panelPomp: Analysis of Panel Data via Partially Observed Markov Processes in R

by Carles Bretó, Jesse Wheeler, Aaron A. King, and Edward L. Ionides

Abstract Panel data arise when time series measurements are collected from multiple, dynamically independent but structurally related systems. Each system's time series can be modeled as a partially observed Markov process (POMP), and the ensemble of these models is called a PanelPOMP. If the time series are relatively short, statistical inference for each time series must draw information from across the entire panel. The component systems in the panel are called units; model parameters may be shared between units or may be unit-specific. Differences between units may be of direct inferential interest or may be a nuisance for studying the commonalities. The R package panelPomp supports analysis of panel data via a general class of PanelPOMP models. This includes a suite of tools for manipulation of models and data that take advantage of the panel structure. The panelPomp package currently highlights recent advances enabling likelihood based inference via simulation based algorithms. However, the general framework provided by panelPomp supports development of additional, new inference methodology for panel data.

1 Introduction

Collections of time series, known as panel data or longitudinal data, are commonplace across the sciences, medicine, engineering, and business. Examples include biomarkers measured over time across a panel of patients in a clinical trial (Ranjeva et al., 2017, 2019), ecological predator-prey dynamics for lake plankton measured within a season across a panel of years or lakes (Marino et al., 2019), and interactions between self-driving cars and pedestrians (Domeyer et al., 2022). Generically, we say each time series is associated with measurements on a unit. Not infrequently, the time series data available on a single unit are too short or too noisy to adequately identify parameters of interest, and yet large collections of such time series analyzed jointly may suffice. This paper presents software enabling the fitting of general nonlinear nonstationary partially observed stochastic dynamic models to panel data. The methodology provides flexibility in model specification, giving the user considerable freedom to develop models appropriate for the system under study.

Scientific motivations to fit a panel of partially observed Markov processes (PanelPOMP) model to panel data (Bretó et al., 2020) are similar to the motivations to fit a partially observed Markov process (POMP) model to time series data from a single unit. POMP models provide a general framework for mechanistic modeling of nonlinear dynamic systems (Bretó et al., 2009). However, relatively few of the many methodologies developed for time series analysis via POMPs have been extended to panel analysis. The larger datasets and higher-dimensional parameter spaces arising in PanelPOMPs have challenged the Monte Carlo methods that have proved successful for nonlinear time series. Our **panelPomp** package takes advantage of newly developed methodology for PanelPOMP models, as well as building a foundation on which additional methodology for this class of models can be built and tested.

Existing widely-used panel methodology has built on the linear Gaussian model (Croissant and Millo, 2008). Panel analysis of generalized linear models with dependence can be carried out using generalized estimating equations (Halekoh et al., 2006). By contrast, **panelPomp** is designed to facilitate analysis using arbitrary PanelPOMP models, with dynamic relationships and dependence on covariate processes specified according to scientific considerations rather than the constraints of statistical software. Beyond supplying implementations of inference algorithms for PanelPOMP models, the **panelPomp** package provides a framework for developing and sharing new models and methods. This is done by providing a way of writing basic mathematical functions that comprise a general

PanelPOMP model as easy-to-write C-code snippets (King et al., 2016), which makes the **panelPomp** package both computationally efficient and readily adaptable to new developments in methodology for PanelPOMP models.

Analysis of experimental or observational studies often assumes independent outcomes for units given their treatment, and here we take the same approach for panel models. That is, the underlying dynamic process that is used to define a PanelPOMP model is assumed to be independent across units. A collection of POMP models with dynamic dependence between units is called a SpatPOMP, a name motivated by the dependence between nearby locations in spatiotemporal models. Software for general SpatPOMP models is available in the **spatPomp** package (Asfaw et al., 2024). Addressing SpatPOMP models introduces complexities in both software and methodologies that can be avoided when the dynamic processes are independent. The goal of the **panelPomp** package is to take full advantage of the independence inherent in the PanelPOMP model structure.

2 Statistical models

The general scope of the **panelPomp** package requires notation concerning random variables and their densities in arbitrary spaces. The notation below allows us to talk about these things using the language of mathematics, enabling precise description of models and algorithms.

Units of the panel can be identified with numeric labels $\{1, 2, \dots, U\}$, which we also write as $1:U$. Let N_u be the number of measurements collected on unit u, and write the data as $y^*_{u,1:N_u} = \{y^*_{u,1}, \dots, y^*_{u,N_u}\}$ where $y^*_{u,n}$ is collected at time $t_{u,n}$ with $t_{u,1} < t_{u,2} < \cdots < t_{u,N_u}$. The data are modeled as a realization of an observable stochastic process $Y_{u,1:N_u}$ which is 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 $\{Y_{u,i}, i \ne n\}$ 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 at the observation times is written $X_{u,n} = X_u(t_{u,n})$. We suppose that $X_{u,n}$ and $Y_{u,n}$ take values in arbitrary spaces $\mathbb{X}_u$ and $\mathbb{Y}_u$ respectively. Using the independence of units, conditional independence of the observable random variables, and the Markov property of the latent states, the joint distribution of the entire collection of latent variables $\mathbf{X} = \{X_{u,0:N_u}\}_{u=1}^U$ and observable variables $\mathbf{Y} = \{Y_{u,1:N_u}\}_{u=1}^U$ can be written as

$$f_{\mathbf{XY}}(\mathbf{x}, \mathbf{y}) = \prod_{u=1}^{U} f_{X_{u,0}}(x_{u,0}; \theta) \prod_{n=1}^{N_u} f_{Y_{u,n}|X_{u,n}}(y_{u,n}|x_{u,n}; \theta) f_{X_{u,n}|X_{u,n-1}}(x_{u,n}|x_{u,n-1}; \theta),$$

where $\theta \in \mathbb{R}^D$ is a parameter vector. This representation is useful as it demonstrates how any PanelPOMP model can be fully described using three primary components: the unit transition densities $f_{X_{u,n}|X_{u,n-1}}(x_{u,n} \mid x_{u,n-1}\,;\,\theta)$, measurement densities $f_{Y_{u,n}|X_{u,n}}(y_{u,n} \mid x_{u,n}\,;\,\theta)$, and initialization densities $f_{X_{u,0}}(x_{u,0}; \theta)$. Each class of densities are permitted to depend arbitrarily on u and n, allowing non-stationary models and the inclusion of covariate time series. In addition to continuous-time dynamics, the framework includes discrete-time dynamic models by specifying $X_{u,0:N_u}$ directly without ever defining $\{X_u(t), t_{u,0} \le t \le t_{u,N_u}\}$. We also permit the possibility that some parameters may affect only a subset of units, so that the parameter vector can be written as $\theta = (\phi, \psi_1, \dots, \psi_U)$, where the densities described above can be written as

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

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 $\psi_u \in \mathbb{R}^B$, so the dimension of the parameter vector θ is $D = A + BU$. In practice, the densities in Eqs. (1)–(3) serve two primary roles in PanelPOMP models: evaluation and simulation. Each function can depend on the unit to which it belongs, and therefore are specified in the unit-specific POMP models that together define the PanelPOMP. This feature is reflected in the software implementation, where the fundamental tasks of simulation and evaluation are defined in the unit `pomp` objects, as described in in Table 1.

Method	Operation	Function
`rprocess`	Simulate from Eq. (1)	$f_{X_{u,n}\|X_{u,n-1}}(x_{u,n} \mid x_{u,n-1}\,;\,\phi,\psi_u)$
`dprocess`	Evaluate Eq. (1)	$f_{X_{u,n}\|X_{u,n-1}}(x_{u,n} \mid x_{u,n-1}\,;\,\phi,\psi_u)$
`rmeasure`	Simulate from Eq. (2)	$f_{Y_{u,n}\|X_{u,n}}(y_{u,n} \mid x_{u,n}\,;\,\phi,\psi_u)$
`dmeasure`	Evaluate Eq. (2)	$f_{Y_{u,n}\|X_{u,n}}(y_{u,n} \mid x_{u,n}\,;\,\phi,\psi_u)$
`rinit`	Simulate from Eq. (3)	$f_{X_{u,0}}(x_{u,0}\,;\,\phi,\psi_u)$
`dinit`	Evalutate Eq. (3)	$f_{X_{u,0}}(x_{u,0}\,;\,\phi,\psi_u)$

Table 1: Methods of a unit model in a `panelPomp` object and their mathematical definitions.

In addition to the functions listed in in Table 1, additional basic mathematical functions can be specified for the unit objects of a `panelPomp` model as needed. For instance, functions `rprior` and `dprior`—which represent the operations of simulating from and evaluating a prior density function, respectively—can be specified as part of the unit objects if desired. The current version of the package (`1.7.0.0`), however, currently emphasizes the likelihood-based, plug-and-play methodologies described in Section 3, which do not require the specification of a prior distribution.

2.1 Model representation in panelPomp

The package uses the S4 functional object oriented programming approach in R (Wickham, 2019) in order to represent PanelPOMP models as `panelPomp` objects. Because a PanelPOMP model comprises multiple POMP models, a `panelPomp` object is essentially a structured collection of `pomp` objects from the **pomp** package (King et al., 2016). The `panelPomp` class contains three attributes: `unit_objects`, a list containing the models for each unit; `shared`, a named numeric vector of parameters that are shared across all units; and `specific`, a matrix of parameters that are unique to each unit.

Typically, creating `panelPomp` objects involves supplying a dataset for each unit and explicitly defining the mathematical functions in Table 1. The functions can be defined either using `R` functions or C-code snippets, the latter offering the advantage of reduced computing times. The construction of `pomp` objects using both `R` functions and C-code snippets is a topic discussed in detail in King et al. (2016) and various online tutorials for the **pomp** package. To avoid redundancy with this existing material, we focus instead on the novel features of the **panelPomp** package. Specifically, we illustrate how to construct a `panelPomp` object when a collection of `pomp` objects is already available.

For demonstration purposes, we consider a stochastic version of the discrete-time Gompertz population model (Winsor, 1932). This model is a popular choice for representing the exponential growth and decay observed in numerous ecological populations (Auger-Méthé et al., 2021; Smith et al., 2023; Lindén and Knape, 2009). It serves as a useful example due to its nonlinear, non-Gaussian nature, which can be transformed into a linear Gaussian model through log transformations. Consequently, the model is a popular choice for demonstrating the capabilities of plug-and-play algorithms on a nonlinear, non-Gaussian model where the likelihood can be exactly calculated using linear Gaussian techniques (Bretó et al., 2020). It further serves as an illustrative case study for **panelPomp**, as researchers may use similar models to describe population dynamics at distinct observation sites or experimental treatments. This approach allows for formal statistical testing to determine whether characteristics of the population dynamics are shared across populations or are unique to each unit.

For each unit u, the latent population density $X_{u,n}$ at time n is recursively modeled according to Eq. (4).

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

where $\epsilon_{n,u}$ are independent and identically-distributed log-normal random variables with $\log \epsilon_{u,n} \sim \mathrm{N}(0, \sigma_u^2)$. The observed population $Y_{u,n}$ at time n is assumed to follow a log-normal distribution, independently and identically distributed, conditioned on the value $X_{u,n}$,

$$\log Y_{u,n} | X_{u,n} \sim \mathrm{N}\big(\log X_{u,n}, \tau_u^2\big).$$

Below, we use the `pomp::gompertz` constructor function to build these unit-specific models. These constructors simultaneously build the unit models described above, and store a simulation from the model in the `data` attribute. The `times` argument in the constructor indicates the observation times for each unit; these times do not need to be the same when constructing `panelPomp` objects, as highlighted below.

```
gomp_u1 <- pomp::gompertz(
  K = 1, r = 0.1, sigma = 0.1, tau = 0.1, X_0 = 1, times = 1:20, seed = 111
)

gomp_u2 <- pomp::gompertz(
  K = 1.5, r = 0.1, sigma = 0.1, tau = 0.07, X_0 = 1, times = 1:21, seed = 222
)

gomp_u3 <- pomp::gompertz(
  K = 1.2, r = 0.1, sigma = 0.1, tau = 0.15, X_0 = 1, times = 3:25, seed = 333
)
```

In this example, we construct three unit objects, `gomp_u1`, `gomp_u2`, and `gomp_u3`, each of which is a `pomp` object. Each object contains a vector of parameters `K`, `r`, `sigma` `tau`, and `X_0` that correspond to the parameters $\kappa_u, r_u, \sigma_u, \tau_u$ and $X_{u,0}$, respectively. To build a `panelPomp` object, we pass a list of these models and specify which parameters are `shared` across units and which are `specific` to each unit.

```
gomp <- panelPomp(
  object = list(gomp_u1, gomp_u2, gomp_u3),
  shared = c("r" = 0.1, "sigma" = 0.1),
  specific = c("K", "tau", "X_0")
)
```

The parameter values of the `shared` vector need to be explicitly defined, replacing the separate values in the parameter vectors for each unit object. The unit-specific parameters are extracted from the constituent `pomp` objects. In the above construction, we are creating a `panelPomp` model that has shared parameters `r` and `sigma`. Mathematically, this is equivalent to assuming that, for all $u \in 1:U$, $r_u = r = 0.1$ and $\sigma_u = \sigma = 0.1$. The choice of which parameters are shared and which are unit-specific can be modified, as the helper function described in the following subsection enables changing the original parameter specifications.

Several pre-built models are also included in the package via constructor functions, including:

- `contacts()` creates a dynamic model for the variation in sexual contacts for the data of Vittinghoff et al. (1999). The model supposes each individual has a latent rate of making sexual contacts that evolves over time, allowing for heterogeneity between and within individuals, auto-correlation in individual rates over time, and a trend in rates over the time of the study (Romero-Severson et al., 2015).

- `panelMeasles()` creates a PanelPOMP model for measles incidence data in several UK cities, based on the model of He et al. (2010). This model is a stochastic compartmental model that describes Measles incidence data using a susceptible, exposed, infected, recovered (SEIR) model. The source code provides a useful demonstration of how users can build compartmental models for panel data from an infectious disease outbreak.
- `panelRandomWalk()` constructs a collection of independent latent Gaussian random walk models. After building the panel model, the constructor populates the `data` slots for the created `unitObjects` by simulating from the created model.
- `panelGompertz()` creates a collection of the stochastic Gompertz population models described in this section. The `data` slots for the `unitObjects` are populated by using simulations from the model.

The primary purpose of these pre-built constructor functions is to provide source code demonstrating how users may create a variety of different `panelPomp` objects, and to enable quick testing and comparison of newly developed PanelPOMP methodology using existing models and data. The parent **pomp** package also contains a number of pre-built models and datasets for similar purposes. In cases where there may be confusion between the constructor functions in these packages, the `panel` prefix is used to clarify the distinction.

2.2 Generic methods for `panelPomp` objects

The created object from the previous section, `gomp`, is a PanelPOMP model with 3 independent units, each with a unique number of observations specified by the length of the `times` argument in their constructor function. In this model κ_u, τ_u and $X_{u,0}$ are treated as unit-specific, and $r_u = r, \sigma_u = \sigma$ are shared for all unit objects. Internally, each `unit_object` treats both types of parameters the same, and thereby the construction of the unit objects does not require renaming unit-specific parameters to allow for distinction across units. For example, though κ_u may be unique for each unit u, the internal functions representing the process model (Eq. (1)) can use the variable name `kappa` to represent the parameter in all unit objects, rather than needing unique variable names for each unit. This important feature allows for changing which parameters are treated as shared and which are unit-specific without having to redefine each of the unit objects.

Model parameters can be extracted and modified using the `coef()` generic function. The default treatment of this function is the convention `<parameter name>[<unit name>]` for unit-specific parameters, and `<parameter name>` for shared parameters. This format is intended to closely reflect the standard mathematical notation for unit-specific parameters. For instance, to change the value of κ_2, we could refer to the corresponding parameter `K[unit2]` in the `gomp` object:

```
coef(gomp)['K[unit2]'] <- 0.9
coef(gomp)
```

```
#>          r      sigma   K[unit1] tau[unit1] X_0[unit1]   K[unit2] tau[unit2]
#>       0.10       0.10       1.00       0.10       1.00       0.90       0.07
#> X_0[unit2]   K[unit3] tau[unit3] X_0[unit3]
#>       1.00       1.20       0.15       1.00
```

It can be more convenient to view unit-specific parameters as a matrix and shared parameters as a vector. This can be done using the `format = 'list'` argument of the `coef()` function, or by alternatively using the `shared()` and `specific()` functions to extract only the shared or unit-specific parameters, respectively.

```
coef(gomp, format = 'list')
```

```
#> $shared
```

```
#>     r sigma
#>   0.1   0.1
#>
#> $specific
#>      unit
#> param unit1 unit2 unit3
#>   K     1.0  0.90  1.20
#>   tau   0.1  0.07  0.15
#>   X_0   1.0  1.00  1.00

shared(gomp)

#>     r sigma
#>   0.1   0.1
```

The `shared()<-` and `specific()<-` setter functions are also convenient for modifying which model parameters are considered shared and unit-specific.

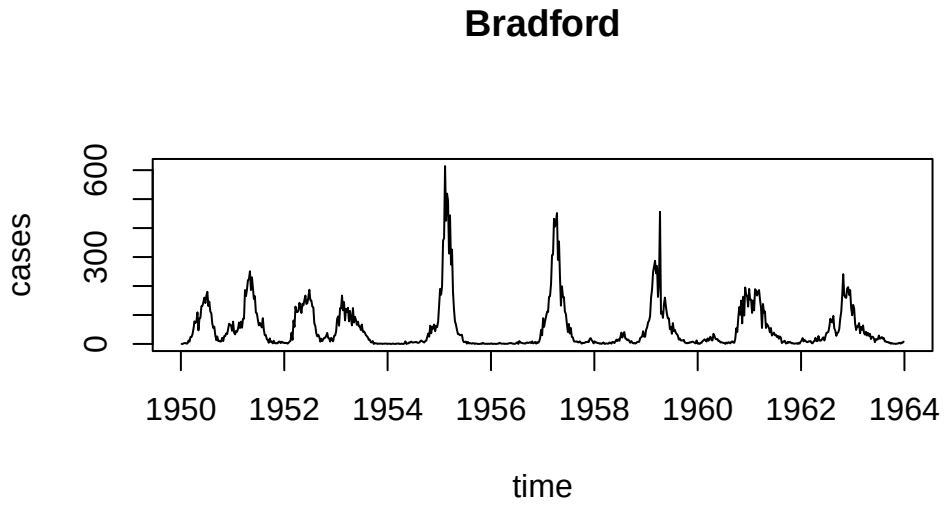
```
shared(gomp) <- c("tau" = 0.15)
shared(gomp)

#>   tau     r sigma
#>  0.15  0.10  0.10

specific(gomp)

#>      unit
#> param unit1 unit2 unit3
#>   K       1   0.9   1.2
#>   X_0     1   1.0   1.0
```

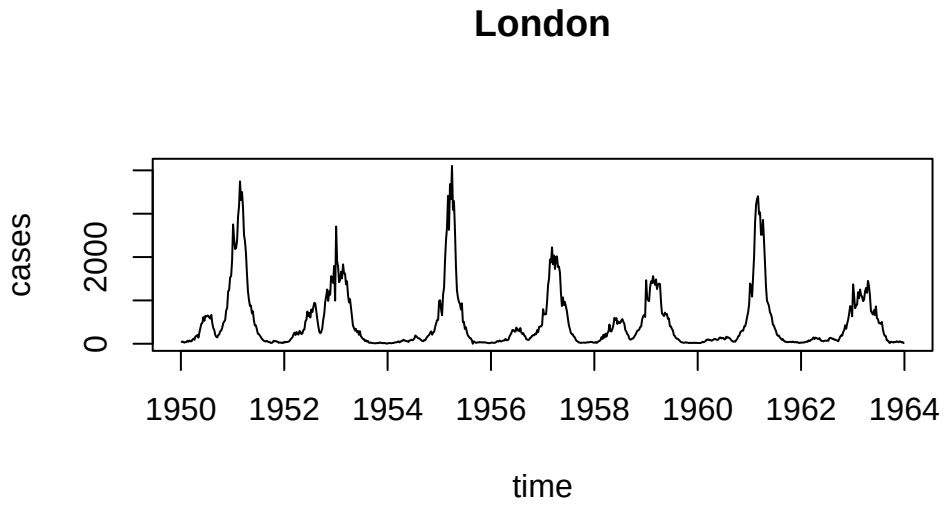
Before altering the definitions of existing parameters, these functions verify whether the parameters are already present in the model. They enable changing parameter values and adjusting whether a parameter is unit-specific or shared, but do not allow for the creation or deletion of model parameters.

A non-exhaustive list of useful methods that are frequently applied to `panelPomp` objects in the course of a data analysis includes:

- `simulate()`. For `panelPomp` objects that contain units with the functions `rinit`, `rprocess`, and `rmeasure`, the `simulate()` function can generate a number of simulations, specified by the `nsim` argument, which defaults as `nsim = 1`. The resulting object is a `panelPomp` object where the simulated results are saved as data in the unit objects.
- `plot()`. This function plots each unit of a `panelPomp` object sequentially. For derived classes, such as `pfilterd.ppomp` resulting from applying `pfilter` to a `panelPomp`, or `mif2d.ppomp` resulting from an application of `mif2`, diagnostic plots are produced for each unit. For example, calling `plot()` on the measles model constructed by the `panelMeasles()` function will plot the measles data available for the specified UK cities:

```
plot(panelMeasles(), units = c('Bradford', 'London'))
```





- `as()`. Many essential features of the basic model components of a `panelPomp` object have already undergone extensive development and testing in the context of the **pomp** package. To avoid unnecessary duplication, the `as()` function offers a way to convert a `panelPomp` object into a list of `pomp` objects. For example, `as(gomp, "list")` and `as(gomp, "pompList")` will convert the `gomp` `panelPomp` object into a `list` or `pompList`, respectively. This is particularly useful when it is appropriate to consider the individual components of the `panelPomp` object.

3 Inference methodology

All POMP methods can in principle be extended to PanelPOMPs; three different ways to represent a PanelPOMP as a POMP were identified by Romero-Severson et al. (2015). The ability to represent a PanelPOMP model as a POMP model, however, does not imply that methodology for POMP models will be feasible for PanelPOMP models. In particular, sequential Monte Carlo algorithms can have prohibitive scaling difficulties with the high dimensionality that arises in PanelPOMP models.

The current version of **panelPomp** emphasizes plug-and-play methods (Bretó et al., 2009; He et al., 2010), also known as likelihood-free (Marjoram et al., 2003; Sisson et al., 2007), that are applicable to dynamic models for which a simulator is available even when the transition densities are unavailable. In the terminology used in this article, this means that `dprocess` (Eq. (1)) is not needed in order to perform inference. This class of algorithms includes methods such as a particle filter (Arulampalam et al., 2002) and the panel iterated filter (PIF) (Bretó et al., 2020), which are methods that can be used to evaluate and maximize model likelihoods, respectively. In this section, we describe and demonstrate a plug-and-play likelihood-based inference workflow using the Stochastic Gompertz population model described previously. To highlight the capability of **panelPomp** to perform inference on high-dimensional models, we reconstruct a model with $U = 50$ measurement units and $N = N_u = 100$ observations per unit using the `panelGompertz()` constructor function.

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

3.1 Log-likelihood evaluation via particle filters

The particle filter, also known as sequential Monte Carlo, is a standard tool for log-likelihood evaluation on nonlinear non-Gaussian POMP models. The log-likelihood function is a central component of Bayesian and frequentist inference. Due to the dynamic independence between units that is a defining feature of a PanelPOMP model, 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. All mathematical functions needed to carry out the particle filter for a PanelPOMP model are represented by the functions `rinit`, `rprocess`, `rmeasure` and `dmeasure` (Eqs. (1)–(3)).

Methods applied to `panelPomp` objects—like the `pfilter` function—typically create new objects of the same class or a child class of the original object. For instance, we can perform a particle filter on the `gomp` object that contains 50 units, and 100 observations per unit in the following way:

```
gomp_pfd <- pfilter(gomp, Np = 1000)
```

In this case, we used `Np` = 1000 particles to obtain a single stochastic estimate of the log-likelihood of the `gomp` model. The resulting object `gomp_pfd` is of class `pfilterd.ppomp`, which is a child class of `panelPomp` and contains the estimated log-likelihood of each unit object, as well as the log-likelihood of the entire panel; these can be accessed using the `unitLogLik()` and `logLik()` functions, respectively.

In practice it is advisable to repeat this Monte Carlo approximation in order to reduce and quantify the error associated with the estimate. Because sequential Monte Carlo algorithms can be computationally expensive, we obtain replicated estimates by taking advantage of multicore computation using the **foreach** (Microsoft and Weston, 2022b) and **doParallel** (Microsoft and Weston, 2022a) packages:

```
pf_results <- foreach(i = 1:10) %dopar% {
    pfilter(gomp, Np = 1000)
}
```

This took 2.68 seconds to run the 10 replicates in parallel, resulting in a list of objects of class `pfilterd.ppomp`. We can use the `logLik` function to extract the Monte Carlo estimate of the log-likelihood $\lambda^{[i]}$ for each replicate i, and `unitLogLik` to extract the vector of component Monte Carlo log-likelihood estimates $\lambda_u^{[i]}$ for each unit $u = 1, \ldots, U$, where $\lambda^{[i]} = \sum_{u=1}^{U} \lambda_u^{[i]}$. For a POMP model, replicated log-likelihood evaluations via the particle filter 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

$$\hat{\lambda}_1 = \log\Big(\frac{1}{I}\sum_{i=1}^{I}\exp\big\{\sum_{u=1}^{U}\lambda_u^{[i]}\big\}\Big),$$

which can be implemented as

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

giving $\hat{\lambda}_1 = 2065.4$ with a jack-knife standard error of 0.5. 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

$$\hat{\lambda}_2 = \log\Big(\prod_{u=1}^{U}\frac{1}{I}\sum_{i=1}^{I}\exp\big\{\hat{\lambda}_u^{[i]}\big\}\Big),$$

which can be obtained using

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

giving $\hat{\lambda}_2 = 2068.5$ with a jack-knife standard error of 1.1. For this model, a Kalman filter log-likelihood evaluation gives an exact answer, $\lambda = 2068.2$.

3.2 Maximum likelihood estimation

Although particle filters can effectively approximate the log-likelihood of non-linear models, it is well known that obtaining a maximum likelihood estimate for fixed parameters using these filters is difficult in practice. To maximize the likelihood, we employ iterated filtering algorithms, which perform repeated particle filtering on an extended version of the model that incorporates time-varying parameter perturbations. With each iteration, the magnitude of these perturbations is reduced, allowing the algorithm to approach a local maximum of the likelihood function. An example of this type of algorithm for general POMP models includes the IF2 iterated filtering algorithm (Ionides et al., 2015). IF2 has been successfully employed for likelihood-based inference in various POMP models, particularly in epidemiology and ecology, as reviewed by Bretó (2018). However, both particle filters and IF2 face scalability issues as model dimensions increase, and IF2 cannot be applied to each unit separately when the PanelPOMP model includes shared parameter values.

A panel iterated filtering (PIF) algorithm was developed by Bretó et al. (2020), extending IF2 to panel data. An implementation of PIF in **panelPomp** is provided by the `mif2` method for class `panelPomp`, following the pseudocode in Algorithm 1. The pseudocode 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 Gaussian perturbations of parameters on a transformed scale. The theory allows considerable flexibility in how the parameters are perturbed, but Gaussian perturbations on an appropriate scale are typically adequate.

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. 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 that 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 on using additional structure should it become necessary.

The unmarginalized PIF was proposed by Bretó et al. (2020), who provided theoretical convergence results for the algorithm. When MARGINALIZE = TRUE, the PIF algorithm

Algorithm 1: `mif2(pp, Nmif =` M`, Np =` J`, start =` $(\phi_a^0, \psi_{b,u}^0)$`, rw_sd =` $(\sigma_{a,n}^{\Phi}, \sigma_{b,u,n}^{\Psi})$`, 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, $f_{X_{u,n}|X_{u,n-1}}(x_{u,n} \mid x_{u,n-1}\,;\phi,\psi_u)$
Evaluator of measurement density, $f_{Y_{u,n}|X_{u,n}}(y_{u,n} \mid x_{u,n}\,;\phi,\psi_u)$
Number of particles, J, and number of iterations, M
Starting shared parameter swarm, $\Phi_{a,j}^0 = \phi_a^0$, a in $1:A$, j in $1:J$
Starting unit-specific parameter swarm, $\Psi_{b,u,j}^0 = \psi_{b,u}^0$, b in $1:B$, j in $1:J$
Random walk intensities, $\sigma_{a,n}^{\Phi}$ and $\sigma_{b,u,n}^{\Psi}$
Parameter transformations, h_a^{Φ} and h_b^{Ψ}, with inverses $\left(h_a^{\Phi}\right)^{-1}$ and $\left(h_b^{\Psi}\right)^{-1}$
Logical variable determining marginalization, MARGINALIZE

output: Final parameter swarm, $\Phi_{a,j}^M$ and $\Psi_{b,u,j}^M$

For m in $1:M$
  $\Phi_{a,0,j}^m = \Phi_{a,j}^{m-1}$
  For u in $1:U$
    $\Phi_{a,u,0,j}^{F,m} = \left(h_a^{\Phi}\right)^{-1}\left(h_a^{\Phi}\left(\Phi_{a,u-1,j}^m\right) + \rho^m \sigma_{a,0}^{\Phi} Z_{a,u,0,j}^{\Phi,m}\right)$ for $Z_{a,u,0,j}^{\Phi,m} \sim N[0,1]$
    $\Psi_{b,u,0,j}^{F,m} = \left(h_b^{\Psi}\right)^{-1}\left(h_b^{\Psi}\left(\Psi_{b,u,j}^{m-1}\right) + \rho^m \sigma_{b,u,0}^{\Psi} Z_{b,u,0,j}^{\Psi,m}\right)$ for $Z_{b,u,0,j}^{\Psi,m} \sim N[0,1]$
    $X_{u,0,j}^{F,m} \sim f_{X_{u,0}}\left(x_{u,0} \mid \Phi_{a,u,0,j}^{F,m}, \Psi_{b,u,0,j}^{F,m}\right)$
    For n in $1:N_u$
      $\Phi_{a,u,n,j}^{P,m} = \left(h_a^{\Phi}\right)^{-1}\left(h_a^{\Phi}\left(\Phi_{a,u,n-1,j}^{F,m}\right) + \rho^m \sigma_{a,n}^{\Phi} Z_{a,u,n,j}^{\Phi,m}\right)$ for $Z_{a,u,n,j}^{\Phi,m} \sim N[0,1]$
      $\Psi_{b,u,n,j}^{P,m} = \left(h_b^{\Psi}\right)^{-1}\left(h_b^{\Psi}\left(\Psi_{b,u,n-1,j}^{F,m}\right) + \rho^m \sigma_{b,u,n}^{\Psi} Z_{b,u,n,j}^{\Psi,m}\right)$ for $Z_{b,u,n,j}^{\Psi,m} \sim N[0,1]$
      $X_{u,n,j}^{P,m} \sim f_{X_{u,n}|X_{u,n-1}}\left(x_{u,n} \mid X_{u,n-1,j}^{F,m}\,;\, \Phi_{a,u,n,j}^{P,m}, \Psi_{b,u,n,j}^{P,m}\right)$
      $w_{u,n,j}^m = f_{Y_{u,n}|X_{u,n}}\left(y_{u,n}^* \mid X_{u,n,j}^{P,m}\,;\, \Phi_{a,u,n,j}^{P,m}, \Psi_{b,u,n,j}^{P,m}\right)$
      Draw $k_{1:J}$ with $\mathbb{P}(k_j = i) = w_{u,n,i}^m \Big/ \sum_{q=1}^J w_{u,n,q}^m$
      $\Phi_{a,u,n,j}^{F,m} = \Phi_{a,u,n,k_j}^{P,m}$, $\Psi_{b,u,n,j}^{F,m} = \Psi_{b,u,n,k_j}^{P,m}$ and $X_{u,n,j}^{F,m} = X_{u,n,k_j}^{P,m}$
      If MARGINALIZE then
        $\Psi_{b,\nu,n,j}^{F,m} = \Psi_{b,\nu,n,j}^{P,m}$ for all $\nu \neq u$
      Else
        $\Psi_{b,\nu,n,j}^{F,m} = \Psi_{b,\nu,n,k_j}^{P,m}$ for all $\nu \neq u$
    End For
    $\Phi_{a,u,j}^m = \Phi_{a,u,N_u,j}^{F,m}$ and $\Psi_{b,u,j}^m = \Psi_{b,u,N_u,j}^{F,m}$
  End For
  $\Phi_{a,j}^m = \Phi_{a,U,j}^m$
End For

is modified such that the particles representing the unit-specific parameter $\psi_{b,u}$ remain unchanged when filtering through a unit $\nu \neq u$. Recent results suggest the marginalized PIF (MPIF) algorithm has superior empirical performance in many situations, but does not yet have theoretical support. For the remainder of this article, the presented results use the unmarginalized version of the algorithm.

Parameter transformations (h_a^{Φ} and h_b^{Ψ}) are used to ensure that parameter estimates remain within accepted bounds. For example, parameters that are required to be positive can be estimated on a log-transformed scale to maintain their positivity when converted back to the natural scale. This process is facilitated by the `parameter_trans` function,

which manages the transformation automatically. This relieves users from the need to handle parameters on the transformed scale directly; they only need to specify the desired transformation. In the case of the gomp model, all model parameters must be non-negative. This requirement can be met by creating unit objects and assigning the partrans argument as follows:

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

This setup ensures that the parameters κ_u, r_u, σ_u, τ_u, and $X_{u,0}$ are all estimated on a log-transformed scale, thereby maintaining their non-negativity on the natural scale. Other common parameter transformations that can be implemented include the logit transformation, which ensures parameters are in the interval $(0, 1)$, and the barycentric transformation, which is used when a collection of parameters must lie in the interval $(0, 1)$, with the additional constraint that they sum to one. Finally, custom transformations can be defined using the toEst and fromEst arguments to the parameter_trans function.

We demonstrate maximum likelihood estimation using parameter transformations for the gomp object described previously. To do so, we need to specify starting parameter values from where to start the search. This can be done by explicitly providing the starting parameters using either the start or the shared.start and specific.start arguments of the mif2() function. Alternatively, the existing parameter values in the panelPomp object will be implicitly used as a starting point, as done in the following example. For simplicity, we fix $\kappa_u = 1$ and the initial condition $X_{u,0} = 1$, maximizing over two shared parameters, r and σ, and one unit-specific parameter τ_u, starting from their default values in the gomp object:

```
gomp_mif2d <- mif2(
  gomp,  # panelPomp model, which contains parameter transformations and values.
  Nmif = 25,  # Number of Iterations (M)
  Np = 250,  # Number of Particles (J)
  cooling.fraction.50 = 0.5,  # Cooling intensity, after 50 iterations
  cooling.type = "geometric",  # Cooling Style
  rw.sd = rw_sd(r = 0.02, sigma = 0.02, tau = 0.02)  # Random Walk SD
)
```

The output gomp_mif2d is a mif2d.ppomp object, which is a child class of panelPomp. 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 τ_u here) the same perturbation is used for all units. As such, functions for coefficient extraction can be used to get the final parameter estimates, for instance:

```
shared(gomp_mif2d)

#>          r      sigma
#> 0.06833055 0.09123509
```

For Monte Carlo maximization, replication from diverse starting points is recommended. Further, larger values of Nmif and Np are typically needed in order to reliably maximize model likelihoods in practice. Smaller initial searches for parameter estimates are useful in that they can be used to estimate the computational cost of a larger search or to help determine values of hyperparameters. The computational complexity of the PIF algorithm is $O(JMNU)$, and so the cost of a larger search may be estimated by noting that the complexity is linear in each of the arguments J and M.

We demonstrate such a maximization search on gomp. The small, single PIF maximization took 40.8 seconds to compute. By increasing the number of iterations (Nmif) and particles (Np) each by a factor of 6, we expect the computation time for a single parameter initialization to

take roughly 24.5 minutes, noting that there is some overhead in the maximization function that is not increased on larger jobs. Using 36 cores, we then expect the maximization routine to take 24.5 minutes for 36 distinct starting values.

To define diverse starting points for the Monte Carlo replicates, we make uniform draws from a specified box. The parameter bounds on this box are not intended to be bounds on the final parameter estimates, as there is a possibility that the data lead the parameter search 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 facilitates the construction and drawing random points from within 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 = 36
)
```

We then carry out a search from each starting point:

```
mif_results <- foreach(start=iter(starts,"row")) %dopar% {
  mif2(
    gomp, start = unlist(start),
    Nmif = 150,
    Np = 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 15.6 minutes using 36 cores, producing a list of objects of class `mifd.ppomp`. 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).

Block optimization

A characteristic of PanelPOMP models is 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. The theoretical guarantees available for the PIF algorithm (Bretó et al., 2020) imply that this block-refinement strategy is not necessary for convergence, but empirically it can help reduce the computational effort needed to fully maximize the likelihood.

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 re-specified random walk standard deviations which ensures that only the unit-specific parameters are perturbed.

```
mif_unit <- function(unit, mifd_gomp, reps = 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 40.4 seconds to compute all 36 block refinements in parallel.

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.

3.3 Parameter uncertainty

A key component of a likelihood-based inference framework is the estimation of parameter uncertainty, often achieved by the estimation of confidence intervals. This task is particularly challenging for general non-linear, latent variable models. Some potential methods include profile likelihoods, observed Fisher information, and the bootstrap method. Here, we demonstrate the profile likelihood approach, which has proven useful for mechanistic models (Simpson and Maclaren, 2023).

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

- 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.
- 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.
- Co-plots, which show how the values of other maximized parameters vary along the profile, may provide insights into parameter trade-offs implied by the data.
- 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 = 20),
  lower = c(coef(gomp)[estimated] / 2, coef(gomp)[fixed])[-1],
  upper = c(coef(gomp)[estimated] * 2, coef(gomp)[fixed])[-1],
  nprof = 5, 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
}
```

In the above code, the `profile_design` function creates $20 \times 5 = 100$ unique starting points to perform the profile search. Parallelized over 36 cores, these 100 searches took 47.6 total 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 additional accuracy is obtained by re-evaluating 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(10, logLik(pfilter(mf, Np = 2500)))
  LL <- logmeanexp(LL, se = TRUE)
  data.frame(t(coef(mf)), loglik = LL[1], loglik.se = LL[2])
}
```

The likelihood evaluations took 2.5 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, and this 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

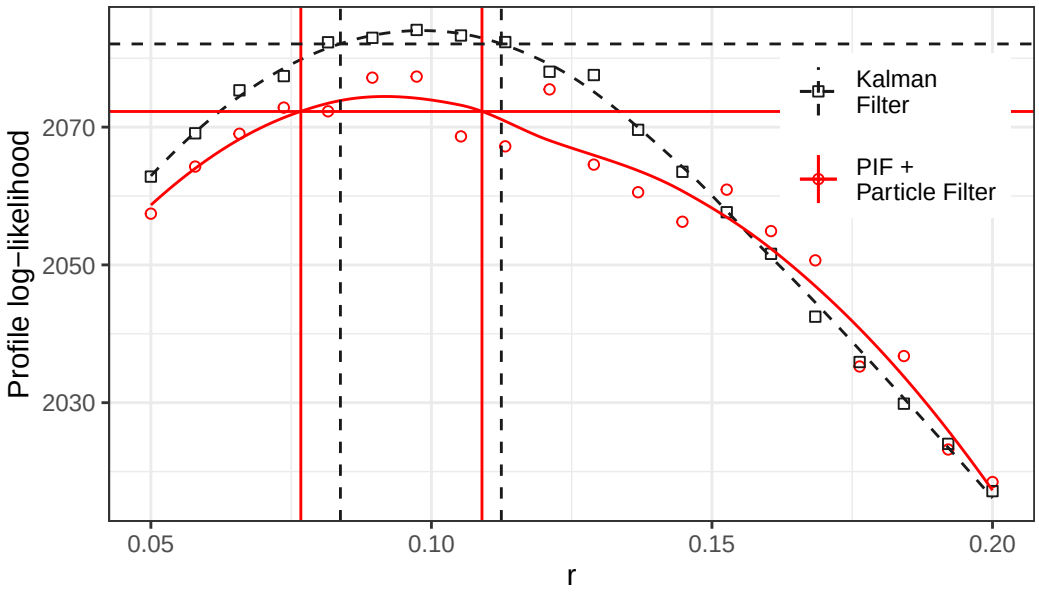


Figure 1: The Monte Carlo adjusted profile confidence interval (solid red lines, evaluation points shown as circles). Construction using deterministic optimization of the likelihood calculated by the Kalman filter (dashed lines, evaluation points show as squares).

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:

```
profile_table <- profile_table |>
  as.data.frame() |>
  dplyr::group_by(r) |>
  dplyr::slice_max(n = 1, order_by = loglik)

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, shown in Fig. 1 (Wickham, 2016). 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 1.7 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 using the **panelPomp** functions we have demonstrated, and they will be incorporated into the package once they have been more extensively researched.

4 Conclusion

The analysis using the `gomp` model illustrates 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, along with its associated workhorse functions, provide an adaptable interface that can accommodate future methodologies. In this sense, **panelPomp** provides an environment for sharing and developing PanelPOMP models and methods, both through future contributions to the **panelPomp** package and through open-source applications that leverage the package. This framework will facilitate the comparison of new methodologies with existing ones, promoting continuous improvement and innovation. Other software packages, such as **pomp**, **spatPomp**, **nimble**, among others summarized in Section 4.2 of Newman et al. (2023), can also be used to perform inference on non-linear mechanistic models. However, none of these packages specifically address the unique challenges posed by high-dimensional PanelPOMP models.

A special class of POMP models arises when the latent process takes values in a discrete and finite space. A model of this type is often referred to as a hidden Markov model (HMM) (Eddy, 2004; Doucet et al., 2001; Glennie et al., 2023; Newman et al., 2023), though this terminology has also been used as a synonym for POMP (King et al., 2016). Under this additional constraint, efficient dynamic-programming algorithms can be used to perform inference, as summarized by McClintock et al. (2020). In principle, **panelPomp** can be leveraged to implement these existing approaches for longitudinal data, though the current version of the package emphasizes methodologies for models that have latent processes with states taking values in spaces that cannot be programmatically searched.

In our example, likelihood evaluation and maximization was used to construct confidence intervals. 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 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. Our example workflow 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 are 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.

5 Availability, documentation and code quality control

panelPomp is available on CRAN and can be installed by executing `install.packages('panelPomp')`. The source code and developmental version of the package are available on GitHub: `https://github.com/panelPomp-org/panelPomp`. Package documentation is created using `roxygen2` and is shipped with the installation of the package, but can also be found at the package website: `https://panelpomp-org.github.io`. Two tutorials are provided on the website; an elementary "Getting Started" guide, and an in-depth introduction which contains examples that are similar to those in this article

(Breto et al., 2024). Continuous integration based on GitHub actions is used to build and test the package. As of writing, unit tests have a 100% line coverage (measured by the **covr** package).

All major computations were performed on a high-performance computing (HPC) node equipped with 2x 3.0 GHz Intel Xeon Gold 6154 processors, totaling 36 cores. The system ran on Red Hat Enterprise Linux 8.8 (Ootpa) on an x86_64-pc-linux-gnu platform, with 180 GB RAM. We used R version 4.4.0 (2024-04-24) for our analyses. This paper introduces **panelPomp** version `1.7.0.0`, and requires **pomp** version 6.2.1.0 or higher.

6 Acknowledgements and funding

This work was supported by National Science Foundation grants DMS-1761603 and DMS-1646108; National Institutes of Health grants 1-U54-GM111274, 1-U01-GM110712, and 1-R01-AI143852; and by MCIN/AEI/10.13039/501100011033 grants PID2020-116242RB-I00 and PID2023-152348NB-I00.

This research was additionally supported in part through computational resources and services provided by Advanced Research Computing at the University of Michigan, Ann Arbor.

References

M. S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp. A tutorial on particle filters for online nonlinear, non-Gaussian Bayesian tracking. *IEEE Transactions on Signal Processing*, 50:174–188, 2002. doi: 10.1109/78.978374. [p187]

K. Asfaw, J. Park, A. A. King, and E. L. Ionides. spatpomp: An R package for spatiotemporal partially observed Markov process models. *Journal of Open Source Software*, 9(104):7008, 2024. doi: 10.21105/joss.07008. URL https://doi.org/10.21105/joss.07008. [p181]

M. Auger-Méthé, K. Newman, D. Cole, F. Empacher, R. Gryba, A. A. King, V. Leos-Barajas, J. Mills Flemming, A. Nielsen, G. Petris, et al. A guide to state–space modeling of ecological time series. *Ecological Monographs*, 91(4):e01470, 2021. doi: https://doi.org/10.1002/ecm.1470. [p182]

C. Bretó. Modeling and inference for infectious disease dynamics: A likelihood-based approach. *Statistical Science*, 33(1):57–69, 2018. doi: 10.1214/17-STS636. [p188]

C. Bretó, D. He, E. L. Ionides, and A. A. King. Time series analysis via mechanistic models. *Annals of Applied Statistics*, 3:319–348, 2009. doi: 10.1214/08-AOAS201. [p180, 187, 195]

C. Bretó, E. L. Ionides, and A. A. King. Panel data analysis via mechanistic models. *Journal of the American Statistical Association*, 115(531):1178–1188, 2020. doi: 10.1080/01621459.2019.1604367. [p180, 182, 187, 188, 191, 194]

C. Breto, J. Wheeler, A. A. King, and E. L. Ionides. A tutorial on panel data analysis using partially observed Markov processes via the R package panelpomp. *arXiv*, 2409.03876, 2024. doi: https://doi.org/10.48550/arXiv.2409.03876. [p196]

Y. Croissant and G. Millo. Panel data econometrics in R: The plm package. *Journal of Statistical Software*, 27(2):1–43, 2008. doi: 10.18637/jss.v027.i02. [p180]

J. E. Domeyer, J. D. Lee, H. Toyoda, B. Mehler, and B. Reimer. Driver-pedestrian perceptual models demonstrate coupling: Implications for vehicle automation. *IEEE Transactions on Human-Machine Systems*, 52(4):557–566, 2022. doi: 10.1109/THMS.2022.3158201. [p180]

A. Doucet, N. De Freitas, N. J. Gordon, et al. *Sequential Monte Carlo methods in practice*, volume 1. Springer, 2001. doi: 10.1007/978-1-4757-3437-9. [p195]

S. R. Eddy. What is a hidden Markov model? *Nature Biotechnology*, 22(10):1315–1316, 2004. doi: 10.1038/nbt1004-1315. [p195]

R. Glennie, T. Adam, V. Leos-Barajas, T. Michelot, T. Photopoulou, and B. T. McClintock. Hidden Markov models: Pitfalls and opportunities in ecology. *Methods in Ecology and Evolution*, 14(1):43–56, 2023. doi: 10.1111/2041-210X.13801. [p195]

U. Halekoh, S. Højsgaard, and J. Yan. The R package geepack for generalized estimating equations. *Journal of Statistical Software*, 15(2):1–11, 2006. doi: 10.18637/jss.v015.i02. [p180]

D. He, E. L. Ionides, and A. A. King. 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, 2010. doi: 10.1098/rsif.2009.0151. [p184, 187]

E. L. Ionides, C. Bretó, and A. A. King. Inference for nonlinear dynamical systems. *Proceedings of the National Academy of Sciences of the USA*, 103:18438–18443, 2006. doi: 10.1073/pnas.0603181103. [p191]

E. L. Ionides, D. Nguyen, Y. Atchadé, S. Stoev, and A. A. King. Inference for dynamic and latent variable models via iterated, perturbed Bayes maps. *Proceedings of the National Academy of Sciences of the USA*, 112(3):719–724, 2015. doi: 10.1073/pnas.1410597112. [p188]

E. L. Ionides, C. Bretó, J. Park, R. A. Smith, and A. A. King. Monte carlo profile confidence intervals for dynamic systems. *Journal of the Royal Society Interface*, 14(132):1–10, 2017. doi: 10.1098/rsif.2017.0126. [p192, 193]

A. A. King, D. Nguyen, and E. L. Ionides. Statistical inference for partially observed Markov processes via the R package pomp. *Journal of Statistical Software*, 69(12):1–43, 2016. doi: 10.18637/jss.v069.i12. URL https://www.jstatsoft.org/index.php/jss/article/view/v069i12. [p181, 182, 191, 195]

A. Lindén and J. Knape. Estimating environmental effects on population dynamics: Consequences of observation error. *Oikos*, 118(5):675–680, 2009. doi: 10.1111/j.1600-0706.2008.17250.x. [p182]

J. A. Marino, S. D. Peacor, D. B. Bunnell, H. A. Vanderploeg, S. A. Pothoven, A. K. Elgin, J. R. Bence, J. Jiao, and E. L. Ionides. Evaluating consumptive and nonconsumptive predator effects on prey density using field time series data. *Ecology*, 100:e02583, 2019. doi: 10.1002/ecy.2583. [p180]

P. Marjoram, J. Molitor, V. Plagnol, and S. Tavare. Markov chain Monte Carlo without likelihoods. *Proceedings of the National Academy of Sciences*, 100(26):15324–15328, 2003. doi: 10.1073/pnas.0306899100. [p187]

B. T. McClintock, R. Langrock, O. Gimenez, E. Cam, D. L. Borchers, R. Glennie, and T. A. Patterson. Uncovering ecological state dynamics with hidden Markov models. *Ecology Letters*, 23(12):1878–1903, 2020. doi: https://doi.org/10.1111/ele.13610. [p195]

Microsoft and S. Weston. *doParallel: Foreach parallel adaptor for the 'parallel' package*, 2022a. URL https://CRAN.R-project.org/package=doParallel. R package version 1.0.17. [p187]

Microsoft and S. Weston. *foreach: Provides foreach looping construct*, 2022b. URL https://CRAN.R-project.org/package=foreach. R package version 1.5.2. [p187]

K. Newman, R. King, V. Elvira, P. de Valpine, R. S. McCrea, and B. J. T. Morgan. State-space models for ecological time-series data: Practical model-fitting. *Methods in Ecology and Evolution*, 14(1):26–42, 2023. doi: 10.1111/2041-210X.13833. [p195]

N. Ning, E. L. Ionides, and Y. Ritov. Scalable monte carlo inference and rescaled local asymptotic normality. *Bernoulli*, 27:2532–2555, 2021. doi: 10.3150/20-BEJ1321. [p194]

S. L. Ranjeva, E. B. Baskerville, V. Dukic, L. L. Villa, E. Lazcano-Ponce, A. R. Giuliano, G. Dwyer, and S. Cobey. Recurring infection with ecologically distinct HPV types can explain high prevalence and diversity. *Proceedings of the National Academy of Sciences of the USA*, 114(51):13573–13578, 2017. doi: 10.1073/pnas.1714712114. [p180]

S. L. Ranjeva, R. Subramanian, V. J. Fang, G. M. Leung, D. K. M. Ip, R. A. P. M. Perera, J. S. M. Peiris, B. J. Cowling, and S. Cobey. Age-specific differences in the dynamics of protective immunity to influenza. *Nature Communications*, 10(1):1660, 2019. doi: 10.1038/s41467-019-09652-6. [p180]

E. Romero-Severson, E. Volz, J. Koopman, T. Leitner, and E. L. Ionides. Dynamic variation in sexual contact rates in a cohort of HIV-negative gay men. *American Journal of Epidemiology*, 182:255–262, 2015. doi: 10.1093/aje/kwv044. [p183, 186]

M. J. Simpson and O. J. Maclaren. Profile-wise analysis: A profile likelihood-based workflow for identifiability analysis, estimation, and prediction with mechanistic mathematical models. *PLoS Computational Biology*, 19(9):e1011515, 2023. doi: 10.1371/journal.pcbi.1011515. [p192]

S. A. Sisson, Y. Fan, and M. M. Tanaka. Sequential Monte Carlo without likelihoods. *Proceedings of the National Academy of Sciences*, 104(6):1760–1765, 2007. doi: 10.1073/pnas.0607208104. [p187]

J. W. Smith, R. Q. Thomas, and L. R. Johnson. Parameterizing lognormal state space models using moment matching. *Environmental and Ecological Statistics*, 30(3):385–419, 2023. doi: 10.1007/s10651-023-00570-x. [p182]

E. Vittinghoff, J. Douglas, F. Judon, D. McKiman, K. MacQueen, and S. P. Buchbinder. Per-contact risk of human immunodeficiency virus transmission between male sexual partners. *American Journal of Epidemiology*, 150(3):306–311, 1999. ISSN 0002-9262. doi: 10.1093/oxfordjournals.aje.a010003. [p183]

H. Wickham. *ggplot2: Elegant graphics for data analysis*. Springer-Verlag New York, 2016. ISBN 978-3-319-24277-4. URL https://ggplot2.tidyverse.org. [p194]

H. Wickham. *Advanced R*. Chapman and Hall/CRC, 2019. ISBN 9781351201315. doi: 10.1201/9781351201315. [p182]

C. P. Winsor. The Gompertz curve as a growth curve. *Proceedings of the National Academy of Sciences of the USA*, 18:1–8, 1932. doi: 10.1073/pnas.18.1.1. [p182]

Carles Bretó
Universitat de València
Department of Economic Analysis
Valencia, Spain
ORCiD: 0000-0003-4695-4902
carles.breto@uv.es

Jesse Wheeler
University of Michigan
Department of Statistics
Ann Arbor, Michigan
ORCiD: 0000-0003-3941-3884
jeswheel@umich.edu

Aaron A. King
University of Michigan and Santa Fe Institute
Department of Ecology and Evolutionary Biology

Ann Arbor, Michigan
ORCiD: 0000-0001-6159-3207
kingaa@umich.edu

Edward L. Ionides
University of Michigan
Department of Statistics
Ann Arbor, Michigan
ORCiD: 0000-0002-4190-0174
ionides@umich.edu