A tutorial on panel data analysis using partially observed Markov processes via the R package panelPomp

Carles Bretó[1], Jesse Wheeler[2], Aaron A. King[2], and Edward L. Ionides[2]

[1]Universitat de València
[2]University of Michigan

September 5, 2024

Abstract

The R package panelPomp supports analysis of panel data via a general class of partially observed Markov process models (PanelPOMP). This package tutorial describes how the mathematical concept of a PanelPOMP is represented in the software and demonstrates typical use-cases of panelPomp. Monte Carlo methods used for POMP models require adaptation for PanelPOMP models due to the higher dimensionality of panel data. The package takes advantage of recent advances for PanelPOMP, including an iterated filtering algorithm, Monte Carlo adjusted profile methodology and block optimization methodology to assist with the large parameter spaces that can arise with panel models. In addition, tools for manipulation of models and data are provided that take advantage of the panel structure.

1 Introduction

This tutorial describes a typical use-case of the `panelPomp` `R` package. Partially observed Markov process (POMP) models—also known as state-space or hidden Markov models—are useful mathematical tools for modeling non-linear dynamic systems. POMP models describe a system via an unobserved dynamic model that has the Markov property, coupled with a model for how observations are drawn from the latent process. Various software packages provide platforms for performing statistical analysis of these systems using POMP models (for instance, `pomp` (King et al., 2016), `spatPomp` (Asfaw et al., 2024), `nimble` (Michaud et al., 2021), and `mcstate` (FitzJohn et al., 2020)). However, particular challenges arise when modeling *panel data* via POMP models; these data arise when time series are measured on a collection of independent units. While each unit may be modeled separately, analyzing the data as a single collection can provide insights into the underlying dynamical system that may not be obtained otherwise. For instance, each time series in the panel may be too short to infer a complex dynamical model, so that inference on an underlying model must combine information across the units. `panelPomp` is, to our knowledge, the first software package specifically addressing these issues. The utility of `panelPomp` has been demonstrated in several scientific applications (Ranjeva et al., 2017, 2019; Wale et al., 2019; Domeyer et al., 2022; Lee et al., 2020).

The current version of `panelPomp` emphasizes simulation-based methods, also known as plug-and-play methods (Bretó et al., 2009; He et al., 2010), or likelihood-free methods (Marjoram et al., 2003; Sisson et al., 2007). Such methods are applicable to dynamic models for which a simulator is available even when the transition densities are unavailable. This class of flexible algorithms allow researchers to build their models based on scientific reasoning rather than statistical convenience, typically at the expense of computational efficiency. In the following sections, this tutorial demonstrates how the `panelPomp` package can be used to model nonlinear dynamic systems using plug-and-play methodologies.

2 PanelPOMP models

Panel data is a collection of multiple time series datasets, each possibly multivariate on its own, where each time series is associated with a unit; the units can represent spatial locations, agents in a system, or other units for which data is collected over time. For convenience of identifying units in the panel, we use numeric labels $\{1, 2, \ldots, U\}$, which we also write as $1\!:\!U$. Time series from each unit may be of different lengths, and so we define N_u as the number of measurements collected on unit u. The observations are modeled as a realization of a stochastic process $Y_{u,1:N_u}$, observed at times $t_{u,1} < t_{u,2} < \cdots < t_{u,N_u}$. An

arbitrary realization of the observable process at time $t_{u,n}$ is denoted as $y_{u,n}$, and the entire collection of data is written as $y^*_{u,1:N_u} = \{y^*_{u,1}, \dots, y^*_{u,N_u}\}$, using the asterisk to differentiate the observed data from an arbitrary realization. The measurement process is assumed to be dependent on a latent Markov process $\{X_u(t), t_{u,0} \le t \le t_{u,N_u}\}$ defined subsequent to an initial time $t_{u,0} \le t_{u,1}$. Requiring that $\{X_u(t)\}$ and $\{Y_{u,i}, i \neq n\}$ are independent of $Y_{u,n}$ given $X_u(t_{u,n})$, for each $n \in 1:N_u$, completes the partially observed Markov process (POMP) model structure for unit u. For a PanelPOMP we require additionally that all units are modeled as independent.

The latent process can be modeled as either a discrete or continuous time process. For the continuous time process, the value of the latent states at observation times is of particular interest, so we write $X_{u,n} = X_u(t_{u,n})$. We suppose that $X_{u,n}$ and $Y_{u,n}$ take values in arbitrary spaces $\mathbb{X}_u$ and $\mathbb{Y}_u$ respectively, and that $X_{u,0:N_u}$ and $Y_{u,1:N_u}$ have a joint density written as $f_{X_{u,0:N_u}Y_{u,1:N_u}}(x_{u,0:N_u}, y_{u,1:N_u}\,;\theta)$ with dependence on an unknown real-valued parameter vector $\theta \in \mathbb{R}^D$. The transition density $f_{X_{u,n}|X_{u,n-1}}(x_{u,n}\,|\,x_{u,n-1}\,;\theta)$ and measurement density $f_{Y_{u,n}|X_{u,n}}(y_{u,n}\,|\,x_{u,n}\,;\theta)$ are permitted to depend arbitrarily on u and n, allowing non-stationary models and the inclusion of covariate time series. The marginal density of $Y_{u,1:N_u}$ at $y_{u,1:N_u}$ is $f_{Y_{u,1:N_u}}(y_{u,1:N_u}\,;\theta)$ and the likelihood function for unit u is $\ell_u(\theta) = f_{Y_{u,1:N_u}}(y^*_{u,1:N_u}\,;\theta)$. The likelihood for the entire panel is $\ell(\theta) = \prod_{u=1}^{U} \ell_u(\theta)$, and any solution $\hat\theta = \arg\max \ell(\theta)$ is a maximum likelihood estimate (MLE). The log likelihood is $\lambda(\theta) = \log \ell(\theta)$.

We introduce a structure to the parameter space which is not part of the general definition of a PanelPOMP model, but which is sufficiently common to deserve attention. Suppose the parameter vector can be written as $\theta = (\phi, \psi_1, \dots, \psi_U)$, where

$$
\begin{aligned}
f_{X_{u,n}|X_{u,n-1}}(x_{u,n}\,|\,x_{u,n-1}\,;\theta) &= f_{X_{u,n}|X_{u,n-1}}(x_{u,n}\,|\,x_{u,n-1}\,;\phi,\psi_u) && (1)\\
f_{Y_{u,n}|X_{u,n}}(y_{u,n}\,|\,x_{u,n}\,;\theta) &= f_{Y_{u,n}|X_{u,n}}(y_{u,n}\,|\,x_{u,n}\,;\phi,\psi_u) && (2)\\
f_{X_{u,0}}(x_{u,0}\,;\theta) &= f_{X_{u,0}}(x_{u,0}\,;\phi,\psi_u) && (3)
\end{aligned}
$$

Then, ψ_u is a vector of *unit-specific* parameters for unit u, and ϕ is a *shared* parameter vector. We suppose $\phi \in \mathbb{R}^A$ and $\psi \in \mathbb{R}^B$, so the dimension of the parameter vector θ is $D = A + BU$. The collection of unit-specific parameters can be considered as a $B \times U$ matrix $[\psi_{b,u}]$. Determining which parameters should be modeled as unit-specific and which should be shared is often itself an interesting scientific and statistical question.

The `panelPomp` class follows the mathematical structure described above, consisting of a list of `pomp` objects together with a specification of shared and unit-specific parameters. The `pomp` objects can be built using the `pomp()` constructor function from the `pomp R` package (King et al., 2016). These objects are bound into a `panelPomp` object by the

Method	Mathematical terminology
`rprocess`	Simulate from $f_{X_{u,n}\vert X_{u,n-1}}(x_{u,n} \,\vert\, x_{u,n-1}\,;\phi,\psi_u)$
`dprocess`	Evaluate $f_{X_{u,n}\vert X_{u,n-1}}(x_{u,n} \,\vert\, x_{u,n-1}\,;\phi,\psi_u)$
`rmeasure`	Simulate from $f_{Y_{u,n}\vert X_{u,n}}(y_{u,n} \,\vert\, x_{u,n}\,;\phi,\psi_u)$
`dmeasure`	Evaluate $f_{Y_{u,n}\vert X_{u,n}}(y_{u,n} \,\vert\, x_{u,n}\,;\phi,\psi_u)$
`rinit`	Simulate from $f_{X_{u,0}}(x_{u,0}\,;\phi,\psi_u)$
`rprior`	Simulate from the prior distribution $\pi(\theta)$
`dprior`	Evaluate the prior density $\pi(\theta)$

Table 1: Basic model components for `pomp` units making up a `panelPomp`.

constructor function `panelPomp()`. The general framework does not insist that units of a PanelPOMP share observation times or other model features—though any additional shared structure may simplify the specification of the list of constituent `pomp` objects. The specification of `pomp` models was discussed by King et al. (2016), and open-source examples are available online for diverse applications including those described in Section 3. Briefly, the model is specified by writing code to evaluate some or all of the following basic model components described in Table 1.

Algorithms written for `panelPomp` may access these functions. An algorithm is defined to be plug-and-play if it does not require `dprocess`. A `pomp` or `panelPomp` object does not need to have all the basic computations defined. In particular, if employing plug-and-play methodology there is no need to specify `dprocess`. Some algorithms may require model components beyond those tabulated above. For example, the `parameter_trans` function defines parameter transformations which may be carried out to facilitate model fitting by removing boundaries and/or shifting to a natural scale scale for exploring additive perturbations. In addition, Bayesian methods may call `rprior` or `dprior` if these have been defined.

A simple example of a PanelPOMP is a stochastic version of the discrete-time Gompertz model for biological population growth (Winsor, 1932). This model supposes that the density, $X_{u,n+1}$, of a population u at time $n+1$ depends on the density, $X_{u,n}$, at time n according to

$$X_{u,n+1} = \kappa_u^{1-e^{-r_u}}\, X_{u,n}^{e^{-r_u}}\, \varepsilon_{u,n}. \tag{4}$$

In (4), κ_u is the carrying capacity of population u, r_u is a positive parameter, and $\{\varepsilon_{u,n}, u \in 1\!:\!U, n \in 1\!:\!N_u\}$ are independent and identically-distributed lognormal random variables with $\log \varepsilon_{u,n} \sim \text{Normal}(0, \sigma^2_{\text{G},u})$. We suppose the population is observed with lognormally distributed errors,

$$\log Y_{u,n} \;\sim\; \text{Normal}\left(\log X_{u,n}, \tau_u^2\right).$$

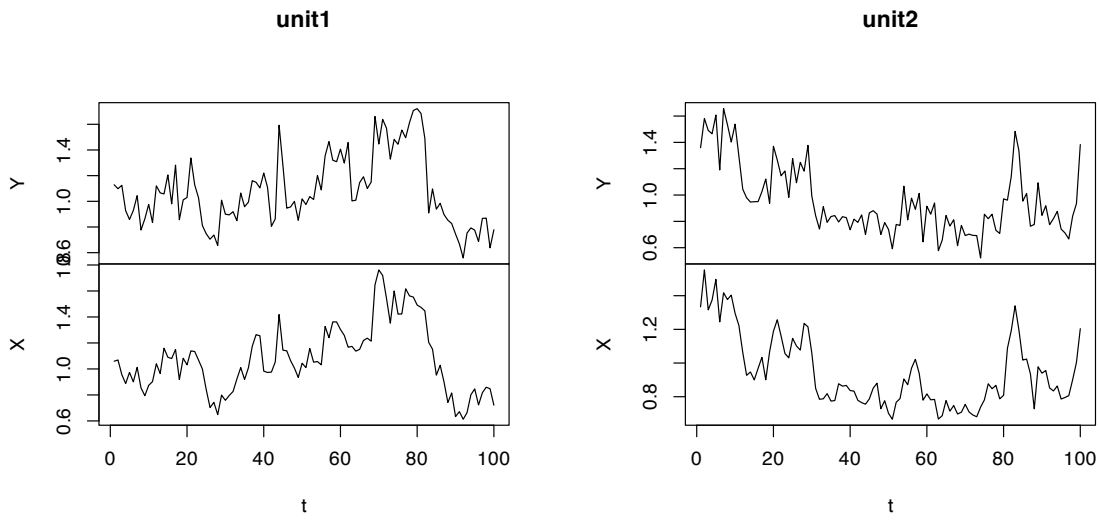

Figure 1: Separate plots produced by `plot(gomp[1:2])`.

This is accessible via

```
gomp <- panelGompertz(N = 100, U = 50)
```

Here, the number of units is `length(gomp)=50`. `panelPomp` uses S4 classes (Chambers, 1998; Genolini, 2008) with `gomp` having the base class `panelPomp`. Because data are a key component of PanelPOMP models, the `panelGompertz` function first creates a PanelPOMP model, and then generates a dataset by simulating from that model using a reproducible seed specified by the `seed` argument to the function.

Commonly, the first thing to do with a new object is to plot it, and Fig. 1 demonstrates the panelPomp `plot` method applied after subsetting `gomp`. The units of a panelPomp do not necessarily share the same variables, so in general a sequence of separate plots is all that can be offered. It may happen that the units can meaningfully be plotted on the same axis, and that can be achieved by coercing the `panelPomp` object to a `pompList` and using the `plot` method for that class (Fig. 2). A third option is to export the `panelPomp` object via `as(gomp,"data.frame")` and work this this to produce customized plots.

A basic operation is simulation. We can generate another simulation with the same parameter values:

```
gomp2 <- simulate(gomp)
```

We can simulate from the same model at new parameter values by giving additional arguments to `simulate`. There are two different representations of parameters within

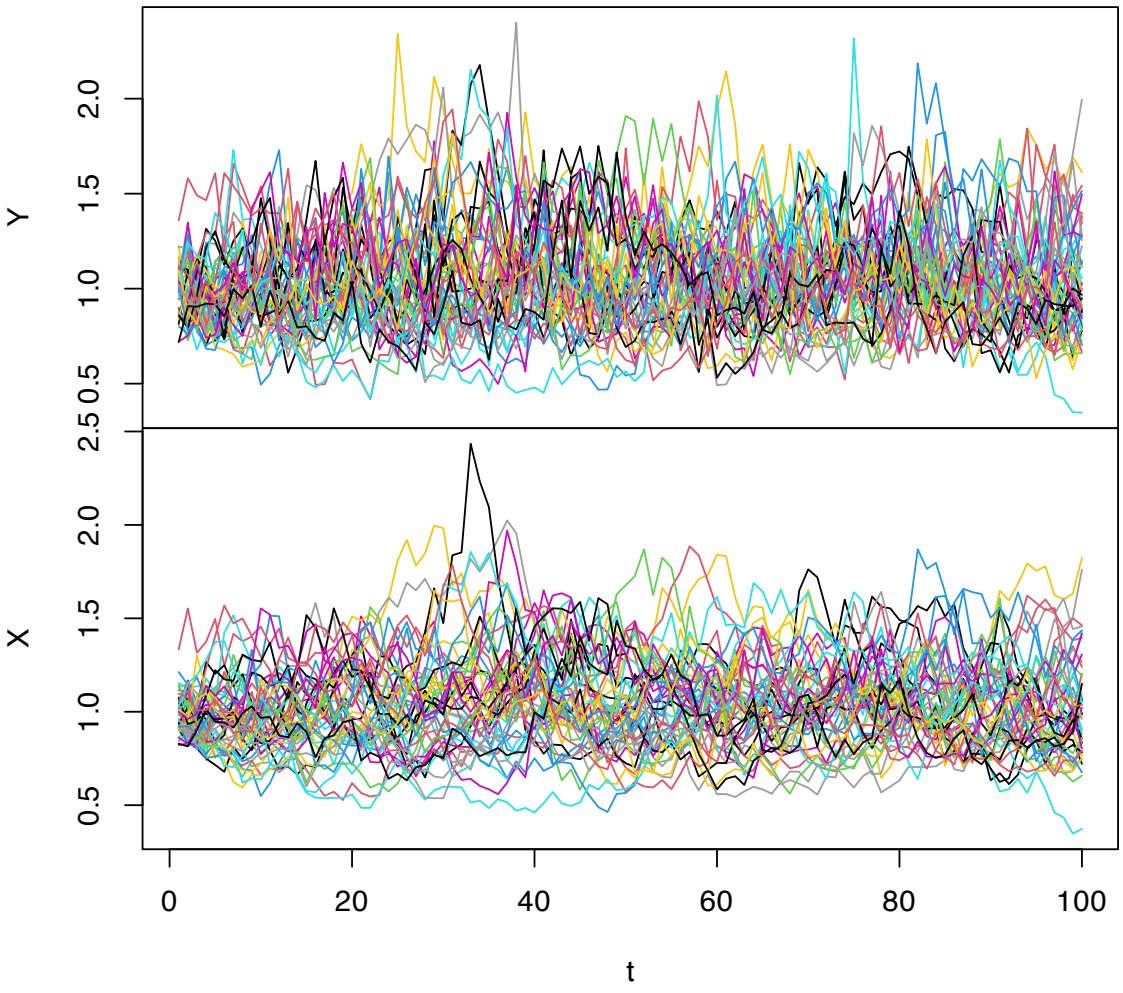

Figure 2: Overlayed time series plot using `plot(as(gomp,'pompList'))`.

`panelPomp` which are convenient in different situations. The unit-specific parameters are naturally represented as a matrix, set with the `specific` argument in the `panelPomp` constructor function, with a column for each units and a row for each parameter. Similarly, the shared parameters are a named vector that can be set using the `shared` argument. Alternatively, we can consider the parameters as a single named vector, with a naming convention that `"beta[unit7]"` is the name of the unit-specific parameter `beta` for unit $u = 7$. Model parameters can be extracted and set in this vector-format using the functions `coef()` and `coef()<-`, respectively. Alternatively, unit-specific and shared parameters can be extracted using functions `specific()` and `shared()`, and modified using the equivalent setter functions `specific()<-` and `shared()<-`. For example, all shared and a subset of unit-specific parameters (from units 1–3) can be extracted in vector format via

```
coef(gomp[1:3])

##          r      sigma   K[unit1] tau[unit1] X.0[unit1]   K[unit2] tau[unit2]
##        0.1        0.1        1.0        0.1        1.0        1.0        0.1
## X.0[unit2]   K[unit3] tau[unit3] X.0[unit3]
##        1.0        1.0        0.1        1.0
```

or in a list format using the `format = 'list'` argument

```
coef(gomp[1:3], format = 'list')

## $shared
##     r sigma
##   0.1   0.1
##
## $specific
##     unit1 unit2 unit3
## K     1.0   1.0   1.0
## tau   0.1   0.1   0.1
## X.0   1.0   1.0   1.0
```

The functions `toParamList` and `toParamVec` facilitate movement between the vector and list formats

```
toParamList(coef(gomp[1:3]))

## $shared
```

```
##     r sigma
##   0.1   0.1
##
## $specific
##      unit
## param unit1 unit2 unit3
##   K     1.0   1.0   1.0
##   tau   0.1   0.1   0.1
##   X.0   1.0   1.0   1.0
```

`panelPomp` seeks to avoid unnecessary duplication with `pomp`. Thus, `panelPomp` requires that `pomp` is loaded and builds on existing functionality of `pomp` where possible. In particular, a list of `pomp` objects for each unit can be extracted from a `panelPomp` object via

```
as(panelPompObject,"list")
```

Some methods in the `pomp` package take advantage of a `pompList` class which is defined as a list of `pomp` objects. These methods can be accessed via

```
as(panelPompObject,"pompList")
```

Many critical issues in computational performance of the basic model components for `panelPomp`, and utilities for assisting with the user specification of the model, have already received extensive development and testing in the context of `pomp`. All the facilities for constructing POMP models in `pomp` are available for constructing the models for each unit in `panelPomp`. This article avoids duplication by referring the reader to King et al. (2016) and the `pomp` documentation for detailed discussion of constructing `pomp` objects. In Section 3, we identify some case studies that provide useful code for scientific applications of `panelPomp`. In Section 4, we proceed to demonstrate inference for `panelPomp` objects, emphasizing methodological issues arising due to the specific requirements of panel data, in the context of a toy example.

3 Implementing mechanistic models: case studies

Developing mechanistic statistical models for new scientific applications requires identifying the essential variables and their functional relationships, and obtaining a satisfactory description of stochasticity in both the measurement process and the system dynamics. This

challenging but valuable exercise is assisted by simulation-based software that permits implementation of a general class of models. Once a suitable model is found, it provides a testable benchmark for subsequent investigations of the system under study and other comparable systems. Published models and methods, equipped with data and reproducible source code, are essential to maintain this progressive development.

Since each unit in a `panelPomp` is itself a `pomp`, we refer to King et al. (2016) for full details of how these are specified, and we focus on the new issues arising in `panelPomp`. Recall from Section 2 that a `panelPomp` is constructed from the list of constituent `pomp` models together with a collection of shared and unit-specific parameters. A unit-specific parameter named `theta` should be called `theta` in each constituent `pomp`. Thus, the value `theta[unit7]` specific to unit 7 is just passed as `theta` when required by the `pomp` model representing this unit.

Existing examples of `panelPomp` analysis have primarily concerned infectious disease dynamics, a topic that has motivated many advances in inference for partially observed stochastic dynamic systems. We discuss five of these below. In addition, the close relationship between `panelPomp` and `pomp` objects means that `panelPomp` model constructions can borrow from the considerable existing resources for `pomp`. The remaining examples give some applications in other domains which have been carried out using `pomp` models but have extensions to `panelPomp` situations.

1. Sexual contacts: behavioral heterogeneity within and between individuals. Romero-Severson et al. (2015) developed a PanelPOMP model to investigate a longitudinal prospective survey of sexual contacts, quantifying the roles of behavioral differences between individuals and differences within an individual over time. The `contacts()` function in `panelPomp` generates one of the models and datasets studied in this paper. The source code to generate this `panelPomp` object is at `https://github.com/cbreto/panelPomp`.

2. Recurrent infection with HPV. Ranjeva et al. (2017) developed a PanelPOMP model to study the strain dynamics of a longitudinal prospective serological survey of human papillomavirus (HPV). Their `panelPomp` code is available at `https://github.com/cobeylab/HPV-model`.

3. Polio: asymptomatic infection and local extinction. Martinez-Bakker et al. (2015) developed a POMP model for polio transmission to investigate the pre-vaccination epidemics in USA, fitting all parameters separately for each state. Bretó et al. (2020) found additional precision in inferences when the states are combined into a PanelPOMP with some shared parameters. The `polio()` function in `panelPomp` generates this model, and the source code is at `https://github.com/cbreto/panelPomp`.

4. Age-specific differences in the dynamics of protective immunity to influenza. Ranjeva et al. (2019) developed a PanelPOMP model to interpret longitudinal study of serological measurements on human influenza immunity. The source code for their `panelPomp` model is at `https://github.com/cobeylab/Influenza-immune-dynamics`.

5. The dynamic struggle between malaria and the immune system. Wale et al. (2019) developed a PanelPOMP model to investigate the dynamics of the immune response to malaria, based on flow cytometry time series for a panel of mice under varying treatments. Their `panelPomp` source code and data are available at `https://doi.org/10.5061/dryad.nk98sf7pk`.

6. Ecological predator-prey dynamics: consumptive and non-consumptive effects. Marino et al. (2019) developed a stochastic seasonal predator-prey POMP model to investigate the relationship between an abundant zooplankton species, *Daphnia mendotae*, and its predator, *Bythotrephes longimanus*, in Lake Michigan. The source code for the `pomp` analysis is available at `https://doi.org/10.5061/dryad.bh688ft`.

7. Stochastic volatility and financial leverage. Bretó (2014) demonstrated the applicability of plug-and-play methods within `pomp` to investigate stochastic volatility in finance using a POMP model for a single index.

Many other POMP models implemented using `pomp` are presented at `https://kingaa.github.io/pomp/biblio.html`.

4 Methodology for PanelPOMP models

All POMP methods can in principle be extended to PanelPOMPs since a PanelPOMP can be written as a POMP. Three different ways to represent a PanelPOMP as a POMP were identified by Romero-Severson et al. (2015): (i) the panels can be concatenated temporally into a long time series; (ii) the panels can be adjoined to form a high-dimensional POMP with a latent state comprised of a vector of latent states for each unit; (iii) time in the POMP representation can correspond to unit, u, with a vector valued state representing the full process for this unit. The existence of these representations does not necessarily imply that POMP methods will be computationally feasible on the resulting PanelPOMP. In particular, sequential Monte Carlo algorithms can have prohibitive scaling difficulties with the high dimensional latent states that can be involved with representations (ii) and (iii).

Here, we focus on describing and demonstrating the plug-and-play likelihood-based inference workflow used in the scienfic examples of Section 3. This approach builds on

likelihood evaluation via the particle filter using `pfilter()` and likelihood maximization via iterated filtering using `mif2()`. These algorithms can be formally justified in terms of representation (i) above (Bretó et al., 2020), though the numerical implementation does not in practice have to explicitly construct the concatenation of the `panelPomp` object into a `pomp` object.

4.1 Log likelihood evaluation via panel particle filtering

The particle filter, also known as sequential Monte Carlo, is a standard tool for log likelihood evaluation on non-Gaussian POMP models. The log likelihood function is a central component of Bayesian and frequentist inference. Due to the dynamic independence assumed between units, particle filtering can be carried out separately on each unit. The `pfilter` method for `panelPomp` objects is therefore a direct extension of the `pfilter` method for `pomp` objects from the `pomp` package. Repeating `pfilter` is advisable to reduce the Monte Carlo error on the log likelihood evaluation and to quantify this error. The following code carries out replicated evaluations of the log likelihood of `gomp`, taking advantage of multicore computation. The Gompertz model is a convenient for testing methodology for nonlinear non-Gaussian models since it has a logarithmic transformation to a linear Gaussian process and therefore the exact likelihood is computable by the Kalman filter (King et al., 2016).

```
pf_results <- foreach(i=1:10) %dopar% pfilter(gomp,
  Np=switch(run_level,10,200,1000))
```

This took 0.05 minutes using 4 cores, resulting in a list of objects of class `pfilterd.ppomp`. We can use `logLik` to extract the Monte Carlo likelihood esimate $\lambda^{[i]}$ for each replicate i, and `unitlogLik` to extract the vector of component Monte Carlo likelihood esimates $\lambda_u^{[i]}$ for each unit $u = 1, \ldots, U$, where $\lambda^{[i]} = \sum_{u=1}^{U} \lambda_u^{[i]}$. For a POMP model, replicated particle filter likelihood evaluations are usually averaged on the natural scale, rather than the log scale, to take advantage of the unbiasedness of the particle filter likelihood estimate. Thus, we have

$$\hat{\lambda}_1 = \log \frac{1}{I} \sum_{i=1}^{I} \exp \left\{ \sum_{u=1}^{U} \lambda_u^{[i]} \right\}$$

which can be implemented as

```
lambda_1 <- logmeanexp(sapply(pf_results,logLik),se=TRUE)
```

giving $\hat{\lambda}_1 = 2066.5$ with a jack-knife standard error of 0.7. Taking advantage of the independence of the units in the panel structure, Bretó et al. (2020) showed it is preferable

to average the replicates of marginal likelihood for each unit before taking a product over units. This corresponds to

$$\hat{\lambda}_2 = \log \prod_{u=1}^{U} \frac{1}{I} \sum_{i=1}^{I} \exp\left\{\hat{\lambda}_u^{[i]}\right\}$$

which can be implemented as

```
lambda_2 <- panel_logmeanexp(sapply(pf_results,unitlogLik),
  MARGIN=1,se=TRUE)
```

giving $\hat{\lambda}_2 = 2067.9$ with a jack-knife standard error of 1.1. For this model, a Kalman filter likelihood evaluation gives an exact answer, $\lambda = 2068.2$.

4.2 Maximum likelihood estimation via Panel Iterated Filtering

Iterated filtering algorithms carry out repeated particle filtering operations on an extended version of the model that includes time-varying perturbations of parameters. At each iteration, the magnitude of the perturbations is decreased, and in a suitable limit the algorithm approaches a local maximum of the likelihood function. The IF2 iterated filtering algorithm (Ionides et al., 2015) has been used for likelihood-based inference on various POMP models arising in epidemiology and ecology (reviewed by Bretó, 2018), superseding the previous IF1 algorithm of Ionides et al. (2006). IF2 is implemented in `pomp` as the `mif2` method for class `pomp`. A panel iterated filtering (PIF) algorithm, extending IF2 to panel data, was developed by Bretó et al. (2020). An implementation of PIF in `panelPomp` is provided by the `mif2` method for class `panelPomp`, following the pseudocode in Algorithm 1. The pseudocode in Algorithm 1 sometimes omits explicit specification of ranges over which variables are to be computed when this is apparent from the context: it is understood that j takes values in $1:J$, a in $1:A$ and b in $1:B$. The $N[0,1]$ notation corresponds to the construction of independent standard normal random variables, leading to to Gaussian perturbations of parameters on a transformed scale. These perturbations could follow an arbitrary distribution within the theoretical frameworks of IF2 and PIF.

At a conceptual level, the PIF algorithm has an evolutionary analogy: successive iterations mutate parameters and select among the fittest outcomes measured by Monte Carlo likelihood evaluation. The theory allows considerable flexibility in how the parameters are perturbed, but Gaussian perturbations on an appropriate scale are typically adequate. Most often, the perturbation parameters $\sigma^{\Phi}_{a,n}$ and $\sigma^{\Psi}_{b,u,n}$ in Algorithm 1 will not depend on n. For parameters set to have uncertainty on a unit scale, the value 0.02 demonstrated here has been commonly used. The help documentation on the `rw_sd` argument gives instruction

Algorithm 1: `mif2(pp, Nmif =` M`, Np =` J`, start =` $(\phi^0_a, \psi^0_{b,u})$`, rw_sd =` $(\sigma^\Phi_{a,n}, \sigma^\Psi_{b,u,n})$`, cooling.factor.50 =` ρ^{50}`)`, where `pp` is a `panelPomp` object containing data and defined `rprocess`, `dmeasure`, `rinit` and `partrans` components.

input: Data, $y^*_{u,n}$, u in $1{:}U$, n in $1{:}N$
Simulator of initial density, $f_{X_{u,0}}(x_{u,0}\,;\phi,\psi_u)$
Simulator of transition density, $f_{X_{u,n}|X_{u,n-1}}(x_{u,n}\,|\,x_{u,n-1}\,;\phi,\psi_u)$
Evaluator of measurement density, $f_{Y_{u,n}|X_{u,n}}(y_{u,n}\,|\,x_{u,n}\,;\phi,\psi_u)$
Number of particles, J, and number of iterations, M
Starting shared parameter swarm, $\Phi^0_{a,j}=\phi^0_a$, a in $1{:}A$, j in $1{:}J$
Starting unit-specific parameter swarm, $\Psi^0_{b,u,j}=\psi^0_{b,u}$, b in $1{:}B$, j in $1{:}J$
Random walk intensities, $\sigma^\Phi_{a,n}$ and $\sigma^\Psi_{b,u,n}$
Parameter transformations, h^Φ_a and h^Ψ_b, with inverses $\left(h^\Phi_a\right)^{-1}$ and $\left(h^\Psi_b\right)^{-1}$

output: Final parameter swarm, $\Phi^M_{a,j}$ and $\Psi^M_{b,u,j}$

For m in $1{:}M$
  $\Phi^m_{a,0,j}=\Phi^{m-1}_{a,j}$
  For u in $1{:}U$
    $\Phi^{F,m}_{a,u,0,j}=\left(h^\Phi_a\right)^{-1}\left(h^\Phi_a\left(\Phi^m_{a,u-1,j}\right)+\rho^m\sigma^\Phi_{a,0}Z^{\Phi,m}_{a,u,0,j}\right)$ for $Z^{\Phi,m}_{a,u,0,j}\sim N[0,1]$
    $\Psi^{F,m}_{b,u,0,j}=\left(h^\Psi_b\right)^{-1}\left(h^\Psi_b\left(\Psi^{m-1}_{b,u,j}\right)+\rho^m\sigma^\Psi_{b,u,0}Z^{\Psi,m}_{b,u,0,j}\right)$ for $Z^{\Psi,m}_{b,u,0,j}\sim N[0,1]$
    $X^{F,m}_{u,0,j}\sim f_{X_{u,0}}\left(x_{u,0}\ ;\ \Phi^{F,m}_{a,u,0,j},\Psi^{F,m}_{b,u,0,j}\right)$
    For n in $1{:}N_u$
      $\Phi^{P,m}_{a,u,n,j}=\left(h^\Phi_a\right)^{-1}\left(h^\Phi_a\left(\Phi^{F,m}_{a,u,n-1,j}\right)+\rho^m\sigma^\Phi_{a,n}Z^{\Phi,m}_{a,u,n,j}\right)$ for $Z^{\Phi,m}_{a,u,n,j}\sim N[0,1]$
      $\Psi^{P,m}_{b,u,n,j}=\left(h^\Psi_b\right)^{-1}\left(h^\Psi_b\left(\Psi^{F,m}_{b,u,n-1,j}\right)+\rho^m\sigma^\Psi_{b,u,n}Z^{\Psi,m}_{b,u,n,j}\right)$ for $Z^{\Psi,m}_{b,u,n,j}\sim N[0,1]$
      $X^{P,m}_{u,n,j}\sim f_{X_{u,n}|X_{u,n-1}}\left(x_{u,n}\ \middle|\ X^{F,m}_{u,n-1,j}\ ;\ \Phi^{P,m}_{a,u,n,j}\,,\Psi^{P,m}_{b,u,n,j}\right)$
      $w^m_{u,n,j}=f_{Y_{u,n}|X_{u,n}}\left(y^*_{u,n}\ \middle|\ X^{P,m}_{u,n,j}\ ;\ \Phi^{P,m}_{a,u,n,j}\,,\Psi^{P,m}_{b,u,n,j}\right)$
      Draw $k_{1:J}$ with $\mathbb{P}(k_j=i)=w^m_{u,n,i}\Big/\sum_{q=1}^J w^m_{u,n,q}$
      $\Phi^{F,m}_{a,u,n,j}=\Phi^{P,m}_{a,u,n,k_j}$, $\Psi^{F,m}_{b,u,n,j}=\Psi^{P,m}_{b,u,n,k_j}$ and $X^{F,m}_{u,n,j}=X^{P,m}_{u,n,k_j}$
    End For
    $\Phi^m_{a,u,j}=\Phi^{F,m}_{a,u,N_u,j}$ and $\Psi^m_{b,u,j}=\Psi^{F,m}_{b,u,N_u,j}$
  End For
  $\Phi^m_{a,j}=\Phi^m_{a,U,j}$
End For

on using additional structure should it become necessary.

For positive parameters, a logarithmic transform can achieve both tasks of removing the boundary and placing uncertainty on a unit scale. For the panel Gompertz model, all the parameters are non-negative valued, and so the `panelGompertz()` code calls `panelPomp` with an argument

```
partrans=parameter_trans(log=c("K","r","sigma","tau","X.0"))
```

Inference methodology can call `partrans(...,dir="toEst")` to work with parameters on a suitable scale, usually one where additive variation is meaningful. The methodology can revert to the original parameterization, presumably chosen to be scientifically convenient or meaningful, using `partrans(...,dir="fromEst")`. Thus, a user who does not have to look 'under the hood' never has to be directly concerned with parameters on the transformed scale, beyond assigning the transformation.

For Monte Carlo maximization, replication from diverse starting points is recommended. We demonstrate such a maximization search on `gomp`. For simplicity, we fix $K_u = 1$ and the initial condition $X_{u,0} = 1$, maximizing over two shared parameters, r and σ, and one unit-specific parameter τ_u. To define the diverse starting points, we make uniform draws from a specified box. We are not promising that the search will stay within this box, and indeed we should be alert to the possibility that the data lead us elsewhere. However, if replicated searches started from this box reliably reach a consensus, we claim we have carefully investigated this part of parameter space. A larger box leads to greater confidence that the relevant part of the parameter space has been searched, at the expense of requiring additional work. The `runif_panel_design` function constructs a matrix of random draws from the box.

```
starts <- runif_panel_design(
  lower = c('r' = 0.05, 'sigma' = 0.05, 'tau' = 0.05, 'K' = 1, 'X.0' = 1),
  upper = c('r' =  0.2, 'sigma' =  0.2, 'tau' =  0.2, 'K' = 1, 'X.0' = 1),
  specific_names = c('K', 'tau', 'X.0'),
  unit_names = names(gomp),
  nseq=switch(run_level,2,4,6)
)
```

We then carry out a search from each starting point:

```
mif_results <- foreach(start=iter(starts,"row")) %dopar% {
  mif2(gomp, start=unlist(start),
    Nmif = switch(run_level,2,20,150),
    Np = switch(run_level,10,500,1500),
    cooling.fraction.50=0.5,
    cooling.type="geometric",
    transform=TRUE,
    rw.sd=rw_sd(r=0.02,sigma=0.02,tau=0.02)
```

```
  )
}
```

This took 15.0 minutes using 4 cores, producing a list of objects of class `mifd.ppomp`. The algorithmic parameters are very similar to those of the `mif2` method for class `pomp`. The perturbations, determined by the `rw_sd` argument, may be a list giving separate instructions for each unit. When only one specification for a unit-specific parameter is given (as we do for K_u here) the same perturbation is used for all units.

We can check on convergence of the searches, and possibly diagnose improvements in the choices of algorithmic parameters, by consulting trace plots of the searches available via the `traces` method for class `mifd.ppomp`. This follows recommendations by (Ionides et al., 2006) and (King et al., 2016).

An issue characteristic of PanelPOMP models is using the panel structure to facilitate the large number of parameters arising when unit-specific parameters are specified for a large number of units. For a fixed value of the shared parameters, the likelihood of the unit-specific parameters factorizes over the units. The factorized likelihood can be maximized separately over each unit, replacing a challenging high-dimensional problem with many relatively routine low-dimensional problems. This suggests a block maximization strategy where unit-specific parameters for each unit are maximized as a block. Bretó et al. (2020) used a simple block strategy where a global search over all parameters is followed by a block maximization over units for unit-specific parameters. We demonstrate this here, refining each of the maximization replicates above. The following function carries out a maximization search of unit-specific parameters for a single unit. The call to `mif2` takes advantage of argument recycling: all algorithmic parameters are re-used from the construction of `mifd_gomp` except for the respecified random walk standard deviations which ensures that only the unit-specific parameters are perturbed.

```
mif_unit <- function(unit,mifd_gomp,reps=switch(run_level,2,4,6)){
  unit_gomp <- unit_objects(mifd_gomp)[[unit]]
  mifs <- replicate(n=reps,mif2(unit_gomp,rw.sd=rw_sd(tau=0.02)))
  best <- which.max(sapply(mifs,logLik))
  coef(mifs[[best]])["tau"]
}
```

Now we apply this block maximization to find updated unit-specific parameters for each replicate, and we insert these back into the `panelPomp`

```
mif_block <- foreach(mf=mif_results) %dopar% {
  mf@specific["tau",] <- sapply(1:length(mf),mif_unit,mifd_gomp=mf)
  mf
}
```

This took 0.8 minutes.

We expect Monte Carlo estimates of the maximized log likelihood functions to fall below the actual (usually unknown) value. This is in part because imperfect maximization can only reduce the maximized likelihood, and in part a consequence of Jensen's inequality applied to the likelihood evaluation: the unbiased SMC likelihood evaluation has a negative bias on estimation of the log likelihood.

4.3 Monte Carlo profile likelihood

The profile likelihood function is constructed by fixing one focal parameter at a range of values and then maximizing the likelihood over all other parameters for each value of the focal parameter. Constructing a profile likelihood function has several practical advantages.

1. Evaluations at neighboring values of the focal parameter provide additional Monte Carlo replication. Typically, the true profile log likelihood is smooth, and asymptotically close to quadratic under regularity conditions, so deviations from a smooth fitted line can be interpreted as Monte Carlo error.

2. Large-scale features of the profile likelihood reveal a region of the parameter space outside which the model provides a poor explanation of the data.

3. Co-plots, showing how the values of other maximized parameters vary along the profile, may provide insights into parameter tradeoffs implied by the data.

4. The smoothed Monte Carlo profile log likelihood can be used to construct an approximate 95% confidence interval. The resulting confidence interval can be properly adjusted to accommodate both statistical and Monte Carlo uncertainty (Ionides et al., 2017).

Once we have code for maximizing the likelihood, only minor adaptation is needed to carry out the maximizations for a profile. The `runif_panel_design` generating the starting values is replaced by a call to `profile_design`, which assigns the focal parameter to a grid of values and randomizes the remaining parameters. The random walk standard deviation for the focal parameter is unassigned, which leads it to be set to zero and therefore the

parameter remains fixed during the maximization process. The following code combines the joint and block maximizations developed above.

```
# Names of the estimated parameters
estimated <- c(
  "r", "sigma", paste0("tau[unit", 1:length(gomp), "]")
)

# Names of the fixed parameters (not estimated)
fixed <- names(coef(gomp))[!names(coef(gomp)) %in% estimated]

profile_starts <- profile_design(
  r = seq(0.05, 0.2, length = switch(run_level, 10, 10, 20)),
  lower = c(coef(gomp)[estimated] / 2, coef(gomp)[fixed])[-1],
  upper = c(coef(gomp)[estimated] * 2, coef(gomp)[fixed])[-1],
  nprof = 2, type = "runif"
)

profile_results <- foreach(start=iter(profile_starts,"row")) %dopar% {
  mf <- mif2(
    mif_results[[1]],
    start = unlist(start),
    rw.sd = rw_sd(sigma = 0.02, tau = 0.02))
  mf@specific["tau", ] <- sapply(1:length(mf), mif_unit,mifd_gomp = mf)
  mf
}
```

The profile searches took 68.2 minutes. However, we are not quite done gathering the results for the profile. The perturbed filtering carried out by `mif2` leads to an approximate likelihood evaluation, but for our main results it is better to re-evaluate the likelihood without perturbations. Also, replication is recommended to reduce and quantify Monte Carlo error. We do this, and tabulate the results.

```
profile_table <- foreach(mf=profile_results,.combine=rbind) %dopar% {
  LL <- replicate(switch(run_level,2,5,10),
    logLik(pfilter(mf,Np=switch(run_level,10,500,2500)))
  )
  LL <- logmeanexp(LL,se=TRUE)
```

```
  data.frame(t(coef(mf)),loglik=LL[1],loglik.se=LL[2])
}
```

The likelihood evaluations took 3.2 minutes. It is appropriate to spend comparable time evaluating the likelihood to the time spent maximizing it: a high quality maximization without high quality likelihood evaluation is hard to interpret, whereas good evaluations of the likelihood in a vicinity of the maximum can inform about the shape of the likelihood surface in this region which may be as relevant as knowing the exact maximum.

The Monte Carlo adjusted profile (MCAP) approach of Ionides et al. (2017) is implemented by the `mcap()` function in `pomp`. This function constructs a smoothed profile likelihood, by application of the `loess` smoother. It computes a local quadratic approximation that is used to derive an extension to the classical profile likelihood confidence interval that makes allowance for Monte Carlo error in the calculation of the profile points. Theoretically, an MCAP procedure can obtain statistically efficient confidence intervals even when the Monte Carlo error in the profile likelihood is asymptotically growing and unbounded (Ning et al., 2021). Log likelihood evaluation has negative bias, as a consequence of Jensen's inequality for an unbiased likelihood estimate. This bias produces a vertical shift in the estimated profile, which fortunately does not have consequence for the confidence interval if the bias is slowly varying.

The profile points evaluated above, and stored in `profile_table`, can be used to compute a 95% MCAP confidence interval as follows:

```
gomp_mcap <- pomp::mcap(logLik=profile_table$loglik,
  parameter=profile_table$r,
  level=0.95)
```

The construction of the confidence interval is best shown by a plot of the smoothed profile likelihood (Fig. 3). In this toy example, the exact likelihood can be calculated using the Kalman filter, and this is carried out by the `panelGompertzLikelihood` function. The likelihood can then be maximized using a general-purpose optimization procedure such as `optim()` in R. With large numbers of parameters, and no guarantee of convexity, this numerical optimization is not entirely routine. One might consider a block optimization strategy, but here we carry out a simple global search, which took 15.5 minutes to compute the profile likelihood, once parallelized. The deterministic search is also not entirely smooth, and so we apply MCAP as for the Monte Carlo search. Both deterministic and Monte Carlo optimizations can benefit from a block optimization strategy which alternates between shared and unit-specific parameters (Bretó et al., 2020). Such algorithms can be built

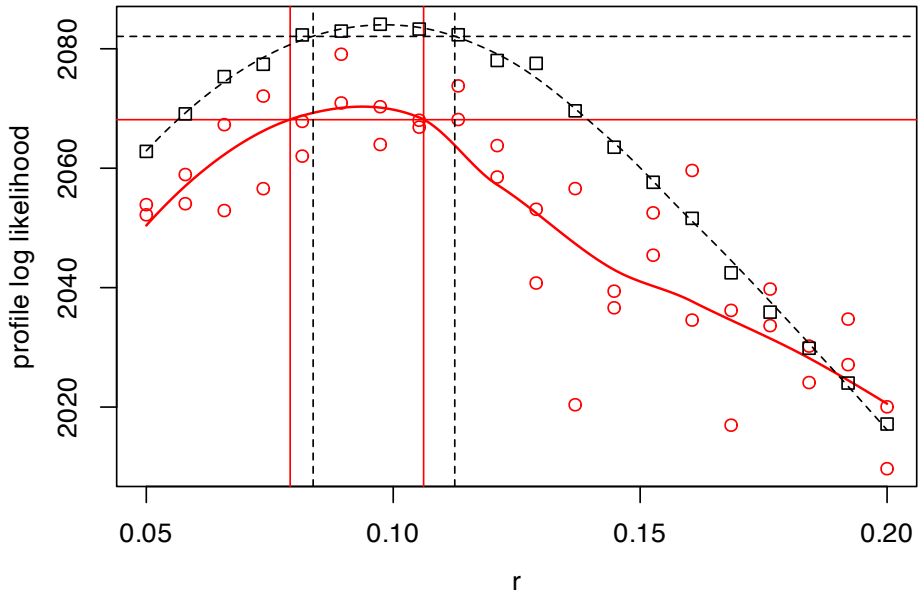

Figure 3: The Monte Carlo adjusted profile confidence interval (solid red lines, evaluation points shown as circles). Construction using deterministic optimization of the likelihood calculated by the Kalman filter (dashed lines, evaluation points show as squares).

using the `panelPomp` functions we have demonstrated, and they will be incorporated into the package once they have been more extensively researched.

5 Conclusion

The analysis demonstrated in Section 4 gives one approach to plug-and-play inference for PanelPOMP models, but the scope of `panelPomp` is far from limited to this approach. `panelPomp` is a general and extensible framework which encourages the development of additional functionality. The `panelPomp` class and the corresponding workhorse functions provide an applications interface available to other future methodologies. In this sense, `panelPomp` provides an environment for sharing and developing PanelPOMP models and methods, both via future contributions to the `panelPomp` package and via open source applications using `panelPomp`. This framework will facilitate comparison of new future methodology with existing methodology.

Likelihood evaluation and maximization was used to construct confidence intervals in Section 4. These calculations also provide a foundation for other techniques of likelihood-based inference, such as likelihood ratio hypothesis tests and model selection via Akaike's information criterion (AIC). The examples discussed in Section 3 provide case studies in the use of these methods for scientific work.

Data analysis using large data sets or complex models may require considerable computing time. Simulation-based methodology is necessarily computationally intensive, and access to a cluster computing environment extends the size of problems that can be tackled. The workflow in Section 4 has a simple parallel structure that can readily take advantage of additional resources. Embarrassingly parallel computations, such as computing the profile likelihood function at a grid of points, or replicated evaluations of the likelihood function, can be parallelized using the `foreach` package.

Panel data is widely available: for many experimental and observational systems it is more practical to collect short time series on many units than to obtain one long time series. For time series data, fitting mechanistic models specified as partially observed Markov processes has found numerous applications for formulating and answering scientific hypotheses. (Bretó et al., 2009; King et al., 2016). However, there are remarkably few examples in the literature fitting mechanistic nonlinear non-Gaussian partially observed stochastic dynamic models to panel data. The `panelPomp` package offers opportunities to remedy this situation.

The source code for `panelPomp` is at `https://github.com/cbreto/panelPomp`. Unit tests that cover 100% of the code are provided at `https://github.com/cbreto/panelPomp/tests`, and these tests also provide useful examples of calls to the functions within `panelPomp`. The source code for this article is at `https://github.com/cbreto/panelPomp/vignettes/articles/package_tutorial`.

Acknowledgments

The results in this paper were obtained using `R` 4.4.1 with `panelPomp` 1.3.0 and pkgpomp 5.9.0.0. This work was supported by National Science Foundation grants DMS-1761603 and DMS-1646108, National Institutes of Health grants 1-U54-GM111274 and 1-U01-GM110712, and by MCIN/AEI/10.13039/501100011033 grants PID2020-116242RB-I00 and PID2023-152348NB-I00

References

Asfaw, K., Park, J., King, A. A., and Ionides, E. L. (2024), "A tutorial on spatiotemporal partially observed Markov process models via the R package spatPomp," *arXiv:2101.01157*.

Bretó, C. (2014), "On idiosyncratic stochasticity of financial leverage effects," *Statistics & Probability Letters*, 91, 20–26.

Bretó, C. (2018), "Modeling and inference for infectious disease dynamics: a likelihood-based approach," *Statistical Science*, 33, 57–69.

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

Bretó, C., Ionides, E. L., and King, A. A. (2020), "Panel data analysis via mechanistic models," *Journal of the American Statistical Association*, 115, 1178–1188, URL `https://doi.org/10.1080/01621459.2019.1604367`.

Chambers, J. (1998), *Programming with Data*, New York: Springer-Verlag.

Domeyer, J. E., Lee, J. D., Toyoda, H., Mehler, B., and Reimer, B. (2022), "Driver-pedestrian perceptual models demonstrate coupling: implications for vehicle automation," *IEEE Transactions on Human-Machine Systems*, 52, 557–566.

FitzJohn, R. G., Knock, E. S., Whittles, L. K., Perez-Guzman, P. N., Bhatia, S., Guntoro, F., Watson, O. J., Whittaker, C., Ferguson, N. M., Cori, A., et al. (2020), "Reproducible parallel inference and simulation of stochastic state space models using odin, dust, and mcstate," *Wellcome Open Research*, 5, 288.

Genolini, C. (2008), "A (Not So) Short Introduction to S4," R Foundation for Statistical Computing, URL `https://CRAN.R-project.org/doc/contrib/Genolini-S4tutorialV0-5en.pdf`.

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

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

Ionides, E. L., Breto, C., Park, J., Smith, R. A., and King, A. A. (2017), "Monte Carlo profile confidence intervals for dynamic systems," *Journal of the Royal Society Interface*, 14, 1–10.

Ionides, E. L., Nguyen, D., Atchadé, Y., Stoev, S., and King, A. A. (2015), "Inference for dynamic and latent variable models via iterated, perturbed Bayes maps," *Proceedings of the National Academy of Sciences of the USA*, 112, 719–724.

King, A. A., Nguyen, D., and Ionides, E. L. (2016), "Statistical inference for partially observed Markov processes via the R package pomp," *Journal of Statistical Software*, 69, 1–43.

Lee, E. C., Chao, D. L., Lemaitre, J. C., Matrajt, L., Pasetto, D., Perez-Saez, J., Finger, F., Rinaldo, A., Sugimoto, J. D., Halloran, M. E., Longini, I. M., Ternier, R., Vissieres, K., Azman, A. S., Lessler, J., and Ivers, L. C. (2020), "Achieving coordinated national immunity and cholera elimination in Haiti through vaccination: A modelling study," *The Lancet Global Health*, 8, e1081–e1089.

Marino, J. A., Peacor, S. D., Bunnell, D. B., Vanderploeg, H. A., Pothoven, S. A., Elgin, A. K., Bence, J. R., Jiao, J., and Ionides, E. L. (2019), "Evaluating consumptive and nonconsumptive predator effects on prey density using field times series data," *Ecology*, 100, e02583.

Marjoram, P., Molitor, J., Plagnol, V., and Tavaré, S. (2003), "Markov chain Monte Carlo without likelihoods," *Proceedings of the National Academy of Sciences*, 100, 15324–15328.

Martinez-Bakker, M., King, A. A., and Rohani, P. (2015), "Unraveling the transmission ecology of polio," *PLoS Biology*, 13, e1002172.

Michaud, N., de Valpine, P., Turek, D., Paciorek, C. J., and Nguyen, D. (2021), "Sequential Monte Carlo methods in the nimble and nimbleSMC R packages," *Journal of Statistical Software*, 100, 1–39.

Ning, N., Ionides, E. L., and Ritov, Y. (2021), "Scalable Monte Carlo inference and rescaled local asymptotic normality," *Bernoulli*, 27, 2532–2555.

Ranjeva, S., Subramanian, R., Fang, V. J., Leung, G. M., Ip, D. K., Perera, R. A., Peiris, J. M., Cowling, B. J., and Cobey, S. (2019), "Age-specific differences in the dynamics of protective immunity to influenza," *Nature Communications*, 10, 1660.

Ranjeva, S. L., Baskerville, E. B., Dukic, V., Villa, L. L., Lazcano-Ponce, E., Giuliano, A. R., Dwyer, G., and Cobey, S. (2017), "Recurring infection with ecologically distinct HPV types can explain high prevalence and diversity," *Proceedings of the National Academy of Sciences of the USA*, 114, 13573–13578.

Romero-Severson, E., Volz, E., Koopman, J., Leitner, T., and Ionides, E. (2015), "Dynamic variation in sexual contact rates in a cohort of HIV-negative gay men," *American Journal of Epidemiology*, 182, 255–262.

Sisson, S. A., Fan, Y., and Tanaka, M. M. (2007), "Sequential Monte Carlo without likelihoods," *Proceedings of the National Academy of Sciences*, 104, 1760–1765.

Wale, N., Jones, M. J., Sim, D. G., Read, A. F., and King, A. A. (2019), "The contribution of host cell-directed vs. parasite-directed immunity to the disease and dynamics of malaria infections," *Proceedings of the National Academy of Sciences*, 116, 22386–22392.

Winsor, C. P. (1932), "The Gompertz curve as a growth curve," *Proceedings of the National Academy of Sciences of the USA*, 18, 1–8.