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

Aaron A. King
University of Michigan

Dao Nguyen
University of Michigan

Edward L. Ionides
University of Michigan

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**.
*This document is an updated version of Red*Journal of Statistical Software* **69**(12): 1–43.

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 (**?**). It has been a challenge to provide a software environment that can effectively handle broad classes of POMP models and take advantage of the wide range of statistical methodologies that have been proposed for such models. The **pomp** software package (**?**) differs from previous approaches by providing a general and abstract representation of a POMP model. Therefore, algorithms implemented within **pomp** are necessarily applicable to arbitrary POMP models. Moreover, 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. However, since **pomp** is designed for general POMP models, methods that exploit additional model structure have yet to be implemented. In particular, when linear, Gaussian approximations are adequate for one's purposes, or when the latent process takes values in a small, discrete set, methods that exploit

these additional assumptions to advantage, such as the extended and ensemble Kalman filter methods or exact hidden Markov model methods, are available, but not yet as part of **pomp**. It is the class of nonlinear, non-Gaussian POMP models with large state spaces upon which **pomp** is focused.

A POMP model may be characterized by the transition density for the Markov process and the measurement density[1]. 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* (**??**). Plug-and-play methods can be employed once one has "plugged" a model simulator into the inference machinery. Since many POMP models of scientific interest are relatively easy to simulate, 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 often be achieved 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, which can be substantial. Nevertheless, plug-and-play methods implemented using **pomp** have proved capable for state-of-the-art inference problems (e.g., **??????????**). The recent surge of interest in plug-and-play methodology for POMP models includes the development of nonlinear forecasting (**?**), iterated filtering (**??**), ensemble Kalman filtering (**?**), approximate Bayesian computation (ABC; **?**), particle Markov chain Monte Carlo (PMCMC; **?**), probe matching (**?**), and synthetic likelihood (**?**). 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. **??** defines mathematical notation for POMP models and relates this to their representation as objects of class '`pomp`' in the **pomp** package. **??** introduces several of the statistical methods currently implemented in **pomp**. **??** constructs and explores a simple POMP model, demonstrating the use of the available statistical methods. **??** illustrates the implementation of more complex POMPs, using a model of infectious disease transmission as an example. Finally, **??** 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 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,

[1] We use the term "density" in this article to encompass both the continuous and discrete cases. Thus, in the latter case, i.e., when state variables and/or measured quantities are discrete, one could replace "probability density function" with "probability mass function".

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.

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 determined 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 $1 \le n \le N$. In particular, we have

$$f_{X_{0:N},Y_{1:N}}(x_{0:N}, y_{1:N};\theta) = f_{X_0}(x_0;\theta)\ \prod_{n=1}^{N} f_{X_n|X_{n-1}}(x_n|x_{n-1};\theta)\ f_{Y_n|X_n}(y_n|x_n;\theta). \qquad (1)$$

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 (**??**) 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 class '`pomp`' class use these components to carry out computations on the model. **??** 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 probabilities $f_{X_n|X_{n-1}}(x_n \,|\, x_{n-1}\,;\theta)$ and measurement densities $f_{Y_n|X_n}(y_n \,|\, x_n\,;\theta)$. Not all of these arguments must be supplied for any specific computation. In particular, plug-and-play method-

ology 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 class '`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 will use a simple example. The **?** model can be constructed via

```
R> library("pomp")
R> ex_gompertz <- 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 **??**. One can view the components of `gompertz` listed in **??** by executing

```
R> obs(ex_gompertz)
R> states(ex_gompertz)
R> as.data.frame(ex_gompertz)
R> plot(ex_gompertz)
R> timezero(ex_gompertz)
R> time(ex_gompertz)
R> coef(ex_gompertz)
```

Executing `pompExample()` 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. An example using covariates is given in **??**.

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`', the package provides a variety of algorithms to assist with these data analysis goals. **??** 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. However, the package developers welcome contributions and collaborations to further expand **pomp**'s functionality in non-plug-and-play directions also. In the remainder of this section, we describe and discuss several inference 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,

$$\ell(\theta) = \sum_{n=1}^{N} \ell_{n|1:n-1}(\theta), \tag{2}$$

where

$$\ell_{n|1:n-1}(\theta) = \log f_{Y_n|Y_{1:n-1}}(y^*_n \,|\, y^*_{1:n-1}\,;\theta), \tag{3}$$

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

$$\ell_{n|1:n-1}(\theta) = \log \int f_{Y_n|X_n}(y^*_n|x_n\,;\theta) f_{X_n|Y_{1:n-1}}(x_n \,|\, y^*_{1:n-1}\,;\theta)\, dx_n \tag{4}$$

(a) Plug-and-play

	Frequentist	Bayesian
Full information	Iterated filtering (`mif`, **??**)	PMCMC (`pmcmc`, **??**)
Feature-based	Nonlinear forecasting (`nlf`, **??**), synthetic likelihood (`probe.match`, **??**)	ABC (`abc`, **??**)

(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**, the 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.

Algorithm 1: Sequential Monte Carlo (SMC, or particle filter): `pfilter(P, Np=`J`)`, using notation from **??** where `P` is a class '`pomp`' object with definitions 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.

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

for n *in* $1{:}N$ **do**

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

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

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

  Apply **??** to select indices $k_{1:J}$ with $\mathbb{P}\,[k_j = m] = \tilde{w}(n,m)$.

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

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

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 $1{:}N$.

complexity: $\mathcal{O}(J)$

Algorithm 2: Systematic resampling: **??** of **??**.

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

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

output: Resampling indices, $k_{1:J}$.
complexity: $\mathcal{O}(J)$

(cf. **??**). 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 **??**), is also known as the 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 (**??**) mutates the particles in the swarm and the filtering step (**??**) corresponds to selection. SMC is implemented in **pomp** in the `pfilter` function. The basic particle filter in **??** possesses the plug-and-play property. Many variations and elaborations to SMC have been proposed; these may improve numerical performance in appropriate situations (**?**) but typically lose the plug-and-play property. **?**, **?**, and **?** have written excellent introductory tutorials on the particle filter and particle methods more generally.

Basic SMC methods fail when an observation is extremely unlikely given the model. This leads to the situation that at most a few particles are consistent with the observation, in which case the effective sample size (**?**) of the Monte Carlo sample is small and the particle filter is said to suffer from *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 **??** of **??**, systematic resampling (**??**) is used in preference to multinomial resampling. **??** reduces Monte Carlo variability while resampling with the proper marginal probability. In particular, if all the particle weights are equal then **??** has the appropriate behavior of leaving the particles unchanged. As pointed out by **?**, stratified resampling performs better than multinomial sampling and **??** is in practice comparable in performance to stratified resampling and somewhat faster.

3.2. Iterated filtering

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 **??** where `P` is a class 'pomp' object with defined `rprocess`, `dmeasure`, `init.state`, and `obs` components.

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: $[\Theta^F_{0,j}]_i \sim \text{Normal}\left([\theta_{m-1}]_i, (Ca^{m-1}\sigma_i)^2\right)$ for i in $1{:}p$, j in $1:J$.
- Initialize states: simulate $X^F_{0,j}\sim f_{X_0}(\,\cdot\,;\Theta^F_{0,j})$ for j in $1{:}J$.
- Initialize filter mean for parameters: $\bar\theta_0=\theta_{m-1}$.
- Define $[V_1]_i=(C^2+1)(a^{m-1}\sigma_i)^2$.
- **for** n *in* $1:N$ **do**
  - Perturb parameters: $\left[\Theta^P_{n,j}\right]_i \sim \text{Normal}\left(\left[\Theta^F_{n-1,j}\right]_i, (a^{m-1}\sigma_i)^2\right)$ for $i\notin I$, j in $1:J$.
  - Simulate prediction particles: $X^P_{n,j}\sim f_{X_n|X_{n-1}}(\,\cdot\,|X^F_{n-1,j};\Theta^P_{n,j})$ 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 **??** to select indices $k_{1:J}$ with $\mathbb{P}\left[k_u=j\right]=\tilde w\,(n,j)$.
  - 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: $\left[\bar\theta_n\right]_i=\sum_{j=1}^J\tilde w(n,j)\left[\Theta^P_{n,j}\right]_i$ for $i\notin I$.
  - Prediction variance: $[V_{n+1}]_i=(a^{m-1}\sigma_i)^2+\sum_j\tilde w(n,j)\left([\Theta^P_{n,j}]_i-[\bar\theta_n]_i\right)^2$ for $i\notin I$.
- **end**
- Update non-IVPs: $[\theta_m]_i=[\theta_{m-1}]_i+[V_1]_i\sum_{n=1}^N[V_n]_i^{-1}\left([\bar\theta_n]_i-[\bar\theta_{n-1}]_i\right)$ for $i\notin I$.
- Update IVPs: $[\theta_m]_i=\frac{1}{J}\sum_j\left[\Theta^F_{L,j}\right]_i$ for $i\in I$.

end

output: Monte Carlo maximum likelihood estimate, θ_M.

complexity: $\mathcal{O}(JM)$

Iterated filtering techniques maximize 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 one. 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 adding diversity to the population of particles.

A3. The additional variability can be exploited to explore the likelihood surface and estimate the gradient of the (smoothed) likelihood, based on the same filtering procedure that is required to estimate 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. By default, `mif` carries out the procedure of **?**. The improved iterated filtering algorithm (IF2) of **?** has shown promise. A limited version of IF2 is available via the `method = "mif2"` option; a full version of this algorithm will be released soon. In all iterated filtering methods, by analogy with annealing, the random walk intensity can be called a temperature, which is decreased according to a prescribed cooling schedule. One strives to ensure that the algorithm will freeze at the maximum of the likelihood as the temperature approaches zero.

The perturbations on the parameters in **????** of **??** follow a normal distribution, with each component, $[\theta]_i$, of the parameter vector perturbed independently. Neither normality nor independence are necessary for iterated filtering, but, rather than varying the perturbation distribution, one can transform the parameters to make these simple algorithmic choices reasonable.

?? 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 **??** perturbs these parameters only at time zero.

?????????? of **??** are exactly an application of SMC (**??**) 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 (**???**) but iterated filtering is distinguished by having theoretical justification for convergence to the maximum likelihood estimate (**?**).

Algorithm 4: Particle Markov chain Monte Carlo: `pmcmc(P, start=`θ_0`, Nmcmc=`M`, Np=`J`, proposal=`q`)`, using notation from **??** where `P` is a class 'pomp' object with defined methods for `rprocess`, `dmeasure`, `init.state`, `dprior`, and `obs`. The supplied `proposal` samples from a symmetric, but otherwise arbitrary, MCMC proposal distribution, $q(\theta^P \,|\, \theta)$.

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 $f_{Y_n|X_n}(y_n \,|\, x_n\,;\theta)$; simulator for $q(\theta^P \,|\, \theta)$; 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 **??** with J particles.

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

3 Draw a parameter proposal, θ^P_m, from the proposal distribution: $\Theta^P_m \sim q\,(\,\cdot\,|\,\theta_{m-1})$.

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

5 Generate $U \sim \text{Uniform}(0,1)$.

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

7 **end**

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

complexity: $\mathcal{O}(JM)$

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 (**?**). 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 into the Metropolis-Hastings update procedure to give an algorithm targeting the desired posterior distribution for the parameters (**?**). PMMH is implemented in `pmcmc`, as described in **??**. In part because it gains only a single likelihood evaluation from each particle filtering operation, PMCMC can be computationally relatively inefficient (**??**). Nevertheless, its invention introduced the possibility of full-information plug-and-play Bayesian inferences in some situations where they had been unavailable.

3.4. Synthetic likelihood of summary statistics

Some motivations to estimate parameters 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 (**?**).

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 (**??**).

Algorithm 5: Synthetic likelihood evaluation: `probe(P, nsim=`J`, probes=`$\mathbb{S}$`)`, using notation from **??** where `P` is a class '`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: $Y^j_{1:N} \sim f_{Y_{1:N}}(\,\cdot\,;\theta)$ for j in $1{:}J$.

3 Compute simulated probes: $s_{ij} = \mathbb{S}_i(Y^j_{1:N})$ 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:

$$\hat{\ell}_{\mathbb{S}}(\theta) = -\frac{1}{2}\,(s^*-\mu)^\top \Sigma^{-1}(s^*-\mu) - \frac{1}{2}\,\log|\Sigma| - \frac{d}{2}\,\log(2\pi). \tag{5}$$

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

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 (**?**).

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.

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 B**??**, one aims to find summary statistics which are as close as possible to sufficient statistics for the unknown parameters. Goals B**??** and B**??** deliberately look for features which discard information from the data; in this context the features have been called probes (**?**). The features are denoted by a collection of functions, $\mathbb{S} = (\mathbb{S}_1, \ldots, \mathbb{S}_d)$, where each $\mathbb{S}_i$ maps an observed time series to a real number. We write $S = (S_1, \ldots, S_d)$ 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, **?** 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` (**??**). 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

Algorithm 6: Approximate Bayesian computation: `abc(P, start=`θ_0`, Nmcmc=`M`, probes=`$\mathbb{S}$`, scale=`$\tau_{1:d}$`, proposal=`q`, epsilon=`ϵ`)`, using notation from **??**, where `P` is a class '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 $f_{Y_n|X_n}(y_n\,|\,x_n\,;\theta)$; simulator for $q(\theta^P\,|\,\theta)$; data, $y^*_{1:N}$; number of proposals, M; vector of probes, $\mathbb{S}=(\mathbb{S}_1,\dots,\mathbb{S}_d)$; 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 Draw a parameter proposal, θ^P_m, from the proposal distribution: $\Theta^P_m\sim q\,(\,\cdot\,|\,\theta_{m-1})$.
4 Simulate dataset: $Y_{1:N}\sim f_{Y_{1:N}}(\,\cdot\,;\theta^P_m)$.
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 $\theta_m=\begin{cases}\theta^P_m, & \text{if } \sum_{i=1}^d\left(\frac{s_i-s^*_i}{\tau_i}\right)^2<\epsilon^2 \text{ and } U<\frac{f_\Theta(\theta^P_m)}{f_\Theta(\theta_{m-1})},\\ \theta_{m-1}, & \text{otherwise.}\end{cases}$
8 **end**

output: Samples, $\theta_{1:M}$, representing the posterior distribution, $f_{\Theta|S_{1:d}}(\theta\,|\,s^*_{1:d})$.
complexity: Nominally $\mathcal{O}(M)$, but performance will depend on the choice of ϵ, τ_i, and σ_i, as well as on the choice of probes $\mathbb{S}$.

substantial difference between the probe approaches of **?** and **?**. Numerical optimization of the synthetic likelihood is implemented by `probe.match`, which offers the choice of either the subplex method (**??**) or any method provided by `optim` or the **nloptr** package (**??**).

3.5. Approximate Bayesian computation (ABC)

ABC algorithms are Bayesian feature-matching techniques, comparable to the frequentist 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 **??**. 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 B**??** of **??** (**???**). Points B**??** and B**??** also apply (**?**).

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 (**?**). 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

$$\hat{\ell}^{ABC}_{\mathbb{S}}(\theta)=\begin{cases}\epsilon^{-d}B_d^{-1}\prod_{i=1}^d\tau_i, & \text{if } \sum_{i=1}^d\left(\frac{s_i-s^*_i}{\tau_i}\right)^2<\epsilon^2,\\ 0, & \text{otherwise,}\end{cases}\tag{6}$$

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=`$c_{1:L}$`)`. Using notation from **??**, `P` is a class '`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: $Y_{1:(B+J)} \sim f_{Y_{1:(B+J)}}(\,\cdot\,;\theta)$.

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

$$\hat{\sigma}^2 = \frac{1}{J}\sum_{n=B+1}^{B+J} \big[Y_n - H(Y_{n-c_1}, Y_{n-c_2}, \ldots, Y_{n-c_L})\big]^2. \tag{7}$$

8 Compute the simulated quasi log likelihood:

$$\hat{\ell}_Q(\theta) = -\frac{N-\overline{c}}{2}\log 2\pi\hat{\sigma}^2 - \sum_{n=1+\overline{c}}^{N} \frac{\big[y^*_n - H(y^*_{n-c_1}, y^*_{n-c_2}, \ldots, y^*_{n-c_L})\big]^2}{2\hat{\sigma}^2}, \tag{8}$$

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

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

complexity: $\mathcal{O}(B) + \mathcal{O}(J)$

where B_d is the volume of the d-dimensional unit ball and τ_i is a scaling chosen for the ith feature. The likelihood approximation in **??** differs from the synthetic likelihood in **??** in that only a single simulation is required. As ϵ becomes small, the bias in **??** decreases but the Monte Carlo variability increases. The ABC implementation `abc` (presented in **??**) is a random walk Metropolis implementation of ABC-MCMC (Algorithm 3 of **?**). In the same style as iterated filtering and PMCMC, we assume a Gaussian random walk in parameter space; the package supports alternative choices of the proposal distribution.

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 **??** 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, $f_{X_n|Y_{1:n-1}}(x_n \,|\, y^*_{1:n-1}\,;\theta)$. By contrast, NLF uses simulations to fit a linear regression of Y_n on the L variables Y_{n-c_1}, …, Y_{n-c_L}, for some choice of positive lags $c_{1:L}$. 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 (**?**). The implementation in `nlf` maximizes the quasi-likelihood computed in **??**, using the subplex method (**??**) or any other optimizer offerered by `optim`. The construction of the quasi-likelihood in `nlf` follows the specific recommendations of **?**. In particular, the choice of radial basis functions, f_k, in **??** and the specification of m_k and s in **????** were proposed by **?** based on trial and error. The quasi-likelihood is mathematically most similar to a likelihood when $\min(c_{1:L}) = 1$, so that $\ell_Q(\theta)$ approximates the factorization of the likelihood in **??**. With this in mind, it is natural to choose contiguous lags $c_{1:L} = 1 : L$. However, **?** found that a two-step prediction criterion, with $\min(c_{1:L}) = 2$, 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 the synthetic likelihood and ABC (**????**):

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. B**??** in **??**).

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 than the likelihood to model misspecification (**?**) (cf. B**??**).

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 (**?**) (cf. B**??**).

C4. NLF does not require that the model be Markovian (cf. B**??**), although the **pomp** implementation, `nlf`, does.

C5. NLF is doubly plug-and-play (cf. B**??**).

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 low-dimensional time series observations.

NLF can be viewed as an estimating equation method, and so standard errors can be computed by standard sandwich estimator or bootstrap techniques (**?**). 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 will first demonstrate the basics of **pomp** using simple discrete-time models, the Gompertz and Ricker models for population growth (**??**). The Ricker model will be introduced in **??** and used in **??**; the remainder of **??** 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

$$X_{t+\Delta t} = K^{1-e^{-r\,\Delta t}}\, X_t^{e^{-r\,\Delta t}}\, \varepsilon_t. \tag{9}$$

In **??**, 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 will assume that the population density is observed with errors in measurement that are lognormally distributed:

$$\log Y_t \;\sim\; \text{Normal}\left(\log X_t, \tau^2\right). \tag{10}$$

Taking a logarithmic transform of **??** gives

$$\log X_{t+\Delta t} \;\sim\; \text{Normal}\left(\left(1-e^{-r\,\Delta t}\right)\, \log K + e^{-r\,\Delta t}\, \log X_t, \sigma^2\right). \tag{11}$$

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; 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 will 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 will 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 (**??**).

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

The translation from the mathematical description (**??**) 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` specifies the time-step size. In this case, the time-step will be 1 unit; we will see below how this is specified.

Next, we will implement a simulator for the observation process, **??**.

```
R> gompertz.meas.sim <- function(X, tau, ...) {
+      c(Y = rlnorm(n = 1, meanlog = log(X), sdlog = tau))
+  }
```

Again the translation from the measurement model **??** 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 (**??**).

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

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

We will need this later on for inference using `pfilter`, `mif` and `pmcmc`. When the function `gompertz.meas.dens` is called, `y` will contain the observation at time `t`, `x` and `params` will be as before, and the parameter `log` will indicate whether the likelihood (`log = FALSE`) or the log likelihood (`log = TRUE`) is required.

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",
+      t0 = 0, rprocess = discrete_time(step.fun = gompertz.proc.sim, delta.t = 1),
+      rmeasure = gompertz.meas.sim, paramnames = c("r", "K", "sigma", "tau",
+          "X_0"), statenames = c("X"), obsnames = c("Y"))
```

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 will 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 have 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 manyfold.

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 us take our parameter vector to be

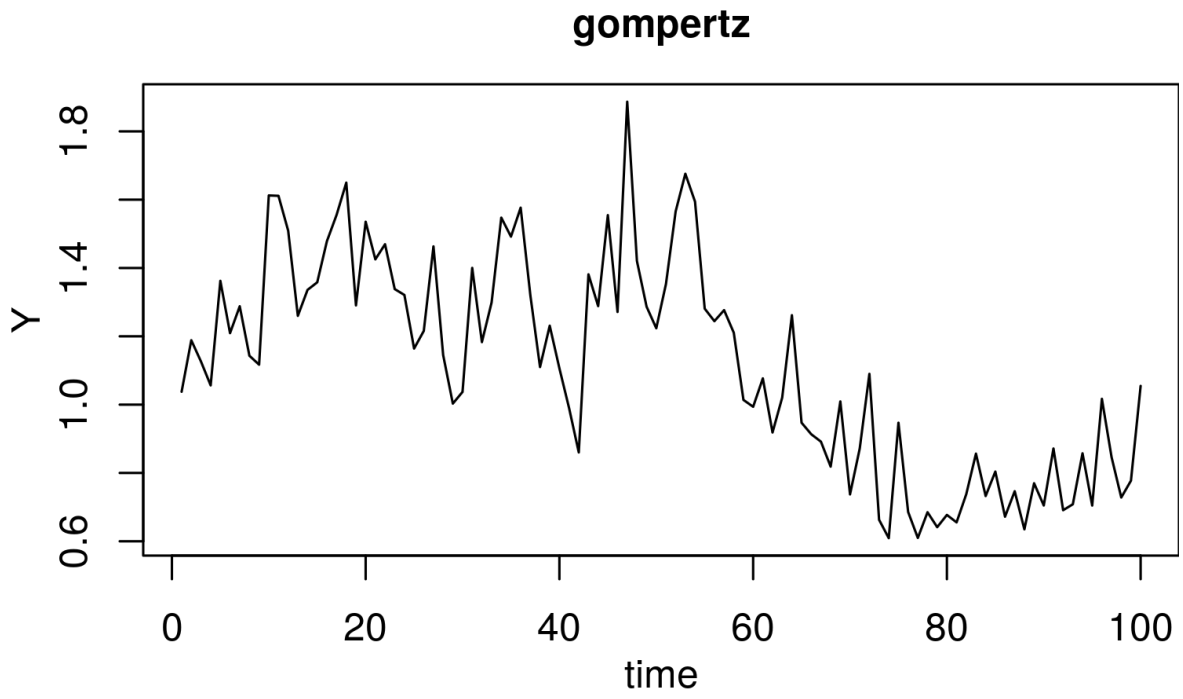

Figure 1: Simulated data from the Gompertz model (**????**). This figure shows the result of executing `plot(gompertz, variables = "Y")`.

```
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, rinit = function(X_0, ...) {
+      c(X = X_0)
+  })
```

Now `gompertz` is identical to what it was before, except that the missing data have been replaced by simulated data. The parameter vector (`theta`) at which the simulations were performed has also been saved internally to `gompertz`. We can plot the simulated data via

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

?? shows the results of this operation.

4.2. Computing the likelihood using SMC

As discussed in **??**, 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 fold it into the class '`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 class '`pomp`' object `gompertz` in every way identical to the one we had before, but with the measurement-model density function `dmeasure` now specified.

To estimate the likelihood of the data, we can use the function `pfilter`, an implementation of **??**. We must decide how many concurrent realizations (*particles*) to use: the larger the number of particles, the smaller the Monte Carlo error but the greater the computational 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] 35.74641
```

Since the true parameters (i.e., the parameters that generated the data) are stored within the class '`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 us 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] 24.89781
```

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 35.75. 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 will demonstrate this in **??**.

4.3. Maximum likelihood estimation via iterated filtering

Let us use the iterated filtering approach described in **??** to obtain an approximate maximum likelihood estimate for the data in `gompertz`. Since the parameters of **????** are constrained to be positive, when estimating, we transform them to a scale on which they are unconstrained. The following encodes such a transformation and its inverse:

```
R> gompertz.log.tf <- function(X_0, r, K, sigma, tau, ...) c(X_0 = log(X_0),
+      r = log(r), K = log(K), sigma = log(sigma), tau = log(tau))
R> gompertz.exp.tf <- function(X_0, r, K, sigma, tau, ...) c(X_0 = exp(X_0),
+      r = exp(r), K = exp(K), sigma = exp(sigma), tau = exp(tau))
```

We add these to the existing class '`pomp`' object via:

```
R> self_defined_gompertz <- pomp(gompertz, params=coef(gompertz), partrans = parameter_tra
```

The following code initializes the iterated filtering algorithm at several starting points around `theta.true` and estimates the parameters r, τ, and σ.

```
R> estpars <- c("r", "sigma", "tau")
R> library("foreach")
R> mif1 <- foreach(i = 1:10, .combine = c) %dopar% {
+    theta.guess <- theta.true
+    theta.guess[estpars] <- rlnorm(n = length(estpars),
+        meanlog = log(theta.guess[estpars]), sdlog = 1)
+    mif2(self_defined_gompertz, Nmif = 100, params = theta.guess,
+        Np = 2000, cooling.fraction = 0.7,
+        rw.sd = rw.sd(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 have set `transform = TRUE` in the call to `mif` above: this causes the parameter transformations we have specified to be applied to enforce the positivity of parameters. Note also that we have used the **foreach** package (**??**) to parallelize the computations.

Each of the 10 `mif` runs ends up with a different point estimate (**??**). 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

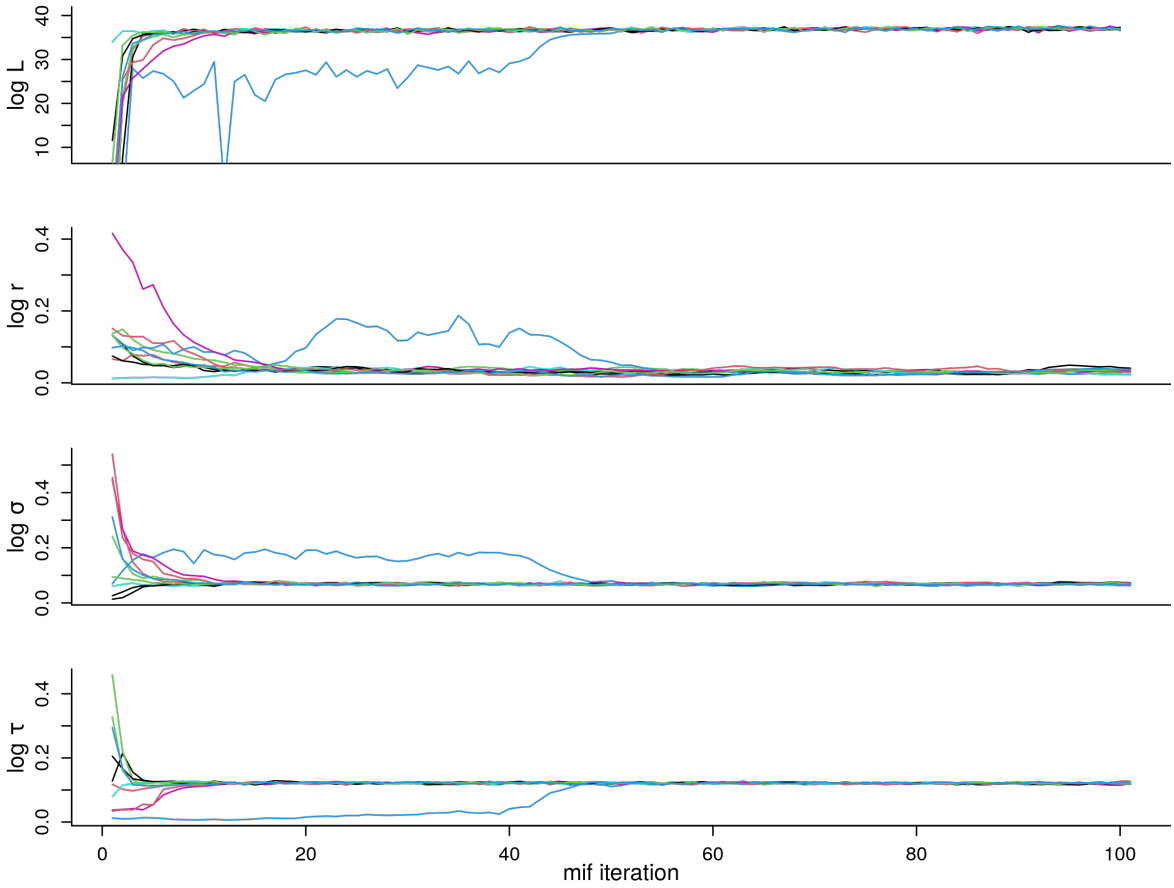

Figure 2: Convergence plots can be used to help diagnose convergence of the iterated filtering (IF) algorithm. These and additional diagnostic plots are produced when `plot` is applied to a class '`mif`' or class '`mifList`' object.

estimate of the likelihood; therefore, we will 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(self_defined_gompertz)
R> loglik.true <- replicate(n = 10, logLik(pfilter(self_defined_gompertz,
+      Np = 20000)))
R> loglik.true <- logmeanexp(loglik.true, se = TRUE)
```

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 maximum likelihood estimate (MLE). Taking advantage

	r	σ	τ	$\hat{\ell}$	s.e.	ℓ
Truth	0.1000	0.1000	0.1000	35.98	0.03	36.01
`mif` MLE	0.0324	0.0722	0.1180	37.85	0.04	37.84
Exact MLE	0.0322	0.0694	0.1170	37.89	0.02	37.88

Table 3: Results of estimating parameters r, σ, and τ of the Gompertz model (**????**) by maximum likelihood using iterated filtering (**??**), compared with the exact MLE and with the true value of the parameter. The first three columns show the estimated values of the three parameters. The next two columns show the log likelihood, $\hat{\ell}$, estimated by SMC (**??**) and its standard error, respectively. The exact log likelihood, ℓ, is shown in the rightmost column. An ideal likelihood-ratio 95% confidence set, not usually computationally available, includes all parameters having likelihood within `qchisq(0.95, df = 3)/2 = 3.91` of the exact MLE. We see that both the `mif` MLE and the truth are in this set. In this example, the `mif` MLE is close to the exact MLE, so it is reasonable to expect that profile likelihood confidence intervals and likelihood ratio tests constructed using the `mif` MLE have statistical properties similar to those based on the exact MLE.

of the Gompertz model's tractability, we also use the Kalman filter to maximize the exact log likelihood, ℓ, and evaluate it at the estimated MLE obtained by `mif`. The resulting estimates are shown in **??**. Usually, the last row and column of **??** 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 random walk Metropolis-Hastings 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(self_defined_gompertz)/10, max = coef(self_defined_gompe
```

```
R> gompertz.dprior <- function (r, K, sigma, tau, X_0, ..., log) {
+    f <- sum(dunif(c(r, K, sigma, tau, X_0), min = hyperparams$min, max = hyperparams$max
+                   log = TRUE))
+    if (log) f else exp(f)
+  }
```

The PMCMC algorithm described in **??** 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. For the same price, iterated filtering obtains, in addition, an estimate of the derivative and a probable improvement of the parameters. For this reason, PMCMC is relatively inefficient

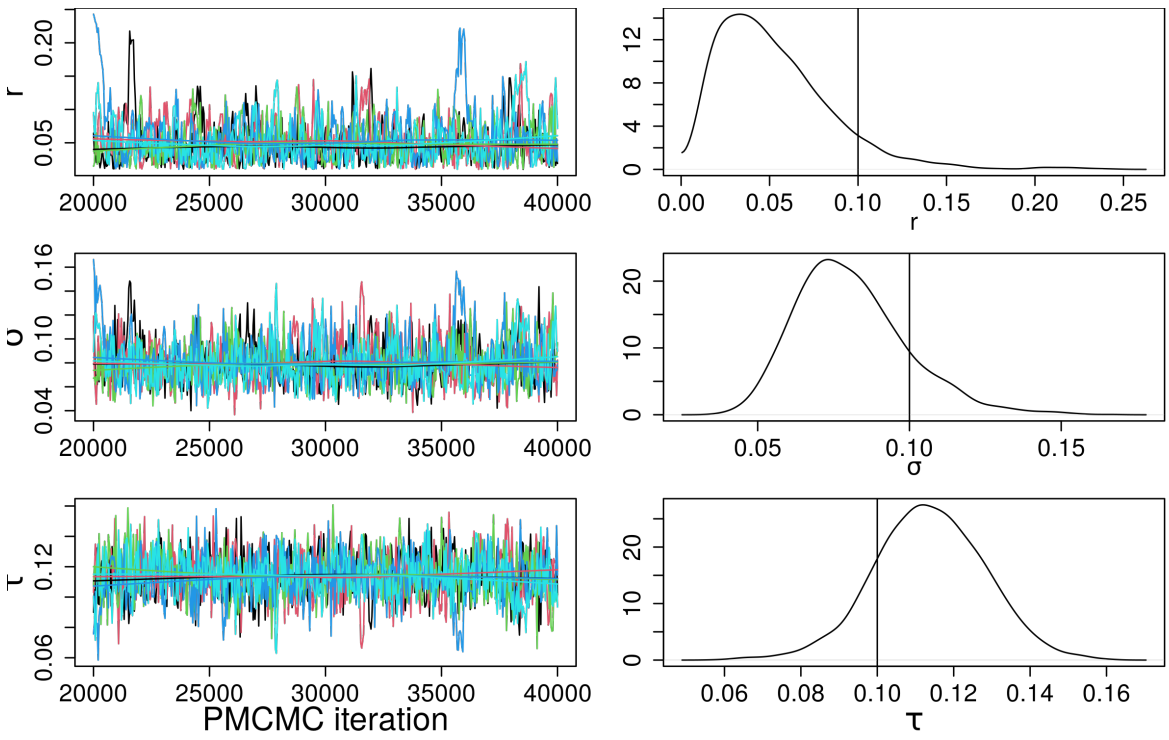

Figure 3: Diagnostic plots for the PMCMC algorithm. The trace plots in the left column show the evolution of 5 independent MCMC chains after a burn-in period of length 20000. Kernel density estimates of the marginal posterior distributions are shown in the right column. The effective sample size of the 5 MCMC chains combined is lowest for the r variable, being equal to 560: 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.

at traversing parameter space. When Bayesian inference is the goal, it is therefore 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 using a multivariate normal random walk proposal (with diagonal variance-covariance matrix, see `?mvn.diag.rw`).

```
R> pmcmc1 <- foreach(i = 1:5, .combine = c) %dopar% {
+    pmcmc(self_defined_gompertz, dprior = gompertz.dprior, params = theta.mif,
+          Nmcmc = 40000, Np = 100,
+          proposal = mvn.diag.rw(c(r = 0.01, sigma = 0.01, tau = 0.01)))
+  }
```

Comparison with the analysis of **??** reinforces the observation of **?** 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 convergence cannot be reliably assessed on the basis of a single chain. To verify the convergence of the approach or to compare the performance with other approaches, we can use diagnostic plots produced by the `plot` methods (see **??**).

4.5. A second example: The Ricker model

In **??**, we will illustrate probe matching (see **??**) using a stochastic version of the Ricker map (**?**). We switch models to allow direct comparison with **?**, whose synthetic likelihood computations are reproduced below. In particular, the results of **??** 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 **?**. We will 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

$$N_{t+1} = r\,N_t\,\exp(-N_t + e_t), \qquad e_t \sim \text{Normal}\left(0, \sigma^2\right). \tag{12}$$

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

$$Y_t \sim \text{Poisson}(\phi\,N_t), \tag{13}$$

where ϕ is a scaling parameter. As before, we will 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 manyfold 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 will write snippets for the `rprocess`, `rmeasure`, and `dmeasure` components.

```
R> ricker.sim <- "
+     e = rnorm(0, sigma);
+     N = r * N * exp(-c * 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> log.trans <- "
+     T_c = log(c);
+     T_r = log(r);
+     T_sigma = log(sigma);
+     T_phi = log(phi);
+     T_N_0 = log(N_0);"
R> exp.trans <- "
+     c = exp(T_c);
+     r = exp(T_r);
+     sigma = exp(T_sigma);
+     phi = exp(T_phi);
+     N_0 = exp(T_N_0);"
```

Note that in the foregoing C snippets, the prefix `T` designates the transformed version of the parameter. A full set of rules for using `Csnippet`s, including illustrative examples, is given in the package help system (`?Csnippet`).

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

```
R> self_defined_ricker <- pomp(data = data.frame(time = seq(0, 50, by = 1), y = NA),
+       rprocess = discrete_time(step.fun = Csnippet(ricker.sim),
+         delta.t = 1), rmeasure = Csnippet(ricker.rmeas),
+       dmeasure = Csnippet(ricker.dmeas),
+       partrans = parameter_trans(toEst = Csnippet(log.trans), fromEst = Csnippet(exp.tra
+       paramnames = c("r", "c", "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, c=1, N.0 = 7, e.0 = 0))
R> self_defined_ricker <- simulate(self_defined_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 **?**. 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(self_defined_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 **??**.] Here, we will apply `probe` to the Ricker model at the true parameters and at a wild guess.

```
R> pb.truth <- probe(self_defined_ricker, probes = plist, nsim = 1000, seed = 1066L)
R> guess <- c(r = 20, sigma = 1, phi = 20, N.0 = 7, e.0 = 0, c = 1)
R> pb.guess <- probe(self_defined_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 **??**. Among the quantities returned by `summary` is the synthetic likelihood (**??**). 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> pobj <- probe_objfun(pb.guess, est = c("r", "sigma", "phi"), seed = 1066L)
R>
R> library(subplex)
R>
R> subplex(fn = pobj, par = coef(pb.guess, c("r", "sigma", "phi"), transform = TRUE),
+      control = list(reltol = 1e-08)) -> pm
```

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 list of probes are stored internally in `pb.guess`, which is why we need not specify them explicitly here. While `probe.match` provides substantial flexibility in the choice of the 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.

To put the synthetic likelihood approach into context, let us compare the results of estimating the Ricker model parameters using probe-matching and using iterated filtering (IF), which is based on the likelihood. The following code runs 600 IF iterations starting at `guess`:

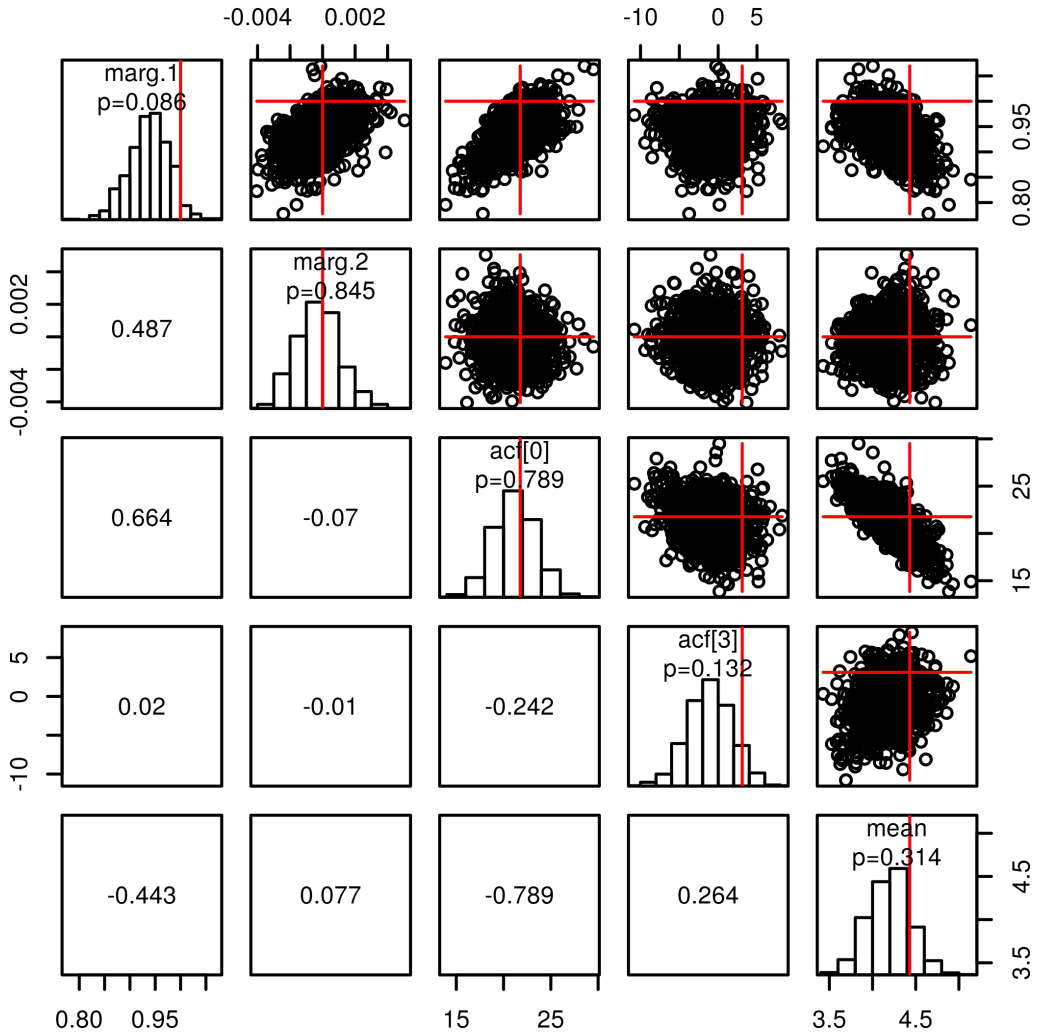

Figure 4: Results of `plot` on a class '`probed.pomp`' object. Above the diagonal, the pairwise scatterplots show the values of the probes on each of the 1000 data sets. The red lines show the values of each of the probes on the data. The panels along the diagonal show the distributions of the probes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

```
R> mf <- mif2(self_defined_ricker, start = guess, Nmif = 100, Np = 1000, transform = TRUE,
+             cooling.fraction = 0.95^50,
+             rw.sd = rw.sd(r = 0.1, sigma = 0.1, phi = 0.1))
R> mf <- continue(mf, Nmif = 500)
```

```
Error in dimnames(x) <- dn:  length of 'dimnames' [2] not equal to array
extent Error in h(simpleError(msg, call)):  error in evaluating the argument
'x' in selecting a method for function 'print':  "align" must have length
                        equal to 9 ( ncol(x) + 1 )
```

Table 4: Parameter estimation by means of maximum synthetic likelihood (**??**) vs. by means of maximum likelihood via iterated filtering (**??**). The row labeled "guess" contains the point at which both algorithms were initialized. That labeled "truth" contains the true parameter value, i.e., that at which the data were generated. The rows labeled "MLE" and "MSLE" show the estimates obtained using iterated filtering and maximum synthetic likelihood, respectively. Parameters r, σ, and τ were estimated; all others were held at their true values. The columns labeled $\hat{\ell}$ and $\hat{\ell}_{\mathbb{S}}$ are the Monte Carlo estimates of the log likelihood and the log synthetic likelihood, respectively; their Monte Carlo standard errors are also shown. While likelihood maximization results in an estimate for which both $\hat{\ell}$ and $\hat{\ell}_{\mathbb{S}}$ exceed their values at the truth, the value of $\hat{\ell}$ at the MSLE is smaller than at the truth, an indication of the relative statistical inefficiency of maximum synthetic likelihood.

?? 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`. These results demonstrate that it is possible, and indeed not difficult, to maximize the likelihood for this model, contrary to the claim of **?**. Since synthetic likelihood discards some of the information in the data, it is not surprising that **??** also shows the statistical inefficiency of the maximum synthetic likelihood relative to that of the likelihood.

4.7. Bayesian feature matching via ABC

Whereas the synthetic likelihood approach carries out many simulations for each likelihood estimation, ABC (as described in **??**) uses only one. Each iteration of ABC is therefore much quicker, essentially corresponding to the cost of SMC with a single particle or the synthetic likelihood approach 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. We return to the Gompertz model to faciliate comparison between ABC and PMCMC.

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

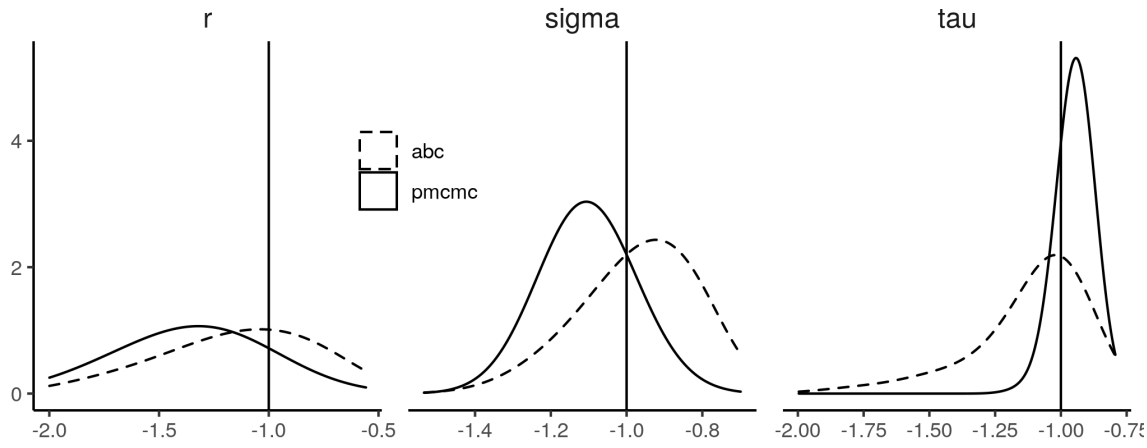

Figure 5: Marginal posterior distributions using full information via `pmcmc` (solid line) and partial information via `abc` (dashed line). Kernel density estimates are shown for the posterior marginal densities of $\log_{10}(r)$ (left panel), $\log_{10}(\sigma)$ (middle panel), and $\log_{10}(\tau)$ (right panel). The vertical lines indicate the true values of each parameter.

```
+        probes = plist, epsilon = 2, scale = scale.dat,
+        proposal = mvn.diag.rw(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 140, as compared to 560 for `pmcmc` in **??**. 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` mixes somewhat more rapidly (as measured by total computational effort) than `pmcmc`. **??** investigates the statistical efficiency of `abc` on this example. We see that `abc` gives rise to somewhat broader posterior distributions than the 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 (**??**). Despite this loss of statistical efficiency, points B**??**–B**??** of **??** identify situations in which ABC may be the only practical method available for Bayesian inference.

4.8. Parameter estimation by simulated quasi-likelihood

With the **pomp** package, it is fairly easy to try a quick comparison to see how `nlf` (**??**) compares with `mif` (**??**) 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 (**??**). The `nlf` implementation we will use to compare to the `mif` call from **??** is

```
R> nlf1 <- nlf_objfun(self_defined_gompertz, ti=100, tf=4000, lags = c(2, 3),
+              start = c(r = 1, K = 2, sigma = 0.5, tau = 0.5, X.0 = 1),
+              est = c("r", "sigma", "tau"))
R> subplex(par = c(r=0.2, sigma=0.15, tau=0.08), fn=nlf1, control=list(reltol=1e-4)) -> nl
```

where the first argument is the class '`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 = TRUE` setting causes the optimization to be performed on the transformed scale, as in **??**. 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 **?**. The quasi-likelihood 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 is not needed. `nlf` defaults to optimization by the subplex method (**??**), though all optimization methods provided by `optim` are available as well. `nasymp` sets the length of the simulation on which the quasi-likelihood 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.

?? compares the true parameter, θ, with the maximum likelihood estimate (MLE), $\hat{\theta}$, from `mif` and the maximized simulated quasi-likelihood (MSQL), $\tilde{\theta}$, from `nlf`. **??**A plots $\hat{\ell}(\hat{\theta}) - \hat{\ell}(\theta)$ against $\hat{\ell}(\tilde{\theta}) - \hat{\ell}(\theta)$, showing that the MSQL estimate can fall many units of log likelihood short of the MLE. **??**B plots $\hat{\ell}_Q(\hat{\theta}) - \hat{\ell}_Q(\theta)$ against $\hat{\ell}_Q(\tilde{\theta}) - \hat{\ell}_Q(\theta)$, showing that likelihood-based inference is almost as good as `nlf` at optimizing the simulated quasi-likelihood criterion which `nlf` targets. **??** suggests that the MSQL approach may be inefficient, since it can give estimates with poor behavior according to the statistically efficient criterion of likelihood. Another possibility is that this particular implementation of `nlf` was unfortunate. Each `mif` optimization took 97.1 sec to run, compared to 11 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. If the motivation for using NLF is preference for 2-step prediction in place of 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 (**??**). The susceptible class (S) contains those that have not yet been infected and are thereby still susceptible to it; the infected

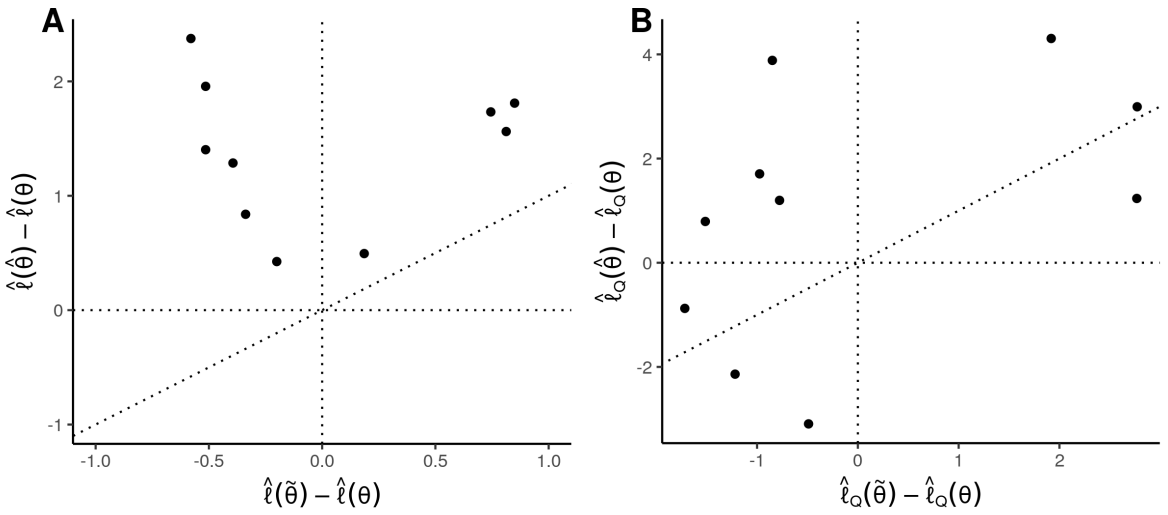

Figure 6: Comparison of `mif` and `nlf` for 10 simulated datasets using two criteria. In both plots, the maximum likelihood estimate (MLE), $\hat{\theta}$, obtained using iterated filtering is compared with the maximum simulated quasi-likelihood (MSQL) estimate, $\tilde{\theta}$, obtained using nonlinear forecasting. (A) Improvement in estimated log likelihood, $\hat{\ell}$, at the point estimate over that at the true parameter value, θ. (B) Improvement in simulated log quasi-likelihood $\hat{\ell}_Q$, at the point estimate over that at the true parameter value, θ. In both panels, the diagonal line is the 1–1 line.

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 as discrete births, deaths, and passages between compartments occur. Let N_{AB} be the stochastic counting process whose value at time t is the number of individuals that have passed from compartment A to compartment B during the interval $[t_0, t)$, where t_0 is an arbitrary starting point not later than the first observation. We use the notation $N_{\bullet \mathrm{S}}$ to refer to births and $N_{\mathrm{A}\bullet}$ to refer to deaths from compartment A. Let us assume that the *per capita* birth and death rates, 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 is 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 will make these assumptions, writing $\lambda(t) = \beta\, I(t)/P$, where β is the transmission rate and $P = S + I + R$ is the population size. We will go further and assume that birth and death rates are equal and independent of infection status; we will let μ denote the common rate. A consequence is that the expected population size remains constant.

The continuous-time Markov process is fully specified by the infinitesimal increment proba-

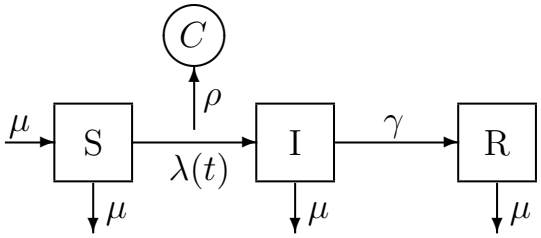

Figure 7: Diagram of the SIR epidemic model. The host population is divided into three classes according to infection status: S, susceptible hosts; I, infected (and infectious) hosts; R, recovered and immune hosts. Births result in new susceptibles and all individuals have a common death rate μ. Since the birth rate equals the death rate, the expected population size, $P = S+I+R$, 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 new infections are recorded with probability ρ. Since diagnosed cases are treated with bed-rest and hence removed, infections are counted upon transition to R.

bilities. Specifically, writing $\Delta N(t) = N(t+h) - N(t)$, we have

$$\begin{aligned}
\mathbb{P}\left[\Delta N_{\bullet \mathrm{S}}(t)=1 \,|\, S(t), I(t), R(t)\right] &= \mu\, P(t)\, h + o(h),\\
\mathbb{P}\left[\Delta N_{\mathrm{SI}}(t)=1 \,|\, S(t), I(t), R(t)\right] &= \lambda(t)\, S(t)\, h + o(h),\\
\mathbb{P}\left[\Delta N_{\mathrm{IR}}(t)=1 \,|\, S(t), I(t), R(t)\right] &= \gamma\, I(t)\, h + o(h),\\
\mathbb{P}\left[\Delta N_{\mathrm{S}\bullet}(t)=1 \,|\, S(t), I(t), R(t)\right] &= \mu\, S(t)\, h + o(h),\\
\mathbb{P}\left[\Delta N_{\mathrm{I}\bullet}(t)=1 \,|\, S(t), I(t), R(t)\right] &= \mu\, I(t)\, h + o(h),\\
\mathbb{P}\left[\Delta N_{\mathrm{R}\bullet}(t)=1 \,|\, S(t), I(t), R(t)\right] &= \mu\, R(t)\, h + o(h),
\end{aligned} \tag{14}$$

together with statement that all events of the form

$$\{\Delta N_{\mathrm{AB}}(t) > 1\} \qquad \text{and} \qquad \{\Delta N_{\mathrm{AB}}(t)=1, \Delta N_{\mathrm{CD}}(t)=1\}$$

for A, B, C, D with $(A,B) \neq (C,D)$ have probability $o(h)$. The counting processes are coupled to the state variables (**?**) via the following identities

$$\begin{aligned}
\Delta S(t) &= \Delta N_{\bullet \mathrm{S}}(t) - \Delta N_{\mathrm{SI}}(t) - \Delta N_{\mathrm{S}\bullet}(t),\\
\Delta I(t) &= \Delta N_{\mathrm{SI}}(t) - \Delta N_{\mathrm{IR}}(t) - \Delta N_{\mathrm{I}\bullet}(t),\\
\Delta R(t) &= \Delta N_{\mathrm{IR}}(t) - \Delta N_{\mathrm{R}\bullet}(t).
\end{aligned} \tag{15}$$

Taking expectations of **????**, dividing through by h, and taking the limit as $h \downarrow 0$, one obtains a system of nonlinear ordinary differential equations which is known as the deterministic skeleton of the model (**?**). Specifically, the SIR deterministic skeleton is

$$\begin{aligned}
\frac{dS}{dt} &= \mu\,(P - S) - \beta\,\frac{I}{P}\,S,\\
\frac{dI}{dt} &= \beta\,\frac{I}{P}\,S - \gamma\, I - \mu\, I,\\
\frac{dR}{dt} &= \gamma\, I - \mu\, R.
\end{aligned} \tag{16}$$

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

$$C(t_1, t_2) \sim \text{NegBin}(\rho\, \Delta N_{\text{SI}}(t_1, t_2), \theta), \tag{17}$$

where $\Delta N_{\text{SI}}(t_1, t_2)$ is the true incidence (the accumulated number of new infections that have occured over the $[t_1, t_2)$ interval), ρ is the *reporting rate*, (the probability that an infection is observed and recorded), θ is the negative binomial "size" parameter, and the notation is meant to indicate that $\mathsf{E}C(t_1, t_2) \,|\, \Delta N_{\text{SI}}(t_1, t_2) = H = \rho\, H$ and $\mathsf{VAR}C(t_1, t_2) \,|\, \Delta N_{\text{SI}}(t_1, t_2) = H = \rho\, H + \rho^2\, H^2/\theta$. The fact that the observed data are linked to an accumulation, as opposed to an instantaneous value, introduces a slight complication, which we discuss below.

5.2. Implementing the SIR model in pomp

As before, we will need to write functions to implement some or all of the SIR model's `rprocess`, `rmeasure`, and `dmeasure` components. As in **??**, we will write these components using **pomp**'s `Csnippet`s. Recall that these are snippets of C code that **pomp** automatically assembles, compiles, and dynamically loads into the running R session.

To start with, we will write snippets that specify the measurement model (`rmeasure` and `dmeasure`):

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

Here, we are using `cases` to refer to the data (number of reported cases) and `H` to refer to the true incidence 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 ensure that the state variable (`H`), observable (`cases`), parameters (`theta`, `rho`), and likelihood (`lik`) are defined in the contexts within which these snippets are executed.

For the `rprocess` portion, we could simulate from the continuous-time Markov process exactly (**?**); the **pomp** function `gillespie.sim` implements this algorithm. However, for practical purposes, the exact algorithm is often 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 needs to be assumed between the Euler simulation interval Δt and the reporting interval, which itself does not even need to 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. For example, if ΔN_{SI} and ΔN_{S} are the numbers of S individuals acquiring infection and dying, respectively, over the Euler simulation interval $[t, t+\Delta t)$, then

$$(\Delta N_{\mathrm{SI}}, \Delta N_{\mathrm{S}}, S - \Delta N_{\mathrm{SI}} - \Delta N_{\mathrm{S}}) \sim \mathrm{Multinom}\left(S(t); p_{S\to I}, p_{S\to}, 1 - p_{S\to I} - p_{S\to}\right), \tag{18}$$

where

$$\begin{aligned} p_{S\to I} &= \frac{\lambda(t)}{\lambda(t)+\mu}\,\left(1 - e^{-(\lambda(t)+\mu)\,\Delta t}\right), \\ p_{S\to} &= \frac{\mu}{\lambda(t)+\mu}\,\left(1 - e^{-(\lambda(t)+\mu)\,\Delta t}\right). \end{aligned} \tag{19}$$

By way of shorthand, we say that the random variable $(\Delta N_{\mathrm{SI}}, \Delta N_{\mathrm{S}})$ in **??** has an *Euler-multinomial* distribution. **pomp** provides convenience functions for such distributions, which arise with some frequency in compartmental models. Specifically, the functions `reulermultinom` and `deulermultinom` respectively draw random deviates from, and evaluate the probability mass function of, such distributions. As the help pages relate, `reulermultinom` and `deulermultinom` parameterize the Euler-multinomial distributions by the size ($S(t)$ in **??**), 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 write this in C to realize faster run-times:

```
R> sir.step <- "
+    double rate[6];
+    double dN[6];
+    double P;
+    P = S + I + R;
+    rate[0] = mu * P;       // birth
+    rate[1] = Beta * I / P; // transmission
+    rate[2] = mu;           // death from S
+    rate[3] = gamma;        // recovery
+    rate[4] = mu;           // death from I
+    rate[5] = mu;           // death from R
+    dN[0] = rpois(rate[0] * dt);
+    reulermultinom(2, S, &rate[1], dt, &dN[1]);
+    reulermultinom(2, I, &rate[3], dt, &dN[3]);
+    reulermultinom(1, R, &rate[5], dt, &dN[5]);
+    S += dN[0] - dN[1] - dN[2];
+    I += dN[1] - dN[3] - dN[4];
+    R += dN[3] - dN[5];
+    H += dN[1];
+  "
R>
```

```
R> initializer <- "S = nearbyint(popsize*S_0 / (S_0+I_0+R_0));
+                  I = nearbyint(popsize*I_0 / (S_0+I_0+R_0));
+                  R = nearbyint(popsize*R_0 / (S_0+I_0+R_0));
+                  H = 0;"
```

As before, **pomp** will ensure that the undeclared state variables and parameters are defined in the context within which the snippet is executed. Note, however, that in the above we do declare certain local variables. In particular, the `rate` and `dN` arrays hold the rates and numbers of transition events, respectively. Note too, that we make use of **pomp**'s C interface to `reulermultinom`, documented in the package help pages (`?reulermultinom`). The package help system (`?Csnippet`) includes instructions for, and examples of, the use of `Csnippet`s.

Two significant wrinkles remain to be explained. First, notice that in `sir.step`, the variable `H` simply accumulates the numbers of new infections: `H` is a counting process that is nondecreasing with time. In fact, the incidence within an interval $[t_1, t_2)$ is $\Delta N_{\mathrm{SI}}(t_1, t_2) = \mathtt{H}(t_2) - \mathtt{H}(t_1)$. This leads to a technical difficulty with the measurement process, however, in that the data are assumed to be records of new infections occurring within the latest reporting interval, while the process model tracks the accumulated number of new infections since time t_0. We can get around this difficulty by re-setting `H` to zero immediately after each observation. We cause **pomp** to do this via the `pomp` function's `zeronames` argument, as we will see in a moment. The section on "accumulator variables" in the `pomp` help page (`?pomp`) 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 by furnishing a custom `initializer` in the call to `pomp`. Let us construct it now and fill it with simulated data.

```
R> sir1 <- 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(
+         step.fun = Csnippet(sir.step), delta.t = 1/52/20),
+       statenames = c("S", "I", "R", "H"), accumvars="H",
+       paramnames = c("gamma", "mu", "theta", "Beta", "popsize",
+         "rho", "S_0", "I_0", "R_0"),
+       rinit = Csnippet(initializer), 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))
R> sir1 <- simulate(sir1, seed = 1914679908L)
```

Notice that we are assuming here that the data are collected weekly and use an Euler step-size of 1/20 wk. Here, we have 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 have assumed a host population size of 500,000 and 10% reporting efficiency. **??** shows one realization of this process.

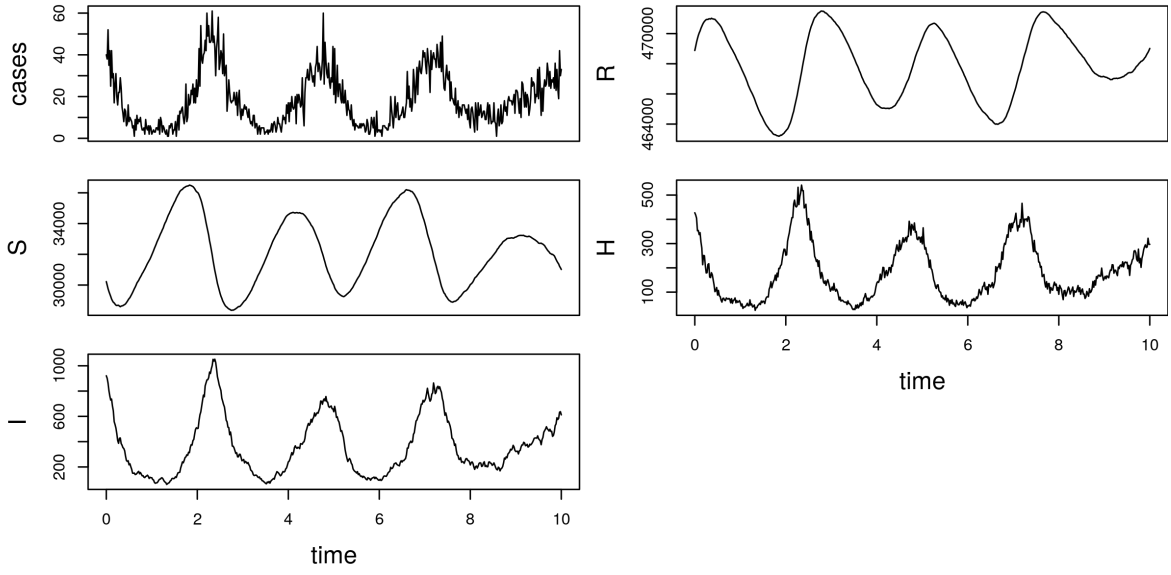

Figure 8: Result of `plot(sir1)`. The class '`pomp`' object `sir1` contains the SIR model with simulated data.

5.3. Incorporating additional model complexity

To illustrate the flexibility afforded by **pomp**'s plug-and-play methods, let us add a bit of real-world complexity to the simple SIR model. We will modify the model to take four facts into account:

1. For many infections, the contact rate is *seasonal*: $\beta = \beta(t)$ varies in more or less periodic fashion with 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 does not need to 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 have included so far.

To incorporate seasonality, we would like to assume a flexible functional form for $\beta(t)$. Here, we will use a three-coefficient Fourier series:

$$\log \beta(t) = b_0 + b_1 \cos 2\pi t + b_2 \sin 2\pi t. \tag{20}$$

There are a variety of ways to account for imported infections. Here, we will simply assume that there is some constant number, ι, of infected hosts visiting the population. Putting this together with the seasonal contact rate results in a force of infection $\lambda(t) = \beta(t)\ (I(t) + \iota)\,/N$.

To incorporate birth-rate information, let us 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 the varying birth rate into our model by passing it as a covariate to the simulation code. When we pass `birthdat` as the `covar` argument to `pomp`, we cause a look-up table to be created and made available to the simulator. The package employs linear interpolation to provide a value of each variable in the covariate table at any requisite time: from the user's perspective, a variable `births` will simply be available for use by the model codes.

Finally, we can allow for extrademographic stochasticity by allowing the force of infection to be itself a random variable. We will accomplish this by assuming a random phase in β:

$$\lambda(t) = \left(\beta(\Phi(t))\,\frac{I(t)+\iota}{N}\right), \tag{21}$$

where the phase Φ satisfies the stochastic differential equation

$$d\Phi = dt + \sigma\, dW_t, \tag{22}$$

where $dW(t)$ is a white noise, specifically an increment of standard Brownian motion. This model assumption attempts to capture variability in the timing of seasonal changes in transmission rates. As σ varies, it can represent anything from a very mild modulation of the timing of the seasonal progression to much more intense variation.

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

```
R> seas.sir.step <- "
+    double rate[6];
+    double dN[6];
+    double Beta;
+    double dW;
+    Beta = exp(b1 + b2 * cos(M_2PI * Phi) + b3 * sin(M_2PI * Phi));
+    rate[0] = births;                // birth
+    rate[1] = Beta * (I + iota) / P; // infection
+    rate[2] = mu;                    // death from S
+    rate[3] = gamma;                 // recovery
+    rate[4] = mu;                    // death from I
+    rate[5] = mu;                    // death from R
+    dN[0] = rpois(rate[0] * dt);
+    reulermultinom(2, S, &rate[1], dt, &dN[1]);
+    reulermultinom(2, I, &rate[3], dt, &dN[3]);
+    reulermultinom(1, R, &rate[5], dt, &dN[5]);
+    dW = rnorm(dt, sigma * sqrt(dt));
+    S += dN[0] - dN[1] - dN[2];
+    I += dN[1] - dN[3] - dN[4];
+    R += dN[3] - dN[5];
```

```
+    P = S + I + R;
+    Phi += dW;
+    H += dN[1];
+    noise += (dW - dt) / sigma;
+  "
R>
R> seas.initializer <- "S = nearbyint(popsize*S_0 / (S_0+I_0+R_0));
+                  I = nearbyint(popsize*I_0 / (S_0+I_0+R_0));
+                  R = nearbyint(popsize*R_0 / (S_0+I_0+R_0));
+                  P = popsize;
+                  H = 0;
+                  Phi = 0;
+                  noise = 0;"
R>
R> sir2 <- pomp(sir1, rprocess = euler(
+    step.fun = Csnippet(seas.sir.step), delta.t = 1/52/20),
+    dmeasure = Csnippet(dmeas), rmeasure = Csnippet(rmeas),
+    covar = covariate_table(birthdat, order="linear", times = "time"), accumvars = c("H",
+    statenames = c("S", "I", "R", "H", "P", "Phi", "noise"),
+    paramnames = c("gamma", "mu", "popsize", "rho", "theta", "sigma",
+                   "S_0", "I_0", "R_0", "b1", "b2", "b3", "iota"),
+    rinit = Csnippet(seas.initializer), params = c(popsize = 500000, iota = 5, b1 = 6, b2
+                  b3 = -0.1, gamma = 26, mu = 1/50, rho = 0.1, theta = 100,
+                  sigma = 0.3, S_0 = 0.055, I_0 = 0.002, R_0 = 0.94))
R> sir2 <- simulate(sir2, seed = 619552910L)
```

?? shows the simulated data and latent states. The `sir2` object we have constructed here contains all the key elements of models used within **pomp** to investigate cholera (**?**), measles (**?**), malaria (**?**), pertussis (**??**), pneumococcal pneumonia (**?**), rabies (**?**), and Ebola virus disease (**?**).

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 specification from the 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., **???????????**).

As a development platform, **pomp** is particularly 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 such as `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 infer-

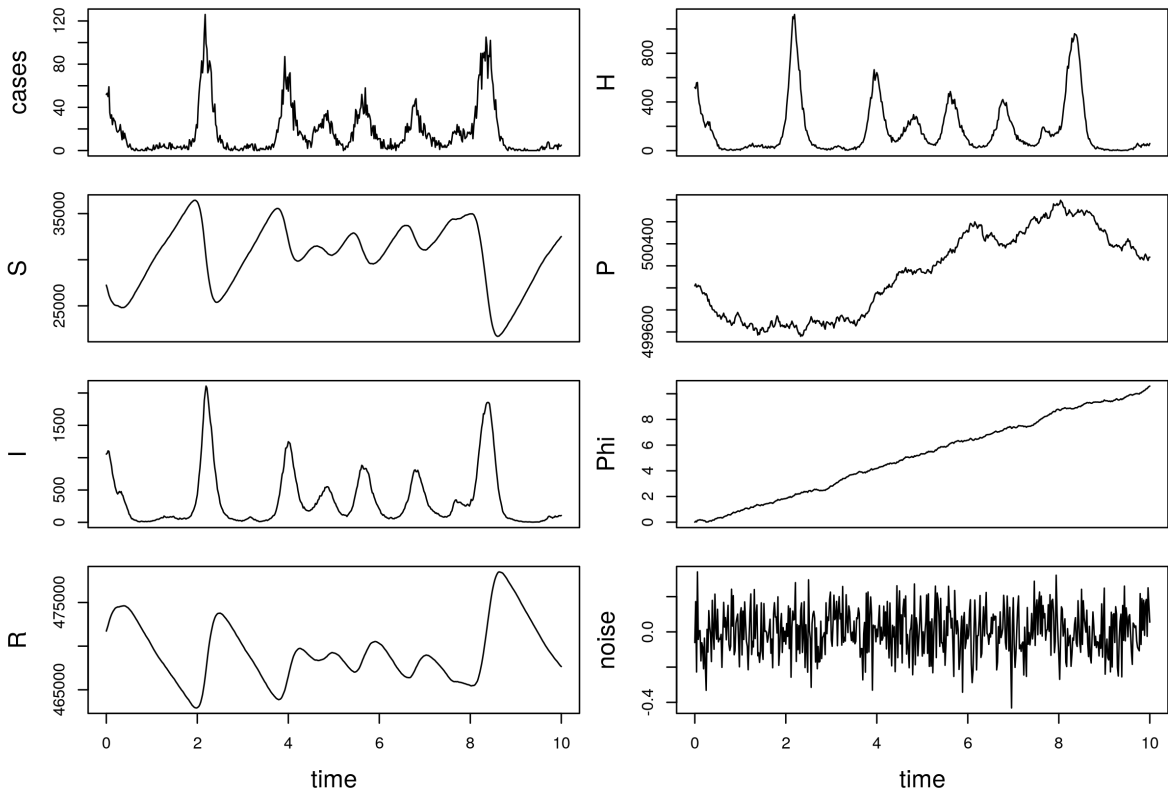

Figure 9: One realization of the SIR model with seasonal contact rate, imported infections, and extrademographic stochasticity in the force of infection.

ence 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** package 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 the assessment of Monte Carlo variability and the investigation of the roles 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 `pompExample`), which can be used as templates for implementation of new models; the R and C codes underlying these examples are 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, http://kingaa.github.io/pomp.

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.

Affiliation:

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

Dao Nguyen, Edward Ionides
Department of Statistics
University of Michigan
48109 Michigan, United States of America
E-mail: nguyenxd@umich.edu, ionides@umich.edu
URL: http://www.stat.lsa.umich.edu/~ionides/