candidate model has tractable transition probabilities, a scientist will frequently wish to consider alternative models for which these probabilities are intractable. In a plug-and-play methodological environment, analysis of variations in the model can be readily implemented by changing a few lines of the model simulator codes. The price one pays for the flexibility of plug-and-play methodology is primarily additional computational effort. However, plug-and-play methods implemented using **pomp** have proved capable for state of the art inference problems (e.g., King *et al.* 2008; Bhadra *et al.* 2011; Shrestha *et al.* 2011, 2013; Earn *et al.* 2012; Roy *et al.* 2012; Blackwood *et al.* 2013a,b; He *et al.* 2013; Bretó 2014). The recent surge of interest in plug-and-play methodology for POMP models includes the development of nonlinear forecasting (Ellner *et al.* 1998), iterated filtering (Ionides *et al.* 2006), ensemble Kalman filtering (Shaman and Karspeck 2012), approximate Bayesian computation (ABC) (Sisson *et al.* 2007), particle Markov chain Monte Carlo (PMCMC) (Andrieu *et al.* 2010), probe matching (Kendall *et al.* 1999), and synthetic likelihood (Wood 2010). Although the **pomp** package provides a general environment for methods with and without the plug-and-play property, development of the package to date has emphasized plug-and-play methods.

The **pomp** package is philosophically neutral as to the merits of Bayesian inference. It enables a POMP model to be supplemented with prior distributions on parameters, and several Bayesian methods are implemented within the package. Thus **pomp** is a convenient environment for those who wish to explore both Bayesian and non-Bayesian data analyses.

The remainder of this paper is organized as follows. Section 2 defines mathematical notation for POMP models and relates this to their representation as objects of class '`pomp`' in the **pomp** package. Section 3 introduces several of the statistical methods currently implemented in **pomp**. Section 4 constructs and explores a simple POMP model, demonstrating the use of the available statistical methods. Section 5 illustrates the implementation of more complex POMPs, using a model of infectious disease transmission as an example. Finally, section 6 discusses extensions and applications of **pomp**.

2. POMP models and their representation in pomp

Let θ be a p-dimensional real-valued parameter, $\theta \in \mathbb{R}^p$. For each value of θ, let $\{X(t;\theta), t \in T\}$ be a Markov process, with $X(t;\theta)$ taking values in $\mathbb{R}^q$. The time index set $T \subset \mathbb{R}$ may

Method	Argument to the `pomp` constructor	Mathematical terminology
`rprocess`	`rprocess`	Simulate from $f_{X_n\|X_{n-1}}(x_n \,\|\, x_{n-1}\,;\theta)$
`dprocess`	`dprocess`	Evaluate $f_{X_n\|X_{n-1}}(x_n \,\|\, x_{n-1}\,;\theta)$
`rmeasure`	`rmeasure`	Simulate from $f_{Y_n\|X_n}(y_n \,\|\, x_n\,;\theta)$
`dmeasure`	`dmeasure`	Evaluate $f_{Y_n\|X_n}(y_n \,\|\, x_n\,;\theta)$
`rprior`	`rprior`	Simulate from the prior distribution $\pi(\theta)$
`dprior`	`dprior`	Evaluate the prior density $\pi(\theta)$
`init.state`	`initializer`	Simulate from $f_{X_0}(x_0\,;\theta)$
`timezero`	`t0`	t_0
`time`	`times`	$t_{1:N}$
`obs`	`data`	$y^*_{1:N}$
`states`	—	$x_{0:N}$
`coef`	`params`	θ

Table 1: Constituent methods for class-'`pomp`' objects and their translation into mathematical notation for POMP models. For example, the `rprocess` method is set using the `rprocess` argument to the `pomp` constructor function.

be an interval or a discrete set. Let $t_i \in T$, $i = 1, \ldots, N$, be the times at which $X(t;\theta)$ is observed, and $t_0 \in T$ be an initial time. Assume $t_0 \le t_1 < t_2 < \cdots < t_N$. We write $X_i = X(t_i\,;\theta)$ and $X_{i:j} = (X_i,\ X_{i+1}, \ldots, X_j)$. The process $X_{0:N}$ is only observed by way of another process $Y_{1:N} = (Y_1, \ldots, Y_N)$ with Y_n taking values in $\mathbb{R}^r$. The observable random variables $Y_{1:N}$ are assumed to be conditionally independent given $X_{0:N}$. The data, $y^*_{1:N} = (y^*_1,\ \ldots,\ y^*_N)$, are modeled as a realization of this observation process and are considered fixed. We suppose that $X_{0:N}$ and $Y_{1:N}$ have a joint density $f_{X_{0:N},Y_{1:N}}(x_{0:n},\ y_{1:n}\,;\theta)$. The POMP structure implies that this joint density is characterized by the initial density, $f_{X_0}(x_0;\theta)$, together with the conditional transition probability density, $f_{X_n|X_{n-1}}(x_n \,|\, x_{n-1}\,;\theta)$, and the measurement density, $f_{Y_n|X_n}(y_n \,|\, x_n\,;\theta)$, for $1 \le n \le N$. Note that this formalism allows the transition density, $f_{X_n|X_{n-1}}$, and measurement density, $f_{Y_n|X_n}$, to depend explicitly on n.

2.1. Implementation of POMP models

pomp is fully object-oriented: in the package, a POMP model is represented by an S4 object (Chambers 1998; Genolini 2008) of class '`pomp`'. Slots in this object encode the components of the POMP model, and can be filled or changed using the constructor function `pomp` and various other convenience functions. Methods for the '`pomp`' class use these components to carry out computations on the model. Table 1 gives the mathematical notation corresponding to the elementary methods that can be executed on a class-'`pomp`' object.

The `rprocess`, `dprocess`, `rmeasure`, and `dmeasure` arguments specify the transition probabilies $f_{X_n|X_{n-1}}(x_n \,|\, x_{n-1}\,;\theta)$ and measurement densities $f_{Y|X}(y \,|\, x\,;\theta)$. Not all of these need to be defined to implement any specific computation. In particular, plug-and-play methodology by definition never uses `dprocess`. An empty *dprocess* slot in a class-'`pomp`' object is therefore acceptable unless a non-plug-and-play algorithm is attempted. In the package, the data and corresponding measurement times are considered necessary parts of a '`pomp`' object whilst specific values of the parameters and latent states are not. Applying the `simulate` function

to an object of class 'pomp' returns another object of class 'pomp', within which the data $y^*_{1:N}$ have been replaced by a stochastic realization of $Y_{1:N}$, the corresponding realization of $X_{0:N}$ is accessible via the `states` method, and the `params` slot has been filled with the value of θ used in the simulation.

To illustrate the specification of models in **pomp** and the use of the package's inference algorithms, we'll use a simple example. The Gompertz (1825) model can be constructed via

```
R> library("pomp")
R> pompExample(gompertz)
```

which results in the creation of an object of class 'pomp', named `gompertz`, in the workspace. The structure of this model and its implementation in **pomp** is described below, in section 4. One can view the components of `gompertz` listed in Table 1 by executing

```
R> obs(gompertz)
R> states(gompertz)
R> as.data.frame(gompertz)
R> plot(gompertz)
R> time(gompertz)
R> coef(gompertz)
R> init.state(gompertz)
```

Executing `pompExamples()` lists other examples provided with the package.

2.2. Initial conditions

In some experimental situations, $f_{X_0}(x_0\,;\theta)$ corresponds to a known experimental initialization, but in general the initial state of the latent process will have to be inferred. If the transition density for the dynamic model, $f_{X_n|X_{n-1}}(x_n\,|\,x_{n-1}\,;\theta)$, does not depend on time and possesses a unique stationary distribution, it may be natural to set $f_{X_0}(x_0\,;\theta)$ to be this stationary distribution. Otherwise, and more commonly in the authors' experience, no clear scientifically motivated choice of $f_{X_0}(x_0\,;\theta)$ exists and one can proceed by treating the value of X_0 as a parameter to be estimated. In this case, $f_{X_0}(x_0\,;\theta)$ concentrates at a point, the location of which depends on θ.

2.3. Covariates

Scientifically, one may be interested in the role of a vector-valued covariate process $\{Z(t)\}$ in explaining the data. Modeling and inference conditional on $\{Z(t)\}$ can be carried out within the general framework for nonhomogeneous POMP models, since the arbitrary densities $f_{X_n|X_{n-1}}$, f_{X_0} and $f_{Y_n|X_n}$ can depend on the observed process $\{Z(t)\}$. For example, it may be the case that $f_{X_n|X_n-1}(x_n\,|\,x_{n-1}\,;\theta)$ depends on n only through $Z(t)$ for $t_{n-1} \le t \le t_n$. The `covar` argument in the **pomp** constructor allows for time-varying covariates measured at times specified in the `tcovar` argument. A example using covariates is given in section 5.

3. Methodology for POMP models

Data analysis typically involves identifying regions of parameter space within which a postulated model is statistically consistent with the data. Additionally, one frequently desires to assess the relative merits of alternative models as explanations of the data. Once the user has encoded one or more POMP models as objects of class 'pomp', **pomp** provides a variety of algorithms to assist with these data analysis goals. Table 2 provides an overview of several inference methodologies for POMP models. Each method may be categorized as full-information or feature-based, Bayesian or Frequentist, and plug-and-play or not plug-and-play.

Approaches that work with the full likelihood function, whether in a Bayesian or frequentist context, can be called full-information methods. Since low-dimensional sufficient statistics are not generally available for POMP models, methods which take advantage of favorable low-dimensional representations of the data typically lose some statistical efficiency. We use the term "feature-based" to describe all methods not based on the full likelihood, since such methods statistically emphasize some features of the data over others.

Many Monte Carlo methods of inference can be viewed as algorithms for the exploration of high-dimensional surfaces. This view obtains whether the surface in question is the likelihood surface or that of some other objective function. The premise behind many recent methodological developments in Monte Carlo methods for POMP models is that generic stochastic numerical analysis tools, such as standard Markov chain Monte Carlo and Robbins-Monro type methods, are effective only on the simplest models. For many models of scientific interest, therefore, methods that leverage the POMP structure are needed. Though **pomp** has sufficient flexibility to encode arbitrary POMP models and methods and therefore also provides a platform for the development of novel POMP inference methodology, **pomp**'s development to date has focused on plug-and-play methods. In the remainder of this Section, we describe and discuss several such methods, all currently implemented in the package.

3.1. The likelihood function and sequential Monte Carlo

The log likelihood for a POMP model is $\ell(\theta) = \log f_{Y_{1:N}}(y^*_{1:N}\,;\theta)$, which can be written as a sum of conditional log likelihoods,

$$\ell(\theta) = \sum_{n=1}^{N} \ell_{n|1:n-1}(\theta), \tag{1}$$

where

$$\ell_{n|1:n-1}(\theta) = \log f_{Y_n|Y_{1:n-1}}(y^*_n \,|\, y^*_{1:n-1}\,;\theta),$$

and we use the convention that $y^*_{1:0}$ is an empty vector. The structure of a POMP model implies the representation

$$\ell_{n|1:n-1}(\theta) = \log \int f_{Y_n|X_n}(y^*_n|x_n\,;\theta) f_{X_n|Y_{1:n-1}}(x_n \,|\, y^*_{1:n-1}\,;\theta)\, dx_n. \tag{2}$$

Although $\ell(\theta)$ typically has no closed form, it can frequently be computed by Monte Carlo methods. Sequential Monte Carlo (SMC) builds up a representation of $f_{X_n|Y_{1:n-1}}(x_n \,|\, y^*_{1:n-1}\,;\theta)$ that can be used to obtain an estimate, $\hat{\ell}_{n|1:n-1}(\theta)$, of $\ell_{n|1:n-1}(\theta)$ and hence an approximation, $\hat{\ell}(\theta)$, to $\ell(\theta)$. SMC (a basic version of which is presented as Algorithm 1), is also known as the

(a) Plug-and-play

	Frequentist	Bayesian
Full information	Iterated filtering (`mif`, section 3.2)	PMCMC (`pmcmc`, section 3.3)
Feature-based	Nonlinear forecasting (`nlf`, section 3.6), synthetic likelihood (`probe.match`, section 3.4)	ABC (`abc`, section 3.5)

(b) Not plug-and-play

	Frequentist	Bayesian
Full information	EM and Monte Carlo EM, Kalman filter	MCMC
Feature-based	Trajectory matching (`traj.match`), extended Kalman filter, Yule-Walker equations	Extended Kalman filter

Table 2: Inference methods for POMP models. For those currently implemented in **pomp**, function name and a reference for description are provided in parentheses. Standard Expectation-Maximization (EM) and Markov chain Monte Carlo (MCMC) algorithms are not plug-and-play since they require evaluation of $f_{X_n|X_{n-1}}(x_n \,|\, x_{n-1}\,;\theta)$. The Kalman filter and extended Kalman filter are not plug-and-play since they cannot be implemented based on a model simulator. The Kalman filter provides the likelihood for a linear, Gaussian model. The extended Kalman filter employs a local linear Gaussian approximation which can be used for frequentist inference (via maximization of the resulting quasi-likelihood) or approximate Bayesian inference (by adding the parameters to the state vector). The Yule-Walker equations for ARMA models provide an example of a closed-form method of moments estimator.

particle filter, since it is conventional to describe the Monte Carlo sample, $\{X^F_{n,j}, j \text{ in } 1:J\}$ as a swarm of particles representing $f_{X_n|Y_{1:n}}(x_n \,|\, y^*_{1:n}\,;\theta)$. The swarm is propagated forward according to the dynamic model and then assimilated to the next data point. Using an evolutionary analogy, the prediction step (line 3) mutates the particles in the swarm and the filtering step (line 7) corresponds to selection. SMC is implemented in **pomp** in the `pfilter` function. The basic particle filter in Algorithm 1 possesses the plug-and-play property. Many variations and elaborations to SMC have been proposed; these may improve numerical performance in appropriate situations (Cappé *et al.* 2007) but typically lose the plug-and-play property.

Basic SMC methods fail when an observation is extremely unlikely given the POMP model. This leads to the situation that all particles are inconsistent with the observation, a phenomenon known as *particle depletion*. Many elaborations of the basic SMC algorithm have been proposed to ameliorate this problem. However, it is often preferable to remedy the situation by seeking a better model. The plug-and-play property assists in this process by facilitating investigation of alternative models.

In line 6 of Algorithm 1, systematic resampling (Algorithm 2) is used in preference to multi-

Algorithm 1: Sequential Monte Carlo (SMC, or particle filter): `pfilter(P, Np = J)`, using notation from Table 1 where `P` is a '`pomp`' object with defined methods for `rprocess`, `dmeasure`, `init.state`, `coef`, and `obs`.

input: Simulator for $f_{X_n|X_{n-1}}(x_n \,|\, x_{n-1}\,;\theta)$; evaluator for $f_{Y_n|X_n}(y_n \,|\, x_n\,;\theta)$; simulator for $f_{X_0}(x_0\,;\theta)$; parameter, θ; data, $y^*_{1:N}$; number of particles, J.

1 Initialize filter particles: simulate $X^F_{0,j} \sim f_{X_0}\,(x_0\,;\theta)$ for j in $1:J$.

2 **for** n *in* $1:N$ **do**

3 Simulate for prediction: $X^P_{n,j} \sim f_{X_n|X_{n-1}}\big(x_n|X^F_{n-1,j};\theta\big)$ for j in $1:J$.

4 Evaluate weights: $w(n,j) = f_{Y_n|X_n}(y^*_n|X^P_{n,j}\,;\theta)$ for j in $1:J$.

5 Normalize weights: $\tilde{w}(n,j) = w(n,j)/\sum_{m=1}^{J} w(n,m)$.

6 Apply Algorithm 2 to select indices $k_{1:J}$ with $\mathbb{P}\,\{k_j = m\} = \tilde{w}(n,m)$.

7 Resample: set $X^F_{n,j} = X^P_{n,k_j}$ for j in $1:J$.

8 Compute conditional log likelihood: $\hat{\ell}_{n|1:n-1} = \log\big(J^{-1}\,\sum_{m=1}^{J} w(n,m)\big)$.

9 **end**

output: Log likelihood estimate, $\hat{\ell}(\theta) = \sum_{n=1}^{N} \hat{\ell}_{n|1:n-1}$; filter sample, $X^F_{n,1:J}$, for n in $1:N$.

nomial resampling. Algorithm 2 minimizes Monte Carlo variability while resampling with the proper marginal probability. In particular, if all the particle weights are equal then Algorithm 2 has the appropriate behavior of leaving the particles unchanged.

3.2. Iterated filtering

Iterated filtering is a technique for maximizing the likelihood obtained by SMC. 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. Over multiple repetitions of the filtering procedure, the intensity of this random walk approaches zero and the modified model approaches the original model. Adding additional variability in this way has four positive effects:

A1. It smooths the likelihood surface, which facilitates optimization.

A2. It combats particle depletion, by dispersing the particles more widely.

A3. The additional variability can be exploited to estimate of the gradient of the (smoothed) likelihood surface based on the same filtering procedure that is required to estimate of the value of the likelihood.

A4. It preserves the plug-and-play property, inherited from the particle filter.

Iterated filtering is implemented in the `mif` function, as described in Algorithm 3, following Ionides *et al.* (2006). By analogy with annealing, the random walk intensity can be called a temperature, which is decreased at a prescribed cooling rate. One hopes that the algorithm will freeze at the maximum of the likelihood as the temperature approaches zero.

The perturbations on the parameters in lines 2 and 6 of Algorithm 3 follow a normal distribution, with each component of the parameter vector perturbed independently. Neither normality nor independence are necessary for iterated filtering, but, rather than varying the

Algorithm 2: Systematic resampling: Line 6 of Algorithm 1.

input: Weights, $\tilde{w}_{1:J}$, normalized so that $\sum_{j=1}^{J} \tilde{w}_j = 1$.

1 Construct cumulative sum: $c_j = \sum_{m=1}^{j} \tilde{w}_m$, for j in $1:J$.
2 Draw a uniform initial sampling point: $U_1 \sim \text{Uniform}[0, J^{-1}]$.
3 Construct evenly spaced sampling points: $U_j = U_1 + (j-1)J^{-1}$, for j in $2:J$.
4 Initialize: set $p = 1$.
5 **for** j *in* $1:J$ **do**
6 **while** $U_j > c_p$ **do**
7 Step to the next resampling index: set $p = p + 1$.
8 **end**
9 Assign resampling index: set $k_j = p$.
10 **end**

output: Resampling indices, $k_{1:J}$.

perturbation distribution, one can transform the parameters to make these simple algorithmic choices reasonable.

Algorithm 3 gives special treatment to a subset of the components of the parameter vector termed initial value parameters (IVPs), which arise when unknown initial conditions are modeled as parameters. These IVPs will typically be inconsistently estimable as the length of the time series increases, since for a stochastic process one expects only early time points to contain information about the initial state. Searching the parameter space using time-varying parameters is inefficient in this situation, and so Algorithm 3 perturbs these parameters only at time zero.

Lines 6–11 of Algorithm 3 are exactly an application of SMC (Algorithm 1) to a modified POMP model in which the parameters are added to the state space. This approach has been used in a variety of previously proposed POMP methodologies (Kitagawa 1998; Liu and West 2001; Wan and Van Der Merwe 2000) but iterated filtering is distinguished by having theoretical justification for convergence to the maximum likelihood estimate (Ionides *et al.* 2011).

3.3. Particle Markov chain Monte Carlo

Full-information plug-and-play Bayesian inference for POMP models is enabled by particle Markov chain Monte Carlo (PMCMC) algorithms (Andrieu *et al.* 2010). PMCMC methods combine likelihood evaluation via SMC with MCMC moves in the parameter space. The simplest and most widely used PMCMC algorithm, termed particle marginal Metropolis-Hastings (PMMH), is based on the observation that the unbiased likelihood estimate provided by SMC can be plugged in to the Metropolis-Hastings update procedure to give an algorithm targeting the desired posterior distribution for the parameters (Andrieu and Roberts 2009). PMMH is implemented in `pmcmc`, as described in Algorithm 4. PMCMC can have a high computational cost, since each SMC computation can itself be costly and tens of thousands of these SMC computations will typically be necessary; this is demonstrated in section 4.4. Nevertheless, its invention introduced the possibility of full-information plug-and-play Bayesian inferences in some situations where they were previously unavailable.

Algorithm 3: Iterated filtering: `mif(P, start=`θ_0`, Nmif=`M`, Np=`J`, rw.sd=`$\sigma_{1:p}$`, ic.lag=`L`, var.factor=`C`, cooling.factor=`a`)`, using notation from Table 1 where `P` is a '`pomp`' object with defined `rprocess`, `dmeasure`, `init.state` and `obs` methods.

input: Starting parameter, θ_0; simulator for $f_{X_0}(x_0;\theta)$; simulator for $f_{X_n|X_{n-1}}(x_n \,|\, x_{n-1};\theta)$; evaluator for $f_{Y_n|X_n}(y_n \,|\, x_n;\theta)$; data, $y^*_{1:N}$; labels, $I \subset \{1,\dots,p\}$, designating IVPs; fixed lag, L, for estimating IVPs; number of particles, J, number of iterations, M; cooling rate, $0 < a < 1$; perturbation scales, $\sigma_{1:p}$; initial scale multiplier, $C > 0$.

for m *in* $1{:}M$ **do**
  Initialize parameters: $[\Theta^F_{0,j}]_i \sim \text{Normal}\big[[\theta_0]_i, (Ca^{m-1}\sigma_i)^2\big]$ for i in $1{:}p$.
  Initialize states: simulate $X^F_{0,j} \sim f_{X_0}\big(x_0;\Theta^F_{0,j}\big)$ for j in $1{:}J$.
  Initialize filter mean for parameters: $\bar\theta_0 = \theta_0$.
  for n *in* $1:N$ **do**
    Perturb parameters: $\big[\Theta^P_{n,j}\big]_i \sim \text{Normal}\big(\big[\Theta^F_{n-1,j}\big]_i, (a^{m-1}\sigma_i)^2\big)$ for $i \notin I$, j in $1:J$.
    Simulate prediction particles: $X^P_{n,j} \sim f_{X_n|X_{n-1}}\big(x_n|X^F_{n-1,j};\Theta^P_{n,j}\big)$ for j in $1:J$.
    Evaluate weights: $w(n,j) = f_{Y_n|X_n}(y^*_n|X^P_{n,j};\Theta^P_{n,j})$ for j in $1:J$.
    Normalize weights: $\tilde w(n,j) = w(n,j)/\sum_{u=1}^J w(n,u)$.
    Apply Algorithm 2 to select indices $k_{1:J}$ with $\mathbb{P}\{k_u = j\} = \tilde w\,(n,j)$.
    Resample particles: $X^F_{n,j} = X^P_{n,k_j}$ and $\Theta^F_{n,j} = \Theta^P_{n,k_j}$ for j in $1:J$.
    Filter mean: $\big[\bar\theta_n\big]_i = \sum_{j=1}^J \tilde w(n,j)\big[\Theta^P_{n,j}\big]_i$ for $i \notin I$.
    Prediction variance: $[\bar V_{n+1}]_i = (a^{m-1}s_i)^2 + \sum_j \tilde w(n,j)\big([\Theta^F_{n,j}]_i - [\bar\theta_n]_i\big)^2$ for $i \notin I$.
  end
  Update non-IVP parameters: $[\theta_m]_i = [\theta_{m-1}]_i + V^i_1 \sum_{n=1}^N \big(V^i_n\big)^{-1}\big(\bar\theta^i_n - \bar\theta^i_{n-1}\big)$ for $i \notin I$.
  Update IVPs: $[\theta_m]_i = \frac{1}{J}\sum_j \big[\Theta^F_{L,j}\big]_i$ for $i \in I$.
end

output: Monte Carlo maximum likelihood estimate, θ_M.

3.4. Synthetic likelihood of summary statistics

Some motivations to estimate parameter based on features rather than the full likelihood include

B1. Reducing the data to sensibly selected and informative low-dimensional summary statistics may have computational advantages (Wood 2010).

B2. The scientific goal may be to match some chosen characteristics of the data rather than all aspects of it. Acknowledging the limitations of all models, this limited aspiration may be all that can reasonably be demanded (Kendall *et al.* 1999; Wood 2001).

B3. In conjunction with full-information methodology, consideration of individual features has diagnostic value to determine which aspects of the data are driving the full-information inferences (Reuman *et al.* 2006).

B4. Feature-based methods for dynamic models typically do not require the POMP model structure. However, that benefit is outside the scope of the **pomp** package.

Algorithm 4: Particle Markov Chain Monte Carlo: `pmcmc(P, start=`θ_0`, Nmcmc=`M`, Np=`J`, rw.sd=`$\sigma_{1:p}$`)`, using notation from Table 1 where `P` is a '`pomp`' object with defined methods for `rprocess`, `dmeasure`, `init.state`, `dprior`, and `obs`.

input: Starting parameter, θ_0; simulator for $f_{X_0}(x_0;\theta)$; simulator for $f_{X_n|X_{n-1}}(x_n \,|\, x_{n-1};\theta)$; evaluator for $f_{Y_n|X_n}(y_n \,|\, x_n;\theta)$; data, $y^*_{1:N}$; number of particles, J; number of filtering operations, M; perturbation scales, $\sigma_{1:p}$; evaluator for prior, $f_\Theta(\theta)$.

1 Initialization: compute $\hat\ell(\theta_0)$ using Algorithm 1 with J particles.

2 **for** m *in* $1:M$ **do**

3 Propose parameters: $[\theta^P_m]_i \sim \text{Normal}\left([\theta_{m-1}]_i, \sigma_i^2\right)$ for i in $1:p$.

4 Compute $\hat\ell(\theta^P_m)$ using Algorithm 1 with J particles.

5 Generate $U \sim \text{Uniform}[0,1]$.

6 Set $\big(\theta_m, \hat\ell(\theta_m)\big) = \begin{cases} \big(\theta^P_m, \hat\ell(\theta^P_m)\big), & \text{if } U < \dfrac{f_\Theta(\theta^P_m)\exp(\hat\ell(\theta^P_m))}{f_\Theta(\theta_{m-1})\exp(\hat\ell(\theta_{m-1}))}, \\ \big(\theta_{m-1}, \hat\ell(\theta_{m-1})\big), & \text{otherwise.} \end{cases}$

7 **end**

output: Samples, $\theta_{1:M}$, representing the posterior distribution, $f_{\Theta|Y_{1:N}}(\theta \,|\, y^*_{1:N})$.

B5. Feature-based methods are typically *doubly plug-and-play*, meaning that they require simulation, but not evaluation, for both the latent process transition density and the measurement model.

When pursuing goal B1, one aims to find summary statistics which are as close as possible to sufficient statistics for the unknown parameters. Goals B2 and B3 deliberately look for features which discard information from the data; in this context the features have been called probes (Kendall *et al.* 1999). The features are denoted by a collection of functions, $\mathbb{S} = (\mathbb{S}_1, \dots, \mathbb{S}_d)$, where each $\mathbb{S}_i$ maps an observed time series to a real number. We write $S = (S_1, \dots, S_d)$ for the vector-valued random variable with $S = \mathbb{S}(Y_{1:N})$, with $f_S(s;\theta)$ being the corresponding joint density. The observed feature vector is s^* where $s^*_i = \mathbb{S}_i(y^*_{1:N})$, and for any parameter set one can look for parameter values for which typical features for simulated data match the observed features. One can define a likelihood function, $\ell_{\mathbb{S}}(\theta) = f_S(s^*;\theta)$. Arguing that S should be approximately multivariate normal, for suitable choices of the features, Wood (2010) proposed using simulations to construct a multivariate normal approximation to $\ell_{\mathbb{S}}(\theta)$, and called this a *synthetic likelihood*.

Simulation-based evaluation of a feature matching criterion is implemented by `probe` (Algorithm 5). The feature matching criterion requires a scale, and a natural scale to use is the empirical covariance of the simulations. Working on this scale, as implemented by `probe`, there is no substantial difference between the probe approaches of Kendall *et al.* (1999) and Wood (2010). Numerical optimization of the synthetic likelihood is implemented by `probe.match`, which offers a choice of the subplex method (Rowan 1990; King 2008) or any method provided by `optim` or the **nloptr** package (Johnson 2014; Ypma 2014).

3.5. Approximate Bayesian computation (ABC)

ABC algorithms are Bayesian feature-matching techniques, comparable to the frequentist

Algorithm 5: Synthetic likelihood evaluation: `probe(P, nsim=`J`, probes=`$\mathbb{S}$`)`, using notation from Table 1 where `P` is a '`pomp`' object with defined methods for `rprocess`, `rmeasure`, `init.state`, and `obs`.

input: Simulator for $f_{X_n|X_{n-1}}(x_n \,|\, x_{n-1}\,;\theta)$; simulator for $f_{X_0}(x_0\,;\theta)$; simulator for $f_{Y_n|X_n}(y_n \,|\, x_n\,;\theta)$; parameter, θ; data, $y^*_{1:N}$; number of simulations, J; vector of summary statistics or *probes*, $\mathbb{S} = (\mathbb{S}_1, \ldots, \mathbb{S}_d)$.

1 Compute observed probes: $s^*_i = \mathbb{S}_i(y^*_{1:N})$ for i in $1{:}d$.
2 Simulate J datasets: $Y_{1:N,j} \sim f_{Y_{1:N}}(y_{1:N}\,;\theta)$ for j in $1{:}J$.
3 Compute simulated probes: $s_{ij} = \mathbb{S}_i(Y_{1:N,j})$ for i in $1:d$ and j in $1{:}J$.
4 Compute sample mean: $\mu_i = J^{-1}\sum_{j=1}^{J} s_{ij}$ for i in $1{:}d$.
5 Compute sample covariance: $\Sigma_{ik} = (J-1)^{-1}\sum_{j=1}^{J}(s_{ij}-\mu_i)(s_{kj}-\mu_k)$ for i and k in $1{:}d$.
6 Compute the log synthetic likelihood:

$$\hat{\ell}_{\mathbb{S}}(\theta) = -\frac{1}{2}\,(s^*-\mu)^T\Sigma^{-1}(s^*-\mu) - \frac{1}{2}\,\log|\Sigma| - \frac{d}{2}\,\log(2\pi).$$

output: Synthetic likelihood, $\hat{\ell}_{\mathbb{S}}(\theta)$.

Algorithm 6: Approximate Bayesian Computation: `abc(P, start=`θ_0`, Nmcmc=`M`, probes=`$\mathbb{S}$`, rw.sd=`$\sigma_{1:p}$`, epsilon=`ϵ`)`, using notation from Table 1, where `P` is a '`pomp`' object with defined methods for `rprocess`, `rmeasure`, `init.state`, `dprior`, and `obs`.

input: Starting parameter, θ_0; simulator for $f_{X_0}(x_0\,;\theta)$; simulator for $f_{X_n|X_{n-1}}(x_n \,|\, x_{n-1}\,;\theta)$; simulator for $f_{Y_n|X_n}(y_n \,|\, x_n\,;\theta)$; data, $y^*_{1:N}$; number of proposals, M; vector of probes, $\mathbb{S} = (\mathbb{S}_1, \ldots, \mathbb{S}_d)$; perturbation scales, $\sigma_{1:p}$; evaluator for prior, $f_\Theta(\theta)$; feature scales, $\tau_{1:d}$; tolerance, ϵ.

1 Compute observed probes: $s^*_i = \mathbb{S}_i(y^*_{1:N})$ for i in $1:d$.
2 **for** m *in* $1:M$ **do**
3 Propose parameters: $[\theta^P_m]_i \sim \text{Normal}\big([\theta_{m-1}]_i, \sigma_i^2\big)$ for i in $1{:}p$.
4 Simulate dataset: $Y_{1:N} \sim f_{Y_{1:N}}(y_{1:N}\,;\theta^P_m)$.
5 Compute simulated probes: $s_i = \mathbb{S}_i(Y_{1:N})$ for i in $1{:}d$.
6 Generate $U \sim \text{Uniform}[0,1]$.
7 Set $\theta_m = \begin{cases} \theta^P_m, & \text{if } \displaystyle\sum_{i=1}^{d}\left(\frac{s_i - s^*_i}{\tau_i}\right)^2 < \epsilon^2 \text{ and } U < \dfrac{f_\Theta(\theta^P_m)}{f_\Theta(\theta_{m-1})}, \\ \theta_{m-1}, & \text{otherwise.} \end{cases}$
8 **end**

output: Samples, $\theta_{1:M}$, representing the posterior distribution, $f_{\Theta|S_{1:d}}(\theta \,|\, s^*_{1:d})$.

generalized method of moments. The vector of summary statistics $\mathbb{S}$, the corresponding random variable S, and the value $s^* = \mathbb{S}(y^*_{1:N})$, are defined as in section 3.4. The goal of ABC is to approximate the posterior distribution of the unknown parameters given $S = s^*$. ABC has typically been motivated by computational considerations, as in point B1 of section 3.4 (Sisson *et al.* 2007; Toni *et al.* 2009; Beaumont 2010). Points B2 and B3 also apply (Ratmann *et al.* 2009).

The key theoretical insight behind ABC algorithms is that an unbiased estimate of the likelihood can be substituted into a Markov chain Monte Carlo algorithm to target the required posterior, the same result that justifies PMCMC (Andrieu and Roberts 2009). However, ABC takes a different approach to approximating the likelihood. The likelihood of the observed features, $\ell_S(\theta) = f_S(s^*;\theta)$, has an approximately unbiased estimate based on a single Monte Carlo realization $Y_{1:N} \sim f_{Y_{1:N}}(\,\cdot\,;\theta)$ given by

$$\hat{\ell}^{ABC}_{\mathbb{S}}(\theta) = \begin{cases} \epsilon^{-d} B_d^{-1} \displaystyle\prod_{i=1}^{d} \tau_i, & \text{if } \displaystyle\sum_{i=1}^{d} \left(\frac{s_i - s^*_i}{\tau_i}\right)^2 < \epsilon^2, \\ 0, & \text{otherwise,} \end{cases} \tag{3}$$

where B_d is the volume of the d-dimensional unit ball. The likelihood approximation in Eq. 3 differs from the synthetic likelihood in Algorithm 5 in that only a single simulation is required. As ϵ become small, the bias in Eq. 3 decreases but the Monte Carlo variability increases. The ABC implementation `abc` (presented in Algorithm 6) is a random walk Metropolis-Hastings algorithm with the likelihood approximation from Eq. 3. In the same style as iterated filtering and PMCMC, we assume a Gaussian random walk in parameter space.

3.6. Nonlinear forecasting

Nonlinear forecasting (NLF) uses simulations to build up an approximation to the one-step prediction distribution that is then evaluated on the data. We saw in section 3.1 that SMC evaluates the prediction density for the observation, $f_{Y_n|Y_{1:n-1}}(y^*_n \,|\, y^*_{1:n-1};\theta)$, by first building an approximation to the prediction density of the latent process, $f_{X_n|Y_{1:n-1}}(x_n \,|\, y^*_{1:n-1};\theta)$. NLF, by contrast, uses simulations to fit a linear regression model which predicts Y_n based on a collection of L lagged variables, $\{Y_{n-c_1}, \ldots, Y_{n-c_L}\}$. The prediction errors when this model is applied to the data give rise to a quantity called the quasi likelihood, which behaves for many purposes like a likelihood (Smith 1993). The implementation in `nlf` maximises the quasi likelihood computed in Algorithm 7, using the `subplex` method (Rowan 1990; King 2008) or any other optimizer offerered by `optim`. The construction of the quasi likelihood in `nlf` follows the specific recommendations of Kendall *et al.* (2005). In particular, the choice of radial basis functions, f_k, in line 5 and the specification of m_k and s in lines 3 and 4 were proposed by Kendall *et al.* (2005) based on trial and error. The quasi likelihood is mathematically most similar to a likelihood when $\min(a_{1:L}) = 1$, so that $\ell_Q(\theta)$ approximates the factorization of the likelihood in Eq. 1. With this in mind, it is natural to set $a_{1:L} = 1:L$. However, Kendall *et al.* (2005) found that a two-step prediction criterion, with $\min(a_{1:L}) = 2$, led to improved numerical performance. It is natural to ask when one might choose to use quasi-likelihood estimation in place of full likelihood estimation implemented by SMC. Some considerations follow, closely related to the considerations for synthetic likelihood and ABC (sections 3.4 and 3.5).

Algorithm 7: Simulated quasi log likelihood for NLF. Pseudocode for the quasi likelihood function optimized by `nlf(P, start=`θ_0`, nasymp=`J`, nconverge=`B`, nrbf=`K`, lags=`L`)`. Using notation from Table 1, `P` is a '`pomp`' object with defined methods for `rprocess`, `rmeasure`, `init.state`, and `obs`.

input: Simulator for $f_{X_n|X_{n-1}}(x_n \,|\, x_{n-1}\,;\theta)$; simulator for $f_{X_0}(x_0\,;\theta)$; simulator for $f_{Y_n|X_n}(y_n \,|\, x_n\,;\theta)$; parameter, θ; data, $y^*_{1:N}$; collection of lags, $c_{1:L}$; length of discarded transient, B; length of simulation, J; number of radial basis functions, K.

1 Simulate long stationary time series: $Y_{1:(B+J)} \sim f_{Y_{1:(B+J)}}(y_{1:(B+J)}\,;\theta)$.

2 Set $Y_{\min} = \min\{Y_{(B+1):(B+J)}\}$, $Y_{\max} = \max\{Y_{(B+1):(B+J)}\}$ and $R = Y_{\max} - Y_{\min}$.

3 Locations for basis functions: $m_k = Y_{\min} + R \times [1.2 \times (k-1)(K-1)^{-1} - 0.1]$ for k in $1{:}K$.

4 Scale for basis functions: $s = 0.3 \times R$.

5 Define radial basis functions: $f_k(x) = \exp\{(x-m_k)^2/2s^2\}$ for k in $1{:}K$.

6 Define prediction function: $H(y_{n-c_1}, y_{n-c_2}, \ldots, y_{n-c_L}) = \sum_{j=1}^{L}\sum_{k=1}^{K} a_{jk} f_k(y_{n-c_j})$.

7 Compute $\{a_{jk} : j \in 1{:}L, k \in 1{:}K\}$ to minimize

$$\hat{\sigma}^2 = \frac{1}{J}\sum_{n=B+1}^{B+J} \big[Y_n - H(Y_{n-c_1}, Y_{n-c_2}, \ldots, Y_{n-c_L})\big]^2.$$

8 Compute the simulated quasi log likelihood:

$$\hat{\ell}_Q(\theta) = -\frac{N-\overline{c}}{2}\log 2\pi\hat{\sigma}^2 - \sum_{n=1+\overline{c}}^{N} \frac{\big[y^*_n - H(y^*_{n-c_1}, y^*_{n-c_2}, \ldots, y^*_{n-c_L})\big]^2}{2\hat{\sigma}^2},$$

where $\overline{c} = \max(c_{1:L})$.

output: Simulated quasi log likelihood, $\hat{\ell}_Q(\theta)$.

C1. NLF benefits from stationarity since (unlike SMC) it uses all time points in the simulation to build a prediction rule valid at all time points. Indeed, NLF has not been considered applicable for non-stationary models and, on account of this, `nlf` is not appropriate if the model includes time-varying covariates. An intermediate scenario between stationarity and full non-stationarity is seasonality, where the dynamic model is forced by cyclical covariates, and this is supported by `nlf` (cf. B1 in section 3.4).

C2. Potentially, quasi-likelihood could be preferable to full likelihood in some situations. It has been argued that a two-step prediction criterion may sometimes be more robust to model misspecification that the likelihood (Xia and Tong 2011) (cf. B2).

C3. Arguably, two-step prediction should be viewed as a diagnostic tool that can be used to complement full likelihood analysis rather than replace it (Ionides 2011) (cf. B3).

C4. NLF does not require that the model be Markovian (cf. B4).

C5. NLF is doubly plug-and-play (cf. B5).

C6. The regression surface reconstruction carried out by NLF does not scale well with the dimension of the observed data. NLF is recommended only for univariate timeseries observations, and this is a requirement for `nlf`.

NLF can be viewed as an estimating equation method, and so standard errors can be computed by standard sandwich estimator or bootstrap techniques (Kendall *et al.* 2005). The optimization in NLF is typically carried out with a fixed seed for the random number generator, so the simulated quasi-likelihood is a deterministic function. If `rprocess` depends smoothly on the random number sequence and on the parameters, and the number of calls to the random number generator does not depend on the parameters, then fixing the seed results in a smooth objective function. However, some common components to model simulators, such as `rnbinom`, make different numbers of calls to the random number generator depending on the arguments, which introduces nonsmoothness into the objective function.

4. Model construction and data analysis: simple examples

4.1. A first example: the Gompertz model

The plug-and-play methods in **pomp** were designed to facilitate data analysis based on complicated models, but we'll first demonstrate the basics of **pomp** using simple discrete-time models, the Gompertz and Ricker models for population growth (Reddingius 1971; Ricker 1954). The Ricker model will be introduced in section 4.5, and the rest of section 4 will use the Gompertz model. The Gompertz model postulates that the density, $X_{t+\Delta t}$, of a population of organisms at time $t+\Delta t$ depends on the density, X_t, at time t according to

$$X_{t+\Delta t} = K^{1-e^{-r\,\Delta t}}\, X_t^{e^{-r\,\Delta t}}\, \varepsilon_t. \tag{4}$$

In Eq. 4, K is the carrying capacity of the population, r is a positive parameter, and the ε_t are independent and identically-distributed lognormal random variables with $\log \varepsilon_t \sim \text{Normal}(0, \sigma^2)$. Additionally, we'll assume that the population density is observed with errors in measurement that are lognormally distributed:

$$\log Y_t \;\sim\; \text{Normal}\left(\log X_t, \tau^2\right). \tag{5}$$

Taking a logarithmic transform of Eq. 4 gives

$$\log X_{t+\Delta t} \;\sim\; \text{Normal}\left(\left(1-e^{-r\,\Delta t}\right)\,\log K + e^{-r\,\Delta t}\,\log X_t, \sigma^2\right). \tag{6}$$

On this transformed scale, the model is linear and Gaussian and so we can obtain exact values of the likelihood function by applying the Kalman filter. Plug-and-play methods are not strictly needed, and this example therefore allows us to compare the results of generally applicable plug-and-play methods with exact results from the Kalman filter. Later we'll look at the Ricker model and a continuous-time model for which no such special tricks are available.

The first step in implementing this model in **pomp** is to construct an R object of class '`pomp`' that encodes the model and the data. This involves the specification of functions to do some or all of `rprocess`, `rmeasure`, and `dmeasure`, along with data and (optionally) other information. The documentation (`?pomp`) spells out the usage of the `pomp` constructor, including detailed specifications for all its arguments and links to several examples.

To begin, we'll 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 Eq. 4.

```
R> gompertz.proc.sim <- function(x, t, params, delta.t, ...) {
+    eps <- exp(rnorm(n = 1, mean = 0, sd = params["sigma"]))
+    S <- exp(-params["r"] * delta.t)
+    setNames(params["K"]^(1 - S) * x["X"]^S * eps, "X")
+  }
```

The translation from the mathematical description Eq. 4 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 likewise be a named numeric vector, with names that match those of `x`. The argument `delta.t` species the time-step size. In this case, the time-step will be 1 unit; we'll see below how this is specified.

Next, we'll implement a simulator for the observation process Eq. 5.

```
R> gompertz.meas.sim <- function(x, t, params, ...) {
+    setNames(rlnorm(n = 1, meanlog = log(x["X"]), sd = params["tau"]), "Y")
+  }
```

Again the translation from the measurement model Eq. 5 is straightforward. When the function `gompertz.meas.sim` is called, the named numeric vector `x` will contain the unobserved states at time `t`; `params` will contain the parameters as before. This return value will be a named numeric vector containing a single draw from the observation process Eq. 5.

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

```
R> gompertz.meas.dens <- function(y, x, t, params, log, ...) {
+    dlnorm(x = y["Y"], meanlog = log(x["X"]), sdlog = params["tau"],
+      log = log)
+  }
```

We'll need this later on for inference using `pfilter`, `mif` and `pmcmc`. With the above in place, we build an object of class '`pomp`' via a call to `pomp`:

```
R> 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 as yet no data; in a moment, we'll generate some simulated data. The second argument (`times`) specifies which of the columns of `data` is the time variable. The `rprocess` argument specifies that the process model simulator will be in discrete time, with each step of duration `delta.t` taken by the function given in the `step.fun` argument. The `rmeasure` argument specifies the measurement model simulator

function. `t0` fixes t_0 for this model; here we have chosen this to be one time unit prior to the first observation.

It is worth noting that implementing the `rprocess`, `rmeasure`, and `dmeasure` components as R functions, as we've done above, leads to needlessly slow computation. As we will see below, **pomp** provides facilities for specifying the model in C, which can accelerate computations many fold.

Before we can simulate from the model, we need to specify some parameter values. The parameters must be a named numeric vector containing at least all the parameters referenced by the functions `gompertz.proc.sim` and `gompertz.meas.sim`. The parameter vector needs to determine the initial condition $X(t_0)$ as well. Let's take our parameter vector to be

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

The parameters r, K, σ, and τ appear in `gompertz.proc.sim` and `gompertz.meas.sim`. The initial condition X_0 is also given in `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 however parameterize the initial condition distribution arbitrarily using `pomp`'s optional `initializer` argument.

We can now simulate the model at these parameters:

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

Now `gompertz` is identical to what it was before, except that the missing data 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

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

Fig. 1 shows the results of this operation.

4.2. Computing likelihood using SMC

As discussed in section 3, some parameter estimation algorithms in the **pomp** package are doubly plug-and-play in that they require only `rprocess` and `rmeasure`. These include the nonlinear forecasting algorithm `nlf`, the probe-matching algorithm `probe.match`, and approximate Bayesian computation via `abc`. The plug-and-play full-information methods in **pomp**, however, require `dmeasure`, i.e., the ability to evaluate the likelihood of the data given the unobserved state. The `gompertz.meas.dens` above does this, but we must incorporate it into the '`pomp`' object in order to use it. We can do this with another call to `pomp`:

```
R> 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`, an implementation of Algorithm 1. We must decide how many concurrent realizations (*particles*) to use: the larger

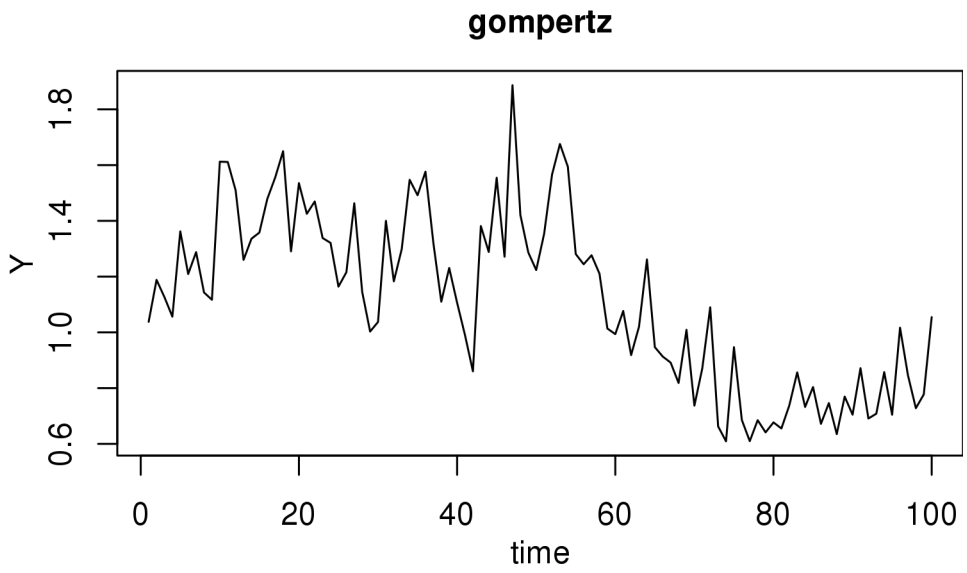

Figure 1: Simulated data from the Gompertz model (Eqs. 4 and 5). This figure shows the result of executing `plot(gompertz, variables = "Y")`.

the number of particles, the smaller the Monte Carlo error but the greater the computational burden. Here, we run `pfilter` with 1000 particles to estimate the likelihood at the true parameters:

```
R> pf <- pfilter(gompertz, params = theta, Np = 1000)
R> loglik.truth <- logLik(pf)
R> loglik.truth
[1] 36.27102
```

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

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

or simply

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

Now let's compute the log likelihood at a different point in parameter space, one for which r, K, and σ are each 50% higher than their true values.

```
R> theta.guess <- theta.true <- coef(gompertz)
R> theta.guess[c("r", "K", "sigma")] <- 1.5 * theta.true[c("r", "K", "sigma")]
R> pf <- pfilter(gompertz, params = theta.guess, Np = 1000)
R> loglik.guess <- logLik(pf)
R> loglik.guess
[1] 25.19585
```

In this case, the Kalman filter computes the exact log likelihood at the true parameters to be 36.01, while the particle filter with 1000 particles gives 36.27. 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; we'll demonstrate this in section 4.3.

4.3. Maximum likelihood estimation via iterated filtering

Let's use the iterated filtering approach described in section 3.2 to obtain an approximate maximum likelihood estimate 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. The following codes accomplish this.

```
R> estpars <- c("r", "sigma", "tau")
R> library("foreach")
R> mif1 <- foreach(i = 1:10, .combine = c) %dopar% {
+    theta.guess <- theta.true
+    rlnorm(n = length(estpars), meanlog = log(theta.guess[estpars]),
+           sdlog = 1) -> theta.guess[estpars]
+    mif(gompertz, Nmif = 100, start = theta.guess, transform = TRUE,
+        Np = 2000, var.factor = 2, cooling.fraction = 0.7,
+        rw.sd = c(r = 0.02, sigma = 0.02, tau = 0.02))
+  }
R> pf1 <- foreach(mf = mif1, .combine = c) %dopar% {
+    pf <- replicate(n = 10, logLik(pfilter(mf, Np = 10000)))
+    logmeanexp(pf)
+  }
```

Note that we've set `transform = TRUE` in the call to `mif` above: this transforms the parameters before performing iterated filtering to enforce the positivity of parameters. We'll see how such parameter transformations are implemented in section 4.5. Note also that we've used the **foreach** package (Revolution Analytics and Weston 2014) to parallelize the computations.

Each of the 10 `mif` runs ends up at a different point estimate (Fig. 2). We focus on that with the highest estimated likelihood, having evaluated the likelihood several times to reduce the Monte Carlo error in the likelihood evaluation. The particle filter produces an unbiased estimate of the likelihood; therefore, we'll average the likelihoods, not the log likelihoods.

```
R> mf1 <- mif1[[which.max(pf1)]]
R> theta.mif <- coef(mf1)
R> loglik.mif <- replicate(n = 10, logLik(pfilter(mf1, Np = 10000)))
R> loglik.mif <- logmeanexp(loglik.mif, se = TRUE)
R> theta.true <- coef(gompertz)
R> loglik.true <- replicate(n = 10, logLik(pfilter(gompertz, Np = 20000)))
R> loglik.true <- logmeanexp(loglik.true, se = TRUE)
```

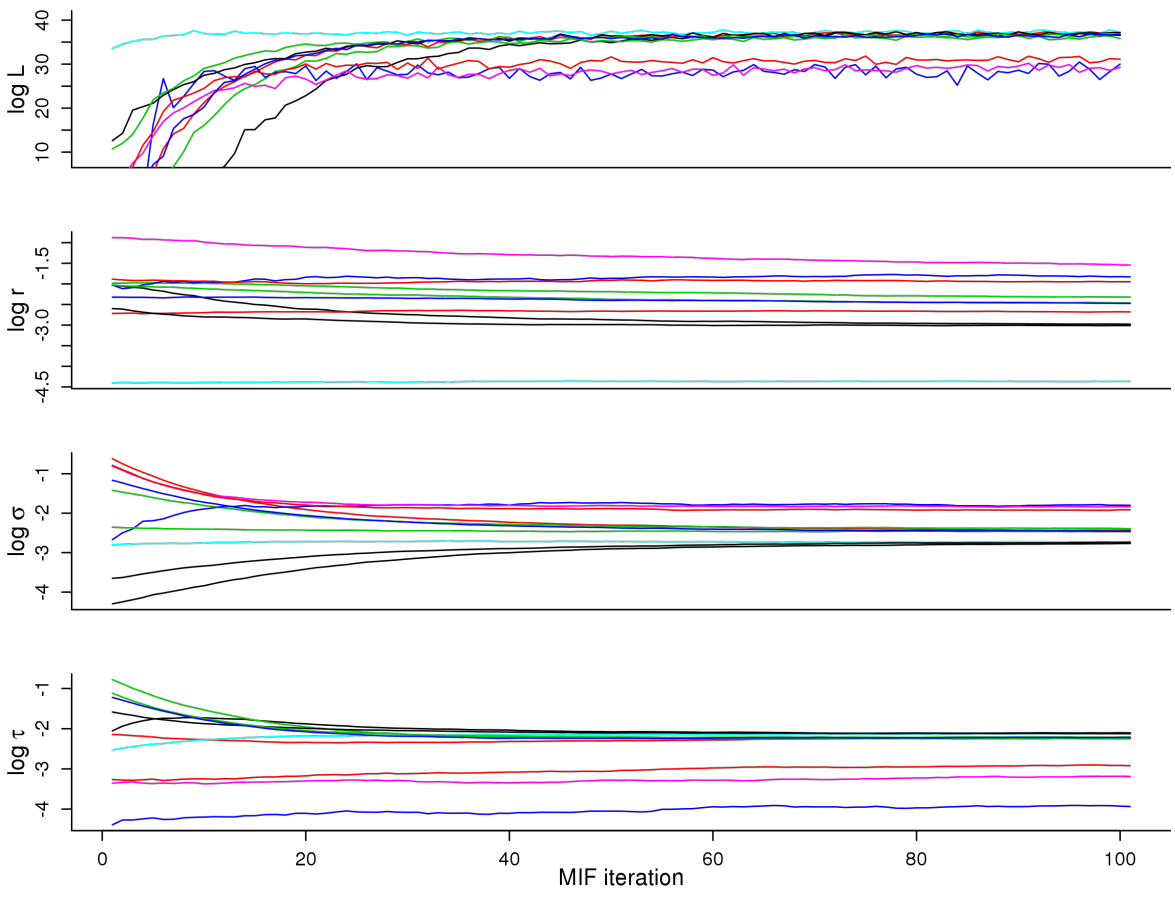

Figure 2: Convergence plots can be used to help diagnose convergence of the iterated filtering algorithm. These and additional diagnostic plots are produced when `plot` is applied to a '`mif`' or '`mifList`' object.

For the calculation above, we have replicated the iterated filtering search, made a careful estimation of the log likelihood, $\hat{\ell}$, and its standard error using `pfilter` at each of the resulting point estimates, and then chosen the parameter corresponding to the highest likelihood as our numerical approximation to the MLE. Taking advantage of the Gompertz model's tractability, we also use the Kalman filter to maximize the exact likelihood, ℓ, and evaluate it at the estimated MLE obtained by `mif`. The results are:

	r	σ	τ	`pfilter` $\hat{\ell}$	s.e.	exact ℓ
truth	0.1000	0.1000	0.1000	36.02	0.07	36.01
`mif` MLE	0.0127	0.0655	0.1200	37.61	0.08	37.62
exact MLE	0.0322	0.0694	0.1170	37.87	0.05	37.88

Usually, the last row and column of this results table would not be available even for a simulation study validating the inference methodology for a known POMP model. In this case, we see that the `mif` procedure is successfully maximizing the likelihood up to an error of about 0.1 log units.

4.4. Full-information Bayesian inference via PMCMC

To carry out Bayesian inference we need to specify a prior distribution on unknown parameters. The `pomp` constructor function provides the `rprior` and `dprior` arguments, which can be filled with functions that simulate from and evaluate the prior density, respectively. Methods based on MCMC require evaluation of the prior density (`dprior`), but not simulation (`rprior`), so we specify `dprior` for the Gompertz model as follows.

```
R> hyperparams <- list(min = coef(gompertz)/10, max = coef(gompertz) * 10)
R> gompertz.dprior <- function (params, ..., log) {
+    f <- sum(dunif(params, min = hyperparams$min, max = hyperparams$max,
+                   log = TRUE))
+    if (log) f else exp(f)
+  }
```

The PMCMC algorithm described in section 3.3 can then be applied to draw a sample from the posterior. Recall that, for each parameter proposal, PMCMC pays the full price of a particle-filtering operation in order to obtain the Metropolis-Hastings acceptance probability. Compare this to iterated filtering, which obtains for the same price an estimate of the derivative and a probable improvement of the parameters. For this reason, PMCMC is relatively inefficient at maximizing the likelihood. When Bayesian inference is the goal, it is advisable to first locate a neighborhood of the MLE using, for example, iterated filtering. PMCMC can then be initialized in this neighborhood to sample from the posterior distribution. The following adopts this approach, running 5 independent PMCMC chains.

```
R> pmcmc1 <- foreach(i=1:5,.combine=c) %dopar% {
+    pmcmc(pomp(gompertz, dprior = gompertz.dprior), start = theta.mif,
+          Nmcmc = 40000, Np = 100, max.fail = Inf,
+          rw.sd = c(r = 0.01, sigma = 0.01, tau = 0.01))
+  }
```

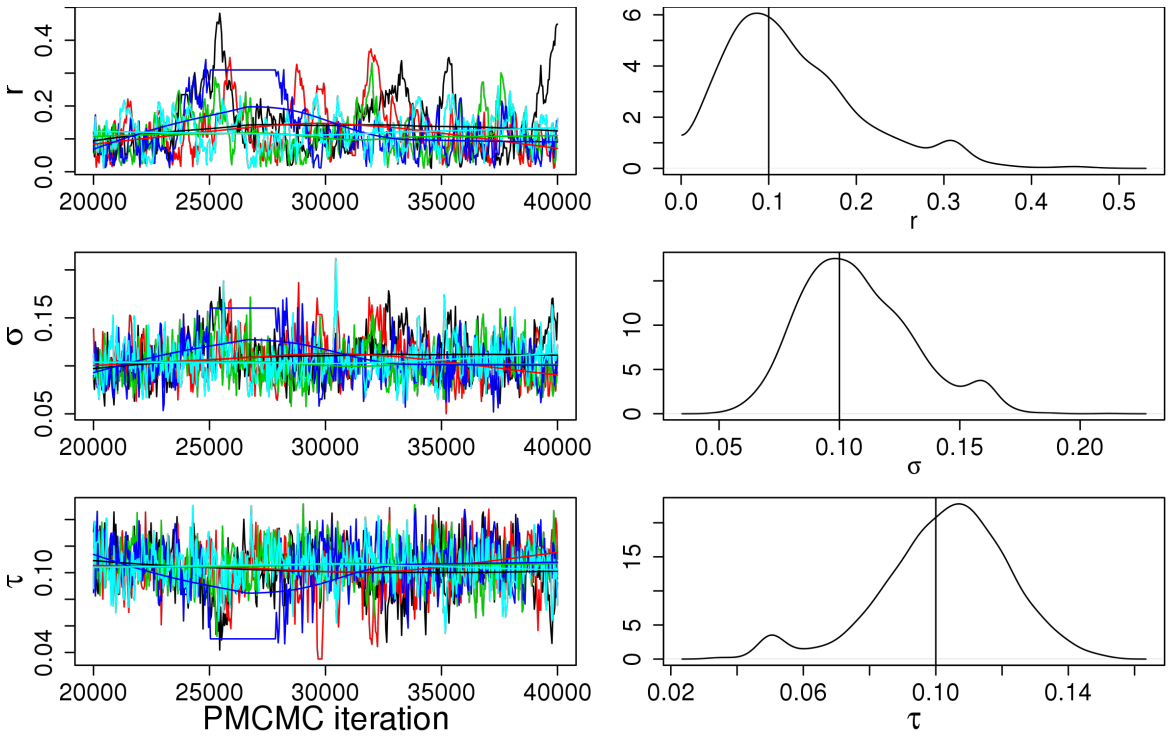

Figure 3: Diagnostic plots for the PMCMC algorithm. The trace plots in the left column show the evolution of 5 independent MCMC chains after a burn-in period of length 20000. Kernel density estimates of the marginal posterior distributions are shown at right. The effective sample size of the 5 MCMC chains combined is lowest for the r variable and is 160: the use of 40000 proposal steps in this case is a modest number. The density plots at right show the estimated marginal posterior distributions. The vertical line corresponds to the true value of each parameter.

Comparison with the analysis of section 4.3 reinforces the observation of Bhadra (2010) that PMCMC can require orders of magnitude more computation than iterated filtering. Iterated filtering may have to be repeated multiple times while computing profile likelihood plots, whereas one successful run of PMCMC is sufficient to obtain all required posterior inferences. However, in practice, multiple runs from a range of starting points is always good practice since assessment of convergence within a single chain is unreliable. To verify the convergence of the approach or to compare the performance with other approaches, we can use diagnostic plots produced by the `plot` method Fig. 3.

4.5. A second example: the Ricker model

In section 4.6, we'll illustrate probe matching (section 3.4) using a stochastic version of the Ricker map (Ricker 1954). We switch models to allow direct comparison with Wood (2010), whose synthetic likelihood computations are reproduced below. In particular, the results of section 4.6 demonstrate frequentist inference using synthetic likelihood and also show that the full likelihood is both numerically tractable and reasonably well behaved, contrary to the claim of Wood (2010). We'll also take the opportunity to demonstrate features of **pomp** that allow acceleration of model codes through the use of R's facilities for compiling and dynamically linking C code.

The Ricker model is another discrete-time model for the size of a population. The population size, N_t, at time t is postulated to obey

$$N_{t+1} = r\,N_t\,\exp(-N_t + e_t), \qquad e_t \sim \text{Normal}[0, \sigma^2]. \tag{7}$$

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

$$Y_t \sim \text{Poisson}(\phi\,N_t). \tag{8}$$

As before, we'll need to implement the model's state-process simulator (`rprocess`). We have the option of writing these functions in R, as we did with the Gompertz model. However, we can realize many-fold speed-ups by writing these in C. In particular, **pomp** allows us to write snippets of C code that it assembles, compiles, and dynamically links into a running R session. To begin the process, we'll write snippets for the `rprocess`, `rmeasure`, and `dmeasure` components.

```
R> ricker.sim <- '
+     e = rnorm(0, sigma);
+     N = r * N * exp(-N + e);
+  '
R> ricker.rmeas <- '
+     y = rpois(phi * N);
+  '
R> ricker.dmeas <- '
+     lik = dpois(y, phi * N, give_log);
+  '
```

Note that, in this implementation, both N and e are state variables. The logical flag `give_log` requests the likelihood when FALSE, the log likelihood when TRUE. Notice that, in these

snippets, we never declare the variables; **pomp** will construct the appropriate declarations automatically.

In a similar fashion, we can add transformations of the parameters to enforce constraints.

```
R> par.trans <- '
+     Tr = exp(r);
+     Tsigma = exp(sigma);
+     Tphi = exp(phi);
+     TN_0 = exp(N_0);
+  '
R> par.inv.trans <- '
+     Tr = log(r);
+     Tsigma = log(sigma);
+     Tphi = log(phi);
+     TN_0 = log(N_0);
+  '
```

Note that in the foregoing `C` snippets, the prefix `T` designates the transformed version of the parameter.

Now we can construct a '`pomp`' object as before and fill it with simulated data:

```
R> pomp(data = data.frame(time = seq(0, 50, by = 1), y = NA),
+       rprocess = discrete.time.sim(step.fun = Csnippet(ricker.sim),
+         delta.t = 1), rmeasure = Csnippet(ricker.rmeas),
+       dmeasure = Csnippet(ricker.dmeas),
+       parameter.transform = Csnippet(par.trans),
+       parameter.inv.transform = Csnippet(par.inv.trans),
+       paramnames = c("r", "sigma", "phi", "N.0", "e.0"),
+       statenames = c("N", "e"), times = "time", t0 = 0,
+       params = c(r = exp(3.8), sigma = 0.3, phi = 10,
+         N.0 = 7, e.0 = 0)) -> ricker
model codes written to '/tmp/Rtmpy9qogI/pomp6CA6735D69F2.c'
link to shared-object library '/tmp/Rtmpy9qogI/pomp6CA6735D69F2.so'
R> ricker <- simulate(ricker,seed=73691676L)
```

4.6. Feature-based synthetic likelihood maximization

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 useful probes are included with the package, including those recommended by Wood (2010). In this illustration, we will make use of these probes: `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 a list of probes:

```
R> 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))
```

Each element of `plist` is a function of a single argument. Each of these functions can be applied to the data in `ricker` and to simulated data sets. Calling **pomp**'s function `probe` results in the application of these functions to the data, and to each of some large number, `nsim`, of simulated data sets, and finally to a comparison of the two. [Note that probe functions may be vector-valued, so a single probe taking values in $\mathbb{R}^k$ formally corresponds to a collection of k probe functions in the terminology of section 3.4.] Here, we'll apply `probe` to the Ricker model at the true parameters and at a wild guess.

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

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

An example of a diagnostic plot (using a smaller set of probes) is shown in Fig. 4. Among the quantities returned by `summary` is the synthetic likelihood (Algorithm 5). One can attempt to identify parameters that maximize this quantity; this procedure is referred to in **pomp** as "probe matching". Let us now attempt to fit the Ricker model to the data using probe-matching.

```
R> pm <- probe.match(pb.guess, est = c("r", "sigma", "phi"), transform = TRUE,
+    method = "Nelder-Mead", maxit = 2000, seed = 1066L, reltol = 1e-08)
```

This code runs `optim`'s 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 need not specify them explicitly here. While `probe.match` provides substantial flexibility in choice of optimization algorithm, for situations requiring greater flexibility, **pomp** provides the function `probe.match.objfun`, which constructs an objective function suitable for use with arbitrary optimization routines.

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`:

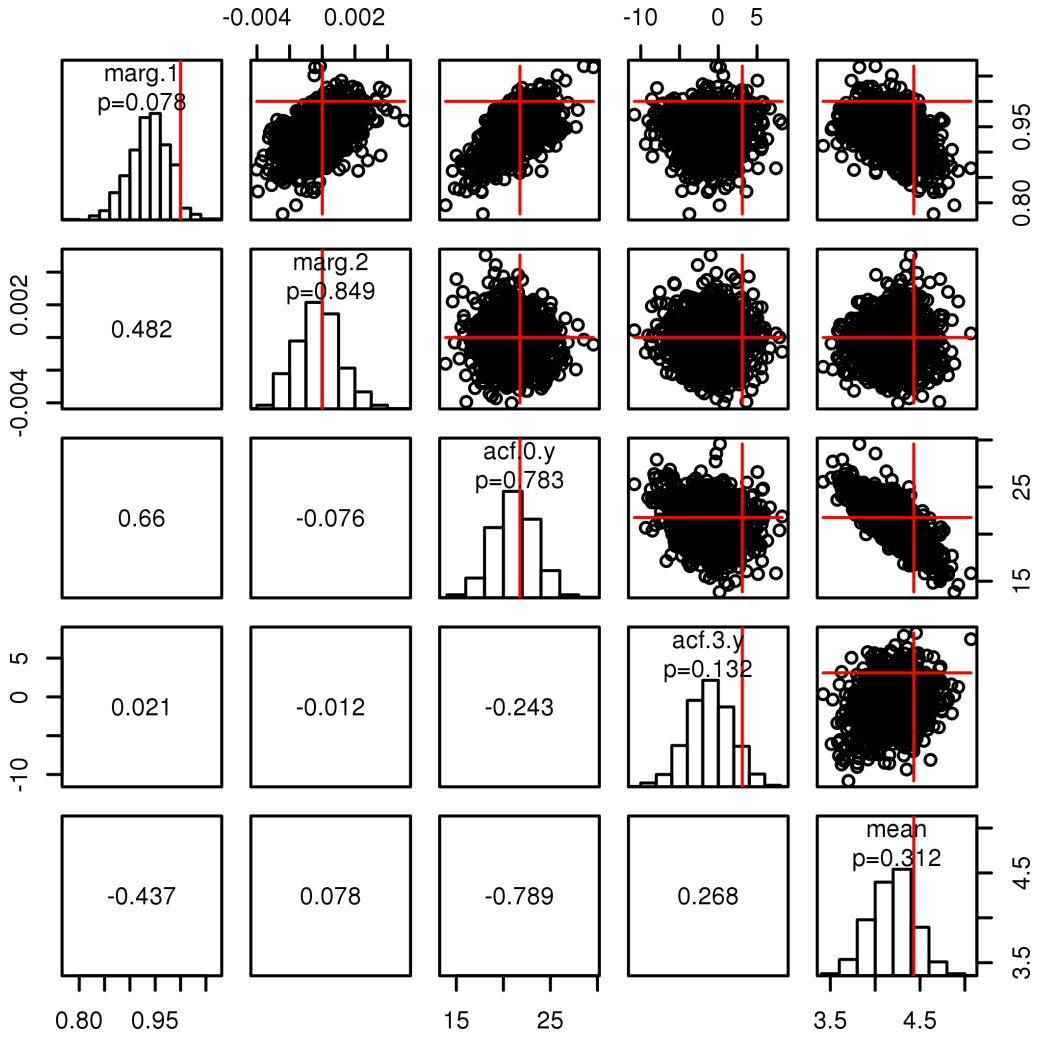

Figure 4: Results of `plot` on a '`probed.pomp`'-class object. Above the diagonal, the pairwise scatterplots show the values of the probes on each of 1000 data sets. The red lines show the values of each of the probes on the data. The panels along the diagonal show the distributions of the probes on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

```
R> mf <- mif(ricker, start = guess, Nmif = 100, Np = 1000, transform = TRUE,
+    cooling.fraction = 0.95^50, var.factor = 2, ic.lag = 3, max.fail = 50,
+    rw.sd = c(r = 0.1, sigma = 0.1, phi = 0.1))
R> mf <- continue(mf, Nmif = 500, max.fail = 20)
```

The following table compares parameters, Monte Carlo likelihoods ($\hat{\ell}$), and synthetic likelihoods ($\hat{\ell}_{\mathbb{S}}$, based on the probes in `plist`) at each of (a) the guess, (b) the truth, (c) the MLE from `mif`, and (d) the maximum synthetic likelihood estimate (MSLE) from `probe.match`.

	r	σ	ϕ	$\hat{\ell}$	$\hat{\ell}_{\mathbb{S}}$
guess	20.0	1.000	20.0	-230.9	-12.3
truth	44.7	0.300	10.0	-139.5	17.5
MLE	45.0	0.186	10.2	-137.7	18.0
MSLE	42.1	0.337	11.3	-145.6	20.4

These results demonstrate that it is possible, and indeed not difficult, to maximize the likelihood for this model, contrary to the claim of Wood (2010).

4.7. Bayesian feature matching via ABC

Whereas synthetic likelihood carries out many simulations for each likelihood estimation, ABC (as described in section 3.5) uses only one. Each iteration of ABC is therefore much quicker, essentially corresponding to the cost of SMC with a single particle or synthetic likelihood with a single simulation. A consequence of this is that ABC cannot determine a good relative scaling of the features within each likelihood evaluation and this must be supplied in advance. One can imagine an adaptive version of ABC which modifies the scaling during the course of the algorithm, but here we do a preliminary calculation to accomplish this.

```
R> plist <- list(probe.mean(var = "Y", transform = sqrt),
+                probe.acf("Y", lags = c(0, 5, 10, 20)),
+                probe.marginal("Y", ref = obs(gompertz)))
+  psim <- probe(gompertz, probes = plist, nsim = 500)
+  scale.dat <- apply(psim$simvals, 2, sd)
R> abc1 <- foreach(i = 1:5, .combine = c) %dopar% {
+    abc(pomp(gompertz, dprior = gompertz.dprior), Nabc = 4e6,
+        probes = plist, epsilon = 2, scale = scale.dat,
+        rw.sd = c(r = 0.01, sigma = 0.01, tau = 0.01))
+  }
```

The effective sample size of the ABC chains is lowest for the r parameter (as was the case for PMCMC) and is 510, as compared to 160 for `pmcmc` in section 4.4. The total computational effort allocated to `abc` here matches that for `pmcmc` since `pmcmc` used 100 particles for each likelihood evaluation but is awarded 100 times fewer Metropolis-Hastings steps. In this example, we conclude that `abc` converges somewhat more rapidly (as measured by total computational effort) than `pmcmc`. Fig. 5 investigates the statistical efficiency of `abc` on this example. We see that `abc` gives rise to somewhat broader posterior distributions than the

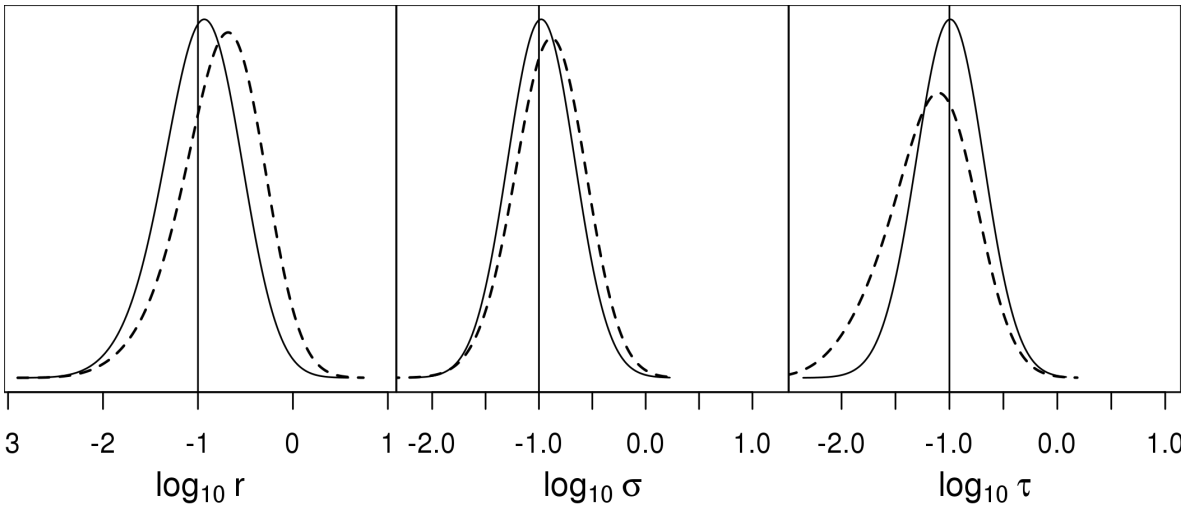

Figure 5: Marginal posterior distributions using full information via `pmcmc` (solid line) and partial information via `abc` (dashed line). Kernel estimates are shown for the posterior marginal density of r (left panel), σ (middle panel), and τ (right panel). The vertical lines indicate the true values of each parameter.

full-information posteriors from `pmcmc` As in all numerical studies of this kind, one cannot readily generalize from one particular example: even for this specific model and dataset, the conclusions might be sensitive to the algorithmic settings. However, one should be aware of the possibility of losing substantial amounts of information even when the features are based on reasoned scientific argument (Shrestha *et al.* 2011; Ionides 2011). Despite this loss of statistical efficiency, points B2–B5 of section 3.4 identify situations in which ABC may be the only practical method available for Bayesian inference.

4.8. Parameter estimation by simulated quasi-likelihood

Within the **pomp** environment, it is fairly easy to try a quick comparison to see how `nlf` (section 3.6) compares with `mif` (section 3.2) on the Gompertz model. Carrying out a simulation study with a correctly specified POMP model is appropriate for assessing computational and statistical efficiency, but does not contribute to the debate on the role of two-step prediction criteria to fit misspecified models (Xia and Tong 2011; Ionides 2011). The `nlf` implementation we will use to compare to the `mif` call from section 4.3 is

```
R> nlf1 <- nlf(gompertz, nasymp = 1000, nconverge = 1000, lags = c(2, 3),
+              start = c(r = 1, K = 2, sigma = 0.5, tau = 0.5, X.0 = 1),
+              est = c("r", "sigma", "tau"), transform.params = TRUE)
```

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. The `transform.params = TRUE` setting causes the optimization to be performed on the trans-

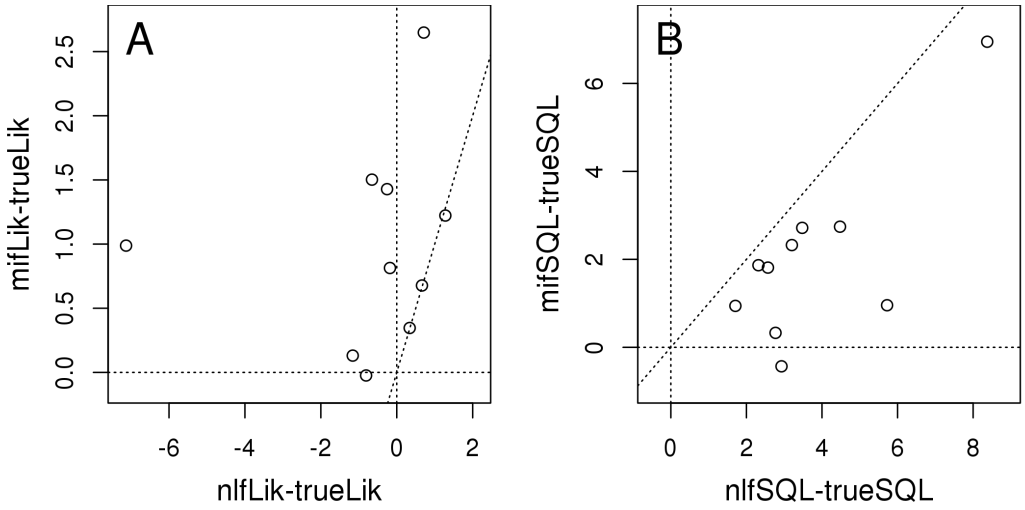

Figure 6: Comparison of `mif` and `nlf` for 10 simulated datasets using two criteria. (A) Improvement in likelihood at point estimate over the true parameter value. (B) Improvement in simulated quasi-likelihood at point estimate over the true parameter value. In both panels, the diagonal line is the 1-1 line.

formed scale, as in section 4.3. In the call above `lags = c(2, 3)` specifies that the autoregressive model predicts each observation, y_t using y_{t-2} and y_{t-3}, as recommended by Kendall *et al.* (2005). The quasilikelihood is optimized numerically, so the reliability of the optimization should be assessed by doing multiple fits with different starting parameter values: the results of a small experiment (not shown) indicate that, on these simulated data, repeated optimization isn't needed. `nlf` defaults to optimization by the subplex method (Rowan 1990; King 2008), though all optimization methods provided by `optim` are available as well. `nasymp` sets the length of the simulation on which the quasilikelihood is based; larger values will give less variable parameter estimates, but will slow down the fitting process. The computational demand of `nlf` is dominated by the time required to generate the model simulations, so efficient coding of `rprocess` is worthwhile.

Fig. 6 compares the true parameter, θ, with the maximum likelihood estimate (MLE), $\hat{\theta}$, from `mif` and the maximized simulated quasi-likelihood (MSQL), $\breve{\theta}$, from `nlf`. Fig. 6A plots $\hat{\ell}(\hat{\theta}) - \hat{\ell}(\theta)$ against $\hat{\ell}(\breve{\theta}) - \hat{\ell}(\theta)$, showing that the MSQL estimate can fall many units of log likelihood short of the MLE. Fig. 6B plots $\hat{\ell}_Q(\hat{\theta}) - \hat{\ell}_Q(\theta)$ against $\hat{\ell}_Q(\breve{\theta}) - \hat{\ell}_Q(\theta)$, showing that likelihood-based inference is almost as good as `nlf` at optimizing the simulated quasi-likelihood criterion which `nlf` targets. Fig. 6 suggests that the MSQL can be inefficient: some weakly identified combination of parameters that is close to its maximum at the MLE nevertheless attains its maximum at some distance from the MLE. Another possibility is that this particular implementation of `nlf` was unfortunate. Each `mif` optimization took 32.8 sec to run, compared to 6.5 sec for `nlf`, and it is possible that extra computer time or other algorithmic adjustments could substantially improve either or both estimators. It is hard to ensure a fair comparison between methods, and in practice there is little substitute for some experimentation with different methods and algorithmic settings on a problem of interest.

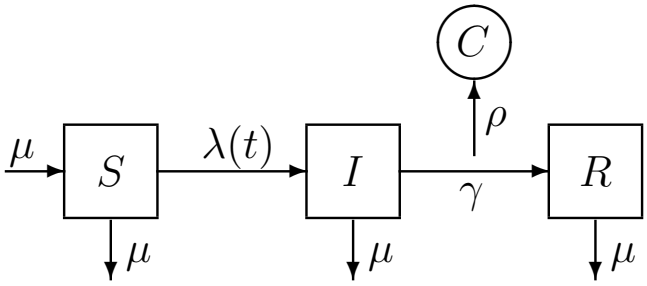

Figure 7: Diagram of the SIR epidemic model. The host population is divided into three classes according to infection status: S, susceptible hosts; I, infected (and infectious) hosts; R, recovered and immune hosts. Births result in new susceptibles and all individuals have a common death rate μ. Since 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.

If the motivation for using NLF is preference for 2-step prediction over the likelihood, a comparison with SMC-based likelihood evaluation and maximization is useful to inform the user of the consequences of that preference.

5. A more complex example: epidemics in continuous time

5.1. A stochastic, seasonal SIR model.

A mainstay of theoretical epidemiology, the SIR model describes the progress of a contagious, immunizing infection through a population of hosts. The hosts are divided into three classes, according to their status vis-à-vis the infection (Fig. 7). The susceptible class (S) contains those that have not yet been infected and are thereby still susceptible to it; the infected class (I) comprises those who are currently infected and, by assumption, infectious; the removed class (R) includes those who are recovered or quarantined as a result of the infection. Individuals in R 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. We will here assume that the birth rate, death rate, and the rate of transition, γ, from I to R are constants. The S to I transition rate, the so-called *force of infection*, $\lambda(t)$, however, should be an increasing function of $I(t)$. For many infections, it's reasonable to assume that the $\lambda(t)$ is jointly proportional to the fraction of the population infected and the rate at which an individual comes into contact with others. Here, we'll make these assumptions, writing $\lambda(t) = \beta I(t)/N$, where β is the transmission rate

and $N = S + I + R$ is the population size. We'll go further and assume that birth and death rates are equal and independent of infection status; we'll let μ denote the common rate. A consequence is that the expected population size remains constant.

Under these assumptions, 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. Specifically, the SIR deterministic skeleton is

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

It is typically impossible to monitor S, I, and R, directly. It sometimes happens, however, that public health authorities keep records of *cases*, i.e., individual infections. 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 negative binomial process

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

where $\Delta_{I\to R}(t_1, t_2)$ is the accumulated number of recoveries that have occured over the interval in question, ρ is the *reporting rate*, i.e., the probability that a given infection is observed and recorded, θ is the negative binomial "size" parameter, and the notation is meant to indicate that $\mathbb{E}\left[C(t_1, t_2)\,|\,\Delta_{I\to R}(t_1, t_2) = H\right] = \rho\,H$ and $\text{Var}\left[C(t_1, t_2)\,|\,\Delta_{I\to R}(t_1, t_2) = H\right] = \rho\,H + \rho^2\,H^2/\theta$. The fact that the observed data are linked to an accumulation, as opposed to an instantaneous value, introduces a little bit of complication, which we discuss below.

5.2. 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`, `rmeasure`, `dmeasure`, and `skeleton` components. We have the option of writing these functions in **R**. However, we can realize many-fold speed-ups by writing these in C. In particular, one writes snippets of C code that are assembled, compiled, and dynamically linked.

To begin the process, we'll write snippets that specify the measurement model (`rmeasure` and `dmeasure`):

```
R> rmeas <- '
+    cases = rnbinom_mu(theta, rho * incid);
+  '
R> dmeas <- '
+    lik = dnbinom_mu(cases, theta, rho * incid, give_log);
+  '
```

Here, we're using `cases` to refer to the data (number of reported cases) and `incid` to refer to the true incidence (number of new infections) over the reporting interval. The negative binomial simulator `rnbinom_mu` and density function `dnbinom_mu` are provided by **R**. The logical flag `give_log` requests the likelihood when `FALSE`, the log likelihood when `TRUE`. Notice

that, in these snippets, we never declare the variables; `pomp` will construct the appropriate declarations automatically.

For the `rprocess` portion, we could 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 would prove prohibitively slow. If we are willing to live with an approximate simulation scheme, we can use the so-called "tau-leap" algorithm, one version of which is implemented in **pomp** via 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 the above assumptions, the number of individuals leaving any of the classes by all available routes over a particular time interval is 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

$$(\Delta_{S\to I}, \Delta_{S\to}, S - \Delta_{S\to I} - \Delta_{S\to}) \;\sim\; \text{multinomial}\,(S(t); p_{S\to I}, p_{S\to}, 1 - p_{S\to I} - p_{S\to})\,. \quad (9)$$

The expression on the right arises with sufficient frequency in compartmental models that **pomp** provides special functions for it. In **pomp**, the random variable $(\Delta_{S\to I}, \Delta_{S\to})$ in Eq. 9 is said to have an *Euler-multinomial* distribution. The **pomp** functions `reulermultinom` and `deulermultinom` provide facilities for respectively drawing random deviates from, and computing the p.m.f. of, such distributions. As the help pages relate, `reulermultinom` and `deulermultinom` parameterize the Euler-multinomial distributions by the size ($S(t)$ in Eq. 9), rates ($\lambda(t)$ and μ), and time interval δt. Obviously, the Euler-multinomial distributions generalize to an arbitrary number of exit routes.

The help page (`?euler.sim`) informs us that to use `euler.sim`, we need to specify a function that advances the states from t to $t+\delta t$. Again, we'll write this in `C` to realize faster run-times:

```
R> sir.step <- '
+    double rate[6];
+    double trans[6];
+    rate[0] = mu * popsize;
+    rate[1] = beta * I / popsize;
+    rate[2] = mu;
+    rate[3] = gamma;
+    rate[4] = mu;
+    rate[5] = mu;
```

```
+    trans[0] = rpois(rate[0] * dt);
+    reulermultinom(2, S, &rate[1], dt, &trans[1]);
+    reulermultinom(2, I, &rate[3], dt, &trans[3]);
+    reulermultinom(1, R, &rate[5], dt, &trans[5]);
+    S += trans[0] - trans[1] - trans[2];
+    I += trans[1] - trans[3] - trans[4];
+    R += trans[3] - trans[5];
+    incid += trans[3];
+  '
```

As before, the undeclared variables will be handled by `pomp`. Note, however, that in the above we do declare certain local variables. In particular, the `rate` and `trans` arrays hold the rates and numbers of transition events, respectively. Note too, that we make use of **pomp**'s C interface to `reulermultinom`.

Two significant wrinkles remains to be explained. First, notice that in `sir.step`, the variable `incid` accumulates the total number of recoveries. Thus, `incid` will be a counting process that is nondecreasing with time. In fact, the number of recoveries within an interval, $\Delta_{I\to R}(t_1,t_2) = \texttt{incid}(t_2) - \texttt{incid}(t_1)$. Clearly, including `incid` 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` function's `zeronames` argument, as we will see in a moment. The section on "accumulator variables" in the "Advanced Topics in **pomp**" document (provided with the package) 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`. Let's construct it now and fill it with simulated data.

```
R> pomp(data = data.frame(cases = NA, time = seq(0, 10, by=1/52)),
+       times = "time", t0 = -1/52, dmeasure = Csnippet(dmeas),
+       rmeasure = Csnippet(rmeas), rprocess = euler.sim(
+         step.fun = Csnippet(sir.step), delta.t = 1/52/20),
+       statenames = c("S", "I", "R", "incid"),
+       paramnames = c("gamma", "mu", "theta", "beta", "popsize",
+         "rho", "S.0", "I.0", "R.0"), zeronames=c("incid"),
+       initializer=function(params, t0, ...) {
+         x0 <- c(S = 0, I = 0, R = 0, incid = 0)
+         fracs <- params[c("S.0", "I.0", "R.0")]
+         x0[1:3] <- round(params['popsize'] * fracs/sum(fracs))
+         x0
```

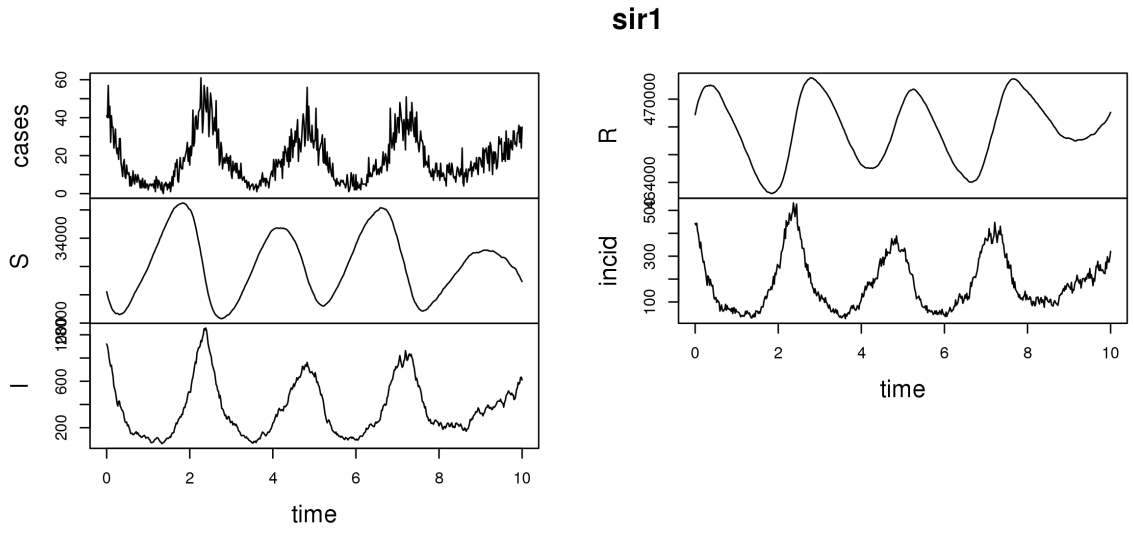

Figure 8: Result of `plot(sir1)`. The '`pomp`' object `sir1` contains the SIR model with simulated data.

```
+       }, params = c(popsize = 500000, beta = 400, gamma = 26,
+             mu = 1/50, rho = 0.1, theta = 100, S.0 = 26/400,
+             I.0 = 0.002, R.0 = 1)) -> sir1
R> simulate(sir1, seed = 1914679908L) -> sir1
```

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 infectious period of 2 wk ($1/\gamma = 1/26$ yr) and a basic reproductive number, R_0 of $\beta/(\gamma+\mu) \approx 15$. We've assumed a host population size of 500,000 and 10% reporting efficiency. Fig. 8 shows one realization of this process.

5.3. 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 four facts into account:

1. For many infections, the contact rate is *seasonal*: $\beta = \beta(t)$ is a periodic function of time.

2. The host population may not be truly closed: *imported infections* arise when infected individuals visit the host population and transmit.

3. The host population need not be constant in size. If we have data, for example, on the numbers of births occurring in the population, we can incorporate this directly into the model.

4. 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.

To incorporate seasonality, we'd like to assume a flexible functional form for $\beta(t)$. Here, we'll use a three-coefficient Fourier series:

$$\log \beta(t) = b_0 + b_1 \cos 2\pi t + b_2 \sin 2\pi t.$$

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

Let's suppose we have data on the number of births occurring each month in this population and that these data are in the form of a data-frame `birthdat` with columns `time` and `births`.

We can incorporate this variable birthrate into our model by passing it as a covariate to the simulation code. When we pass `birthdat` as the `covar` argument to `pomp`, we'll create a look-up table that will be used within the simulator. The linear interpolation of the lookup table will be handled transparently by **pomp**: from the user's perspective, a variable `births` will simply be available for use.

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 random phase in β:

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

where Φ satisfies the stochastic differential equation

$$d\Phi = dt + \sigma\, dW_t,$$

where $dW(t)$ is a white noise, specifically an increment of standard Brownian motion.

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

```
R> seas.sir.step <- '
+    double rate[6];
+    double trans[6];
+    double beta;
+    double dW;
+    beta = exp(b1 + b2 * cos(2 * M_PI * phase) + b3 * sin(2 * M_PI * phase));
+    rate[0] = births;
+    rate[1] = beta * (I + iota) / popsize;
+    rate[2] = mu;
+    rate[3] = gamma;
+    rate[4] = mu;
+    rate[5] = mu;
+    trans[0] = rpois(rate[0] * dt);
+    reulermultinom(2, S, &rate[1], dt, &trans[1]);
+    reulermultinom(2, I, &rate[3], dt, &trans[3]);
```

```
+    reulermultinom(1, R, &rate[5], dt, &trans[5]);
+    dW = rnorm(dt, sigma * sqrt(dt));
+    S += trans[0] - trans[1] - trans[2];
+    I += trans[1] - trans[3] - trans[4];
+    R += trans[3] - trans[5];
+    incid += trans[3];
+    pop += trans[0] - trans[2] - trans[4] - trans[5];
+    phase += dW;
+    noise += (dW - dt) / sigma;
+  '
R> pomp(sir1, rprocess = euler.sim(
+         step.fun = Csnippet(seas.sir.step), delta.t = 1/52/20),
+       dmeasure = Csnippet(dmeas), rmeasure = Csnippet(rmeas),
+       covar = birthdat, tcovar = "time", zeronames = c("incid", "noise"),
+       statenames = c("S", "I", "R", "incid", "pop", "phase", "noise"),
+       paramnames = c("gamma", "mu", "popsize", "rho","theta","sigma",
+         "S.0", "I.0", "R.0", "b1", "b2", "b3", "iota"),
+       initializer = function(params, t0, ...) {
+         x0 <- c(S = 0, I = 0, R = 0, incid = 0, pop = 0,
+                 noise = 0, phase = 0)
+         fracs <- params[c("S.0", "I.0", "R.0")]
+         x0[1:3] <- round(params['popsize'] * fracs / sum(fracs))
+         x0[5] <- params['popsize']
+         x0
+       }, params = c(popsize = 500000, iota = 5, b1 = 6, b2 = 0.2,
+            b3 = -0.1, gamma = 26, mu = 1/50, rho = 0.1, theta = 100,
+            sigma = 0.2, S.0 = 0.055, I.0 = 0.002, R.0 = 0.94)) -> sir2
R> simulate(sir2, seed = 1914679908L) -> sir2
```

Fig. 9 shows the simulated data and latent states. The `sir2` object we've constructed here contains all the key elements of models used within the **pomp** to investigate cholera (King *et al.* 2008), measles (He *et al.* 2010), malaria (Bhadra *et al.* 2011), pertussis (Blackwood *et al.* 2013a; Lavine *et al.* 2013), pneumonia (Shrestha *et al.* 2013), and rabies (Blackwood *et al.* 2013b).

6. Conclusion

The **pomp** package is designed to be both a tool for data analysis based on POMP models and a sound platform for the development of inference algorithms. The model specification language provided by **pomp** is very general. Implementing a POMP model in **pomp** makes a wide range of inference algorithms available. Moreover, the separation of model from inference algorithm facilitates objective comparison of alternative models and methods. The examples demonstrated in this paper are relatively simple, but the package has been instrumental in a number of scientific studies (e.g., King *et al.* 2008; Bhadra *et al.* 2011; Shrestha *et al.* 2011; Earn *et al.* 2012; Roy *et al.* 2012; Shrestha *et al.* 2013; Blackwood *et al.* 2013a,b; Lavine *et al.* 2013; He *et al.* 2013; Bretó 2014). As a development platform, **pomp** is particularly

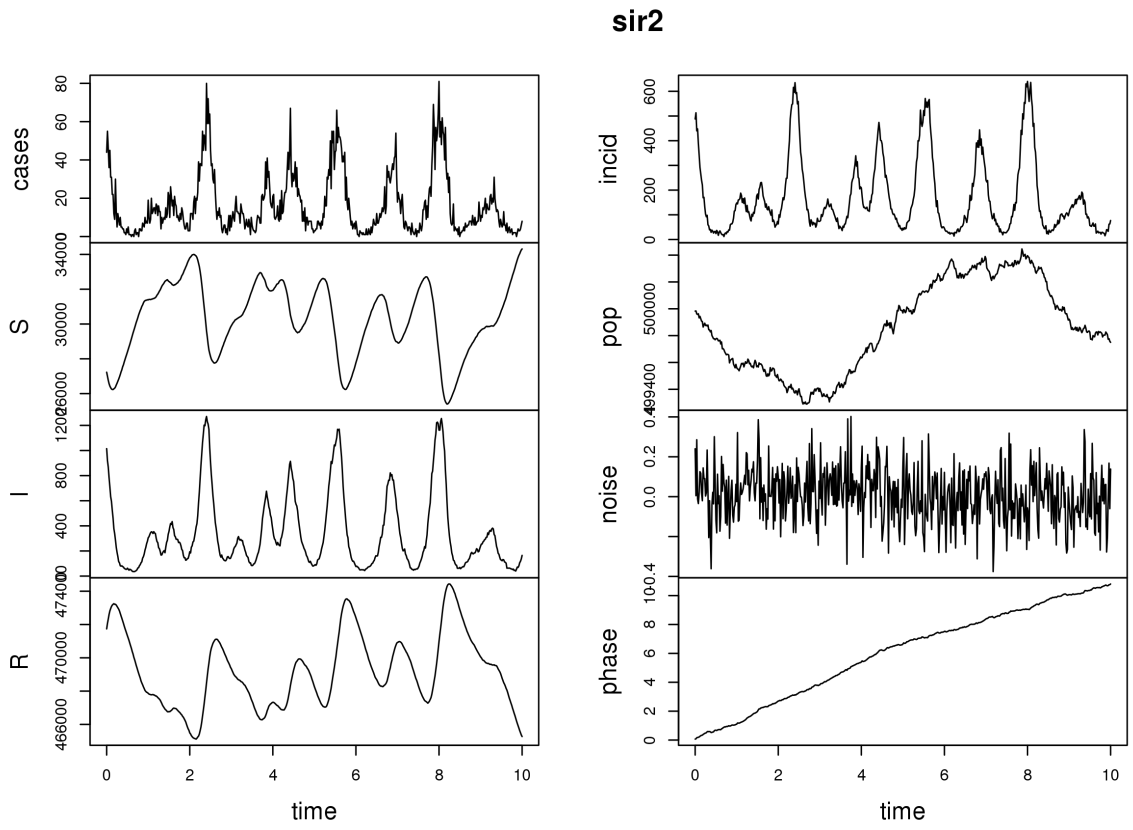

Figure 9: One realization of the SIR model with seasonal contact rate, imported infections, and extrademographic stochasticity in the force of infection.

convenient for implementing algorithms with the plug-and-play property, since models will typically be defined by their `rprocess` simulator, together with `rmeasure` and often `dmeasure`, but can accommodate inference methods based on other model components (e.g., `dprocess` and `skeleton`, the deterministic skeleton of the latent process). As an open-source project, the package readily supports expansion, and the authors invite community participation in the **pomp** project in the form of additional inference algorithms, improvements and extensions of existing algorithms, additional model/data examples, documentation contributions and improvements, bug reports, and feature requests.

Complex models and large datasets can challenge computational resources. With this in mind, key components of the **pomp** are written in C, and **pomp** provides facilities for users to write models either in R or, for the acceleration that typically proves necessary in applications, in C. Multi-processor computing also becomes necessary for ambitious projects. The two most common computationally intensive tasks are assessment of Monte Carlo variability and investigation of the role of starting values and other algorithmic settings on optimization routines. These analyses require only embarrassingly parallel computations and need no special discussion here.

The package contains more examples (via `pompExamples`), which can be used as templates for implementation of new models; the R and C code underlying these examples is provided with the package. In addition, **pomp** provides a number of interactive demos (via `demo`). Further documentation and an introductory tutorial are provided with the package and on the **pomp** website.

Acknowledgments Initial development of **pomp** was carried out as part of the *Inference for Mechanistic Models* working group supported from 2007 to 2010 by the National Center for Ecological Analysis and Synthesis, a center funded by the U.S. National Science Foundation (Grant DEB-0553768), the University of California, Santa Barbara, and the State of California. Participants were C. Bretó, S. P. Ellner, M. J. Ferrari, G. J. Gibson, G. Hooker, E. L. Ionides, V. Isham, B. E. Kendall, K. Koelle, A. A. King, M. L. Lavine, K. B. Newman, D. C. Reuman, P. Rohani and H. J. Wearing. As of this writing, the **pomp** development team is A. A. King, E. L. Ionides, and D. Nguyen. Financial support was provided by grants DMS-1308919, DMS-0805533, EF-0429588 from the U.S. National Science Foundation and by the Research and Policy for Infectious Disease Dynamics program of the Science and Technology Directorate, U.S. Department of Homeland Security and the Fogarty International Center, U.S. National Institutes of Health.

References

Andrieu C, Doucet A, Holenstein R (2010). "Particle Markov Chain Monte Carlo." *Journal of the Royal Statistical Society B*, **72**(3), 269–342. doi:10.1111/j.1467-9868.2009.00736.x.

Andrieu C, Roberts GO (2009). "The Pseudo-Marginal Approach for Efficient Computation." *The Annals of Statistics*, **37**(2), 697–725. doi:10.1214/07-AOS574.

Beaumont MA (2010). "Approximate Bayesian Computation in Evolution and Ecology."

Annual Review of Ecology, Evolution, and Systematics, **41**, 379–406. doi:10.1146/annurev-ecolsys-102209-144621.

Bhadra A (2010). "Discussion of 'Particle Markov Chain Monte Carlo Methods' by C. Andrieu, A. Doucet and R. Holenstein." *Journal of the Royal Statistical Society B*, **72**, 314–315. doi:10.1111/j.1467-9868.2009.00736.x.

Bhadra A, Ionides EL, Laneri K, Pascual M, Bouma M, Dhiman R (2011). "Malaria in Northwest India: Data Analysis via Partially Observed Stochastic Differential Equation Models Driven by Lévy Noise." *Journal of the American Statistical Association*, **106**(494), 440–451. doi:10.1198/jasa.2011.ap10323.

Blackwood JC, Cummings DAT, Broutin H, Iamsirithaworn S, Rohani P (2013a). "Deciphering the Impacts of Vaccination and Immunity on Pertussis Epidemiology in Thailand." *Proceedings of the National Academy of Sciences of the USA*, **110**(23), 9595–9600. doi:10.1073/pnas.1220908110.

Blackwood JC, Streicker DG, Altizer S, Rohani P (2013b). "Resolving the Roles of Immunity, Pathogenesis, and Immigration for Rabies Persistence in Vampire Bats." *Proceedings of the National Academy of Sciences of the USA*. doi:10.1073/pnas.1308817110.

Bretó C (2014). "On Idiosyncratic Stochasticity of Financial Leverage Effects." *Statistics & Probability Letters*, **91**, 20–26. ISSN 0167-7152. doi:10.1016/j.spl.2014.04.003.

Bretó C, He D, Ionides EL, King AA (2009). "Time Series Analysis via Mechanistic Models." *The Annals of Applied Statistics*, **3**, 319–348. doi:10.1214/08-AOAS201.

Cappé O, Godsill S, Moulines E (2007). "An Overview of Existing Methods and Recent Advances in Sequential Monte Carlo." *Proceedings of the IEEE*, **95**(5), 899–924. doi:10.1109/JPROC.2007.893250.

Chambers JM (1998). *Programming with Data*. Springer-Verlag, New York. ISBN 0-387-98503-4, URL http://cm.bell-labs.com/cm/ms/departments/sia/Sbook/.

Commandeur JJF, Koopman SJ, Ooms M (2011). "Statistical Software for State Space Methods." *Journal of Statistical Software*, **41**(1), 1–18. URL http://www.jstatsoft.org/v41/i01.

Earn DJD, He D, Loeb MB, Fonseca K, Lee BE, Dushoff J (2012). "Effects of School Closure on Incidence of Pandemic Influenza in Alberta, Canada." *The Annals of Internal Medicine*, **156**(3), 173–181. doi:10.7326/0003-4819-156-3-201202070-00005.

Ellner SP, Bailey BA, Bobashev GV, Gallant AR, Grenfell BT, Nychka DW (1998). "Noise and Nonlinearity in Measles Epidemics: Combining Mechanistic and Statistical Approaches to Population Modeling." *American Naturalist*, **151**(5), 425–440. doi:10.1086/286130.

Genolini C (2008). "A (Not So) Short Introduction to S4." *Technical report*, The R Project for Statistical Computing. URL http://cran.r-project.org/doc/contrib/Genolini-S4tutorialV0-5en.pdf.

Gillespie DT (1977). "Exact Stochastic Simulation of Coupled Chemical Reactions." *Journal of Physical Chemistry*, **81**(25), 2340–2361. doi:10.1021/j100540a008.

Gompertz B (1825). "On the Nature of the Function Expressive of the Law of Human Mortality, and on a New Mode of Determining the Value of Life Contingencies." *Philosophical Transactions of the Royal Society of London*, **115**, 513–583. doi:10.1098/rstl.1825.0026.

He D, Dushoff J, Day T, Ma J, Earn DJD (2013). "Inferring the Causes of the Three Waves of the 1918 Influenza Pandemic in England and Wales." *Proceedings of the Royal Society of London B*, **280**(1766), 20131345. doi:10.1098/rspb.2013.1345.

He D, Ionides EL, King AA (2010). "Plug-and-play Inference for Disease Dynamics: Measles in Large and Small Towns as a Case Study." *Journal of the Royal Society Interface*, **7**(43), 271–283. doi:10.1098/rsif.2009.0151.

Ionides EL (2011). "Discussion on "Feature Matching in Time Series Modeling" by Y. Xia and H. Tong." *Statistical Science*, **26**, 49–52. doi:10.1214/11-STS345C.

Ionides EL, Bhadra A, Atchadé Y, King AA (2011). "Iterated Filtering." *The Annals of Statistics*, **39**(3), 1776–1802. doi:10.1214/11-AOS886.

Ionides EL, Bretó C, King AA (2006). "Inference for Nonlinear Dynamical Systems." *Proceedings of the National Academy of Sciences of the USA*, **103**(49), 18438–18443. doi:10.1073/pnas.0603181103.

Johnson SG (2014). *The **NLopt** Nonlinear-Optimization Package*. Version 2.4.2, URL http://ab-initio.mit.edu/nlopt.

Kendall BE, Briggs CJ, Murdoch WW, Turchin P, Ellner SP, McCauley E, Nisbet RM, Wood SN (1999). "Why do Populations Cycle? A Synthesis of Statistical and Mechanistic Modeling Approaches." *Ecology*, **80**(6), 1789–1805. doi:10.1890/0012-9658(1999)080[1789:WDPCAS]2.0.CO;2.

Kendall BE, Ellner SP, McCauley E, Wood SN, Briggs CJ, Murdoch WW, Turchin P (2005). "Population Cycles in the Pine Looper Moth: Dynamical Tests of Mechanistic Hypotheses." *Ecological Monographs*, **75**(2), 259–276. URL http://www.jstor.org/stable/4539097.

King AA (2008). ***subplex**: Subplex Optimization Algorithm*. R package, version 1.1-4, URL http://subplex.r-forge.r-project.org.

King AA, Ionides EL, Bretó CM, Ellner SP, Ferrari MJ, Kendall BE, Lavine M, Nguyen D, Reuman DC, Wearing H, Wood SN (2014). ***pomp**: Statistical Inference for Partially Observed Markov Processes*. R package, version 0.53-1, URL http://pomp.r-forge.r-project.org.

King AA, Ionides EL, Pascual M, Bouma MJ (2008). "Inapparent Infections and Cholera Dynamics." *Nature*, **454**(7206), 877–880. doi:10.1038/nature07084.

Kitagawa G (1998). "A Self-organising State Space Model." *Journal of the American Statistical Association*, **93**, 1203–1215. doi:10.1080/01621459.1998.10473780.

Lavine JS, King AA, Andreasen V, Bjørnstad ON (2013). "Immune Boosting Explains Regime-Shifts in Prevaccine-Era Pertussis Dynamics." *PLoS ONE*, **8**(8), e72086. doi:10.1371/journal.pone.0072086.

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

Ratmann O, Andrieu C, Wiuf C, Richardson S (2009). "Model Criticism based on Likelihood-free Inference, with an Application to Protein Network Evolution." *Proceedings of the National Academy of Sciences of the USA*, **106**(26), 10576–10581. doi:10.1073/pnas.0807882106.

Reddingius J (1971). *Gambling for Existence. A Discussion of some Theoretical Problems in Animal Population Ecology*, volume 20 (Supplement). B. J. Brill, Leiden.

Reuman DC, Desharnais RA, Costantino RF, Ahmad OS, Cohen JE (2006). "Power Spectra Reveal the Influence Of Stochasticity on Nonlinear Population Dynamics." *Proceedings of the National Academy of Sciences of the USA*, **103**(49), 18860–18865. doi:10.1073/pnas.0608571103.

Revolution Analytics, Weston S (2014). ***foreach**: Foreach Looping Construct for R*. R package version 1.4.2, URL http://CRAN.R-project.org/package=foreach.

Ricker WE (1954). "Stock and Recruitment." *Journal of the Fisheries Research Board of Canada*, **11**, 559–623. doi:10.1139/f54-039.

Rowan T (1990). *Functional Stability Analysis of Numerical Algorithms*. Ph.D. thesis, Department of Computer Sciences, University of Texas at Austin.

Roy M, Bouma MJ, Ionides EL, Dhiman RC, Pascual M (2012). "The Potential Elimination of Plasmodium Vivax Malaria by Relapse Treatment: Insights from a Transmission Model and Surveillance Data from NW India." *PLoS Neglected Tropical Diseases*, **7**(1), e1979. doi:10.1371/journal.pntd.0001979.

Shaman J, Karspeck A (2012). "Forecasting Seasonal Outbreaks of Influenza." *Proceedings of the National Academy of Sciences of the USA*, **109**(50), 20425–20430. doi:10.1073/pnas.1208772109.

Shrestha S, Foxman B, Weinberger DM, Steiner C, Viboud C, Rohani P (2013). "Identifying the Interaction between Influenza and Pneumococcal Pneumonia Using Incidence Data." *Science Translational Medicine*, **5**(191), 191ra84. doi:10.1126/scitranslmed.3005982.

Shrestha S, King AA, Rohani P (2011). "Statistical Inference for Multi-Pathogen Systems." *PLoS Computational Biology*, **7**, e1002135. doi:10.1371/journal.pcbi.1002135.

Sisson SA, Fan Y, Tanaka MM (2007). "Sequential Monte Carlo without Likelihoods." *Proceedings of the National Academy of Sciences of the USA*, **104**(6), 1760–1765. doi:10.1073/pnas.0607208104.

Smith AA (1993). "Estimating Nonlinear Time-series Models using Simulated Vector Autoregression." *Journal of Applied Econometrics*, **8**(S1), S63–S84. doi:10.1002/jae.3950080506.

Toni T, Welch D, Strelkowa N, Ipsen A, Stumpf MP (2009). "Approximate Bayesian Computation Scheme for Parameter Inference and Model Selection in Dynamical Systems." *Journal of the Royal Society Interface*, **6**(31), 187–202. doi:10.1098/rsif.2008.0172.

Wan E, Van Der Merwe R (2000). "The Unscented Kalman Filter for Nonlinear Estimation." In *Adaptive Systems for Signal Processing, Communications, and Control*, pp. 153–158. doi:10.1109/ASSPCC.2000.882463.

Wood SN (2001). "Partially Specified Ecological Models." *Ecological Monographs*, **71**, 1–25. doi:10.1890/0012-9615(2001)071[0001:PSEM]2.0.CO;2.

Wood SN (2010). "Statistical Inference for Noisy Nonlinear Ecological Dynamic Systems." *Nature*, **466**(7310), 1102–1104. doi:10.1038/nature09319.

Xia Y, Tong H (2011). "Feature Matching in Time Series Modelling." *Statistical Science*, **26**(1), 21–46. doi:10.1214/10-STS345.

Ypma J (2014). ***nloptr**: R Interface to **NLopt**.* R package, version 1.0.0, URL http://cran.r-project.org/web/packages/nloptr/.

Affiliation:

Aaron A. King
Departments of Ecology & Evolutionary Biology and Mathematics
Center for the Study of Complex Systems
University of Michigan
48109 Michigan, USA
E-mail: kingaa@unich.edu
URL: http://kinglab.eeb.lsa.umich.edu/

Dao Nguyen
Department of Statistics
University of Michigan
48109 Michigan, USA
E-mail: nguyenxd@unich.edu

Edward Ionides
Department of Statistics
University of Michigan
48109 Michigan, USA
E-mail: ionides@unich.edu
URL: http://www.stat.lsa.umich.edu/~ionides

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume VV, Issue II *Submitted:* yyyy-mm-dd
MMMMMM YYYY *Accepted:* yyyy-mm-dd