Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *doi: 10.18637/jss.v000.i00*

pypomp: Inference for partially observed Markov process models in Python with JAX DRAFT IN PROGRESS

Aaron J. Abkemeier*
University of Michigan

Jun Chen*
Duke University

Kevin Tan
University of Pennsylvania

Jesse Wheeler
Idaho State University

Bo Yang
Columbia University

Kunyang He
University of Michigan

Jonathan Terhorst
University of Michigan

Aaron A. King

Edward L. Ionides
University of Michigan

***These authors contributed equally**

Abstract

—!!!—an abstract is required—!!!—

Keywords: —!!!—at least one keyword is required—!!!—.

1. Introduction

[Topic] Partially Observable Markov Process (POMP) models, also known as state-space models or hidden Markov models, provide a flexible and mechanistic framework for modeling time-series dynamic systems, particularly suited for scenarios where latent states are only partially observable. Characterized by transition densities and measurement densities of Markov processes, this framework bridges complex underlying dynamics with limited information in real-world data. Consequently, POMP models find extensive application in epidemiology (Mietchen, Clancey, McMichael, and Lofgren 2024; Fox, Lachmann, Tec, Pasco, Woody, Du, Wang, Ingle, Javan, Dahan, Gaither, Escott, Adler, Johnston, Scott, and Meyers 2022; Wen, Yin, Li, Peng, and He 2024), ecology (Auger-Méthé, Newman, Cole, Empacher, Gryba, King, Leos-Barajas, Mills Flemming, Nielsen, Petris, and Thomas 2021; Marino, Peacor, Bunnell, Vanderploeg, Pothoven, Elgin, Bence, Jiao, and Ionides 2019; Blackwood, Streicker, Altizer, and Rohani 2013), finance (Bretó 2014), and other domains.

[Existing package discussion] The rich POMP package ecosystem built in `R` has provided a solid, standardized, and extensible framework for modeling time series data using nonlinear, stochastic partially observed mechanism dynamic models. The `R` **`pomp`** package has become a well-established tool for fitting POMP models using a general and abstract representation, that supports multiple inference techniques. Its extension packages **`panelPomp`**, **`spatpomp`**, and **`phyloPomp`** further enhance its capabilities for panel, spatio-temporal and phylodynamic data analysis respectively

[computational challenges + potential limitations] While conceptually powerful, statistical inference for POMP models using the above `R` packages poses substantial computational challenges. From a methodological perspective, likelihood-based inference for POMP models typically relies on perturbations within iterated filtering (adding ref related to iterative filtering) algorithms. While many of them are stable and effective in locating a neighborhood containing the likelihood maximum, they exhibit numerical inefficiency for obtaining a precise identification of the maximum value. Particularly, when the latent states are high-dimensional or when repeated model evaluations are required, the fitting process could be computationally prohibitive by the constraints.

On the other hand, these POMP models have demonstrated strong potential to be sped up considerably. Many of the processes are embarassingly parallel, such as simulating the state process for each of thousands of particles, running the particle filter multiple times for the same parameter set, and running iterated filtering from multiple starting parameter sets, especially when estimating a profile likelihood to construct a confidence interval as per (Ionides, Breto, Park, Smith, and King 2017). Graphics Processing Units (GPUs) are well-suited for such operations. However, the existing family of POMP packages (**pomp**, **panelPomp**, and **spatPomp** (King, Nguyen, and Ionides 2016, Bretó, Wheeler, King, and Ionides (2025),Asfaw, Park, King, and Ionides (2024))) only runs on CPUs.

[introducing AD/GPU/JAX + pypomp] As the demand grows for scalable and parallelizable inference algorithms, there is a increasing need for a accelerated POMP modeling framework. Automatic differentiation (AD) is a technique that enables efficient and accurate computation of numerical differentiation by systematically applying the chain rule to fundamental operations within computer programs. While several general-purpose AD libraries are available, we directly integrate AD technique into the inference of generic POMP models, particularly the particle filter. This leads to a novel class of algorithms, which are termed automatic differentiation particle filters (ADPF) for POMP modls. The ADPF and differentiable iterated filtering interfaces enable the gradient-based optimization, effectively resolving the previous numerous inefficiencies in the traditional algorithms. Our approach maintains the plug-and-play property (Ionides, Breto, and King 2006), allowing users to specify dynamic models solely through simulators that generate latent state trajectories between arbitrary time points. Furthermore, these methods are implemented in JAX(Bradbury, Frostig, Hawkins, Johnson, Leary, Dougal Maclaurin, Necula, Paszke, VanderPlas, Skye Wanderman-Milne, and Zhang 2018), a high-performance numerical computing library that supports hardware acceleration (GPU) and vectorization. JAX's just-in-time (JIT) compilation further accelerates inference. With the combination of ADPF methods, JAX implementation, and GPU hardware supports, instead of merely a port from the R package `pomp`, `pypomp`(Abkemeier, Chen, Ionides, Wheeler, and Tan 2024) establishes a modern platform for POMP modeling.

[structure] The remainder of this paper is organized as follows. Section 2 discusses the Motivation for `pypomp` design using specific examples. Section 3 demonstrates the mathematical notation for POMP models and their related implementation in `pypomp`. Section 4 introduces the embedded methodologies. Section 5 presents data analysis workflows and benchmarking results. Section 6 concludes with a discussion of future directions.

[NOTE key points for introduction section: add discussion of panelpomp]

Motivation for pypomp

NOTE: This section is for some extra detailed numeric cost estimates and dataset descriptions to illustrate motivation based on Aaron's draft. It is a bit redundant now.

2. Real-world computational bottleneck

Computational speed is a major bottleneck in the practical application of iterated filtering methods to POMP models. In Korevaar, Metcalf, and Grenfell (2020) 's dataset, fitting and evaluating likelihoods of POMP models for 180 units required 8 days on 36 CPU cores (two 3.0 GHz Intel Xeon Gold 6154 CPUs). Scaling this up to the full dataset of 1422 units would require almost eight times as much effort, equivalent to running 36 cores for two months or 288 cores for 8 days. This is not only time consuming, but also incurs substantial computational costs, highlighting the urgent need for more efficient inference software for large-scale POMP analyses Importantly, this cost only accounts for one round of iterated filtering. In practice, to further refine the likelihood estimates, multple rounds are required, which would increase the computational burden significantly. This motivates the development of accelerated, scalable tools to make large-scale POMP inference feasible.

3. Opportunities for speeding up the POMP models

Many of the processes involved in fitting POMP models are embarrassingly parallel. Examples include simulating the state process for each of thousands of particles, running the particle filter repeatedly under the same parameter set, and executing iterated filtering from multiple starting parameter sets. Such parallelism is especially advantageous when estimating a profile likelihood to construct confidence intervals (Ionides *et al.* 2017). Harnessing parallel computing resources can therefore dramatically reduce computation time and make large-scale inference feasible.

Graphics Processing Units (GPUs) are well-suited for embarrassingly parallel operations, but the existing family of POMP packages (**pomp**, **panelPomp**, and **spatPomp** (King *et al.* 2016, Bretó *et al.* (2025),Asfaw *et al.* (2024))) are limited to CPU computation. None provide support for GPU acceleration or automatic differentiation. These two technologies are key to enabling scalable and efficient inference for modern POMP applications.

4. Our solution: pypomp

To address this computational bottleneck, we are creating `pypomp`(Abkemeier *et al.* 2024), a python implementation of the `R` package `pomp`. It draws inspiration from `pomp`, but further implements new methods incoporating automatic differentiation techniques by forking the source code used in Tan (Tan, Hooker, and Ionides 2024), as well as leverages JAX's just-in-time(JIT) compilation and GPU core parallelization (Bradbury *et al.* 2018), allowing practitionersto run filtering methods significantly faster and cheaper. For example, in an SPX comparison model, we show that, compared to `pomp` with 36 CPU cores, `pypomp` can run at least 7 times faster and can finish the job at 5% of the price using 1 GPU and 1 CPU core (5120 CUDA cores on a NVIDIA Tesla V100 and one core from a 2.4 GHz Intel Xeon Gold 6148 CPU).

In addition, `pypomp` is gradually including functionality from `panelPomp` and `spatPomp`, offering a unified Python interface for entire POMP methodologies across multiple `R` packages. It also takes advantage of JAX's implementation of automatic differentiation (AD), which can be used in conjunction with the differentiable measurement off-parameter with discount factor α (MOP-α) particle filter to improve local optimization of the likelihood surface (Tan *et al.* 2024).

5. Summary of key features

Table 1 summarizes the main differences between `pypomp` and `pomp`, highlighting the new capabilities of `pypomp`.

Feature	pypomp	pomp
Backend and Acceleration	JAX (GPU/CPU, JIT,`vmap`, `jax.grad`, `jax.Hessian`)	R and C Snippets (CPU only)
Automatic Differentiation and gradient-based inference	Yes (gradient/Hessian via AD supported)	No
Particle Filtering Methods	Yes (PF, MOP-α, IF2, IFAD)	Yes (PF, IF2, pMCMC, etc.)
Plug-and-Play Property	Yes	Yes
Object Design	In-place updates on current objects, stored in the object attribute `results_history`	Returns new objects

Table 1: Feature comparison between `pypomp` and `pomp` in R ecosystem.

POMP Models in pypomp

This section introduces the structure of POMP models and its implementation in `pypomp`, including both mathematical setup and the package implementation.

6. Model setup

A **partially observed Markov process (POMP)** model has two main components: (i) a latent Markov process that evolves over time and (ii) an observation process that links the latent states. Together, these jointly specify the mechanistic model for the observed time series, providing a framework for modeling dynamic systems where measurements are noisy. Formally, suppose $t_1 < t_2 < ... < t_N$ be a collection of times at which measurements are available, and let t_0 be some time prior to t_1 at which the model is initialized. Let $\{Y_t\}_{t=1}^N$ denote the observations at time $t_1, \dots, t_N$, and $\{X_t\}_{t=1}^N$ denote the postulated latent (unobserved) Markov process at the corresponding time. A POMP model is specified by three building components:

1. initial density: $f_{X_0}(x_0; \theta)$ describes the initial distribution of latent state X_0;
2. transition density: $f_{X_t|X_{t-1}}(x_t \mid x_{t-1}; \theta)$ characterizes the latent Markov process evolution;
3. measurement density: $f_{Y_t|X_t}(y_t \mid x_t; \theta)$ links the observations and the latent states.

The joint density of $(X_{0:N}, Y_{1:N})$ can be expressed as the product of the initial distribution, the transition densities, and the measurement densities:

The joint density of latent states ($X_{0:N}$ and observation $Y_{1:N}$) can be expressed as the product of initial density, , transition density and the measurement density:

$$f_{X_{0:N},Y_{1:N}}(x_{0:N}, y_{1:N}; \theta) = f_{X_0}(x_0; \theta) \prod_{t=1}^{N} f_{X_t|X_{t-1}}(x_t \mid x_{t-1}; \theta) \prod_{t=1}^{N} f_{Y_t|X_t}(y_t \mid x_t; \theta)$$

The marginal likelihood of the observations is $\mathcal{L}(\theta) = f_{Y_{1:N}}(y_{1:N}; \theta) = \int f_{X_{0:N},Y_{1:N}}(x_{0:N}, y_{1:N}; \theta)\, dx_{0:N}$. In practice, this integral is intractable for most nonlinear or non-Gaussian POMP models, motivating the use of simulation-based inference methods such as particle filtering.

In our software, these model components are specified by user-provided functions (`rinit`, `rproc`, `dmeas`), and the package provides various implementations of likelihood evaluation and parameter inference.

7. Implementations of POMP models in pypomp

7.1. Object-oriented interface

A POMP model in `pypomp` is represented as an object of class `Pomp`, which encapsulates the model components: the initial state distribution, process model, and measurement model.

This object-oriented interface allows users to specify by passing components to the constructor, including observations, model parameters, model mechanics such as simulators and the measurement density, covariates, and times. After the components are passed into the constructor, the constructor automatically generates additional internal elements, such as extended observations and covariates required for interpolation

Table 2 summarizes the main arguments to the Pomp constructor and their correspondence to mathematical objects

Constructor	Argument	Type	Description / Mathematical representation
Pomp	`rinit`	`RInit`	simulate initial states $X_0 \sim f_{X_0}(x_0;\theta)$
	`rproc`	`RProc`	simulate state transitions $X_n \sim f_{X_n\mid X_{n-1}}(x_n \mid x_{n-1};\theta)$
	`rmeas`	`RMeas`	simulate observations $Y_n \sim f_{Y_n\mid X_n}(y_n \mid x_n;\theta)$
	`dmeas`	`DMeas`	evaluate measurement density $f_{Y_n\mid X_n}(y_n \mid x_n;\theta)$
	`ys`	`pandas.DataFrame`	observations $y^*_{1:N}$ with times $t_{1:N}$
	`covars`	`pandas.DataFrame`	covariates $z^*_{1:N}$ with times $s_{1:N}$
	`theta`	`list` or `dict`	parameters θ
RInit	`t0`	`float`	initial time point t_0 for simulation
RProc	`step_type`	`str`	method of process evolution: `"fixedstep"` or `"euler"`
	`nstep`	`int`	number of steps if `step_type="fixedstep"`
	`dt`	`float`	time step if `step_type="euler"`
	`accumvars`	`tuple`	indices of state variables to be accumulated
RMeas	`ydim`	`int`	observation dimension $\dim(Y)$

Table 2: Main arguments to the `Pomp` class and related constructor objects.

We demonstrate here how to create a `Pomp` object. Specifically, we show how to create the linear Gaussian model included in the package as `LG()`. We begin by importing necessary packages and defining helper functions for handling the parameters. Because `pypomp` will run our defined model components within JAX JIT-compiled code, it is necessary to write the components to be JAX-compliant. Naturally, the JAX package has many useful functions for this purpose. We also generate a pseudorandom number generation (PRNG) key to be used with JAX random number generators. All stochastic simulations in `pypomp` are controlled via JAX PRNG keys, ensuring full reproducibility when using the same seed.

```
import pypomp as pp
import pandas as pd
import jax
import jax.numpy as jnp
```

```
from functools import partial

def get_thetas(theta):
    theta = jnp.asarray(theta)
    A = theta[0:4].reshape((2, 2))
    C = theta[4:8].reshape((2, 2))
    Q = theta[8:12].reshape((2, 2))
    R = theta[12:16].reshape((2, 2))
    return A, C, Q, R


def transform_thetas(A, C, Q, R):
    return jnp.concatenate([A.ravel(), C.ravel(), Q.ravel(), R.ravel()])

# create PRNG key correctly
key = jax.random.PRNGKey(1)
```

7.2. Model Components

We refer to model components describing initialization, transfer, or measurement processes as model mechanisms, including `rinit`, `rproc`, `dmeas`, and `rmeas`. Users must define these processes as Python functions. Specifically, we require users to provide function code to the object constructor, which verifies that all necessary function arguments are included and in the correct order. This requirement stems from `pypomp`'s internal mechanism: it vectorizes component functions using `jax.vmap()` to efficiently run thousands of particles. Since `jax.vmap()` maps functions to input arrays by position rather than keyword, users must strictly adhere to parameter order. While all expected parameters must be included, the function does not need to utilize all of them.

Illustrated in Table 2,`pypomp` also includes object constructors for components describing the model mechanics: `RInit`, `RProc`, `DMeas`, and `RMeas`. Some constructors also require additional arguments, such as `t0` for `RInit`. Notably, `RProc` takes `step_type`, `dt`, and `nstep` arguments. `step_type` determines how `RProc` should be run at intermediate steps between two observation times. If we want to model the state process as evolving in continuous time, setting `step_type="euler"`uses an Euler approximation, running `rproc` at intermediate steps based on the time step size, `dt`. The number of steps taken is given by the number of times `dt` divides the difference between two observation times, rounded up, and is consequently dynamic. Otherwise, if we instead want a fixed number of steps for each observation time interval, we can use `step_type="fixedstep"`, in which case `rproc` will run at `nstep` intermediate steps equally spaced between two observation times, starting from the first observation time. Consequently, setting `step_type="fixedstep"` and `nstep=1` only runs `rproc` at the observation times. Here is an example of defining the object constructors for components under the linear gaussian model.In practice, at least one of `dmeas` or `rmeas` must be provided, while the construction of `RInit` and `RProc` are always required.

```
import pypomp as pp
```

```
@partial(pp.RInit, t0=0.0)
def rinit(theta_, key, covars=None, t0=None):
    A, C, Q, R = get_thetas(theta_)
    return jax.random.multivariate_normal(key=key, mean=jnp.array([0.0, 0.0]), cov=Q)

@partial(pp.RProc, step_type="fixedstep", nstep=1)
def rproc(X_, theta_, key, covars=None, t=None, dt=None):
    A, C, Q, R = get_thetas(theta_)
    return jax.random.multivariate_normal(key=key, mean=A @ X_, cov=Q)

@pp.DMeas
def dmeas(Y_, X_, theta_, covars=None, t=None):
    A, C, Q, R = get_thetas(theta_)
    # return logpdf of Y given X (mean = C @ X_, cov = R)
    return jax.scipy.stats.multivariate_normal.logpdf(Y_, mean=C @ X_, cov=R)

@partial(pp.RMeas, ydim=2)
def rmeas(X_, theta_, key, covars=None, t=None):
    A, C, Q, R = get_thetas(theta_)
    return jax.random.multivariate_normal(key=key, mean=C @ X_, cov=R)
```

7.3. Parameters

The `Pomp` constructor also requires model parameters. These can be provided either as a dictionary or as a list of dictionaries. Each item in a dictionary should include the parameter name as the key and the parameter value as the dictionary value. If the parameter sets are provided as a list of dictionaries, methods such as `pfilter()` run on each set of parameters. Here, we use `Pomp.sample_params()` to sample sets of parameters from uniform distributions with bounds passed as a dictionary of length-2 tuples. `Pomp.sample_params()` returns a ready-to-use list of dictionaries with the sampled parameters. Internally, parameters, even are multi-dimensional, are stored as flat dictionaries to facilitate JAX transformations and compilation.

```
theta = {
    "A11": jnp.cos(0.2), "A12": -jnp.sin(0.2),
    "A21": jnp.sin(0.2), "A22": jnp.cos(0.2),
    "C11": 1.0, "C12": 0.0, "C21": 0.0, "C22": 1.0,
    "Q11": 0.01, "Q12": 1e-6, "Q21": 1e-6, "Q22": 0.01,
    "R11": 0.1, "R12": 0.01, "R21": 0.01, "R22": 0.1,
}
param_bounds = {k: (v * 0.9, v * 1.1) for k, v in theta.items()}
n = 5
key = jax.random.PRNGKey(1)
key, subkey = jax.random.split(key)
theta_list = pp.Pomp.sample_params(param_bounds, n, subkey)
```

7.4. Covariates

Scientifically, POMP models often involve external time-varying inputs, referred to as covariates, which can influence either the latent process or the measurement model. Examples include seasonality, interventions, or environmental drivers in ecological applications. In `pypomp`, covariates are supplied as a `pandas.DataFrame` indexed by time. The time at which the covariates were observed should be specified in the `ctime` argument. Importantly, the covariate time points may differ from the observation times, necessitating interpolation. Given the observation times, covariate times, and the step type specified in `RProc`, the model automatically aligns and interpolates observations and covariates to ensure consistency with the simulation of the latent and observation processes. The linear gaussian model doesn't involve any covariates, and an example using covariates is given in the Data Analysis Section.

7.5. POMP Object Construction

We do not have real data in this `LG` example, so we generate our own. To make this example cleaner, we here use the function `LG()` to construct the completed linear Gaussian model object and then generate the data using `simulate()`. Observation times are provided to the `Pomp` constructor via the `pandas.DataFrame` row index. If covariates were provided, the times at which the covariates were observed would also be provided by the `pandas.DataFrame` row index.

```
import jax, jax.numpy as jnp
import pandas as pd
import pypomp as pp

T = 100
# ensure `key` exists; if not, uncomment the next line
# key = jax.random.PRNGKey(1)

key, subkey = jax.random.split(key)
sims = pp.LG(T=T).simulate(key=subkey)

ys = pd.DataFrame(
    sims[0]["Y_sims"].squeeze(),
    index=range(1, T + 1),
    columns=["Y1", "Y2"],
)

LG_obj = pp.Pomp(
    rinit=rinit,
    rproc=rproc,
    dmeas=dmeas,
    rmeas=rmeas,
    ys=ys,
    theta=theta_list,
    covars=None,
```

```
)
```

```
print("LG_obj created; ys.shape =", ys.shape)
```

```
LG_obj created; ys.shape = (100, 2)
```

Each argument to `Pomp` is accessible from the object as an attribute.

```
print(LG_obj.rinit) # access POMP model components
print(LG_obj.rproc)
print(LG_obj.dmeas)
print(LG_obj.rmeas)
print(LG_obj.theta)   # access parameters
print(LG_obj.ys.head())  # access observations
```

```
<pypomp.model_struct.RInit object at 0x13f9682c0>
<pypomp.model_struct.RProc object at 0x157a57fb0>
<pypomp.model_struct.DMeas object at 0x157a55880>
<pypomp.model_struct.RMeas object at 0x1579d3890>
[{'A11': 0.963512122631073, 'A12': -0.17880238592624664, 'A21': 0.20370502769947052, 'A22'
         Y1        Y2
1 -0.087193  0.639745
2 -0.270096  0.156701
3  0.078600 -0.056542
4 -0.014927 -0.499934
5  0.291701  0.426928
```

7.6. Premade models:

Beyond the linear gaussian model, `pypomp` includes several ready-to-use model constructors that serve both as examples and as tested templates for custom model development:

1. `LG()` — a simple linear-Gaussian model with 2-dimensional latent and observed states; useful to validate API usage and diagnostics.
2. `spx()` — the S&P500 log-return model from Sun et al. (Sun 2024).
3. `dacca()` — the cholera transmission model from King et al. (King, Ionides, Pascual, and Bouma 2008).
4. `UKMeasles.Pomp()` — the measles district model from He et al. (He, Ionides, and King 2010), wired to the Korevaar et al. dataset (Korevaar *et al.* 2020). Panel and spatial variants (PanelPOMP/SpatPOMP style) are planned.

These examples show correct component wiring (`rinit`, `rproc`, `dmeas`, `rmeas`), recommended `step_type`/`dt` usage, and typical diagnostics. If a user model errors or runs slowly, compare its components to the matching premade model to find mistakes and performance opportunities. Meanwhile, these premade models can also replicate well-know case studies in the R pomp ecosystem, allowing direct comparison and validation.

7.7. JAX Numerical Backend and Interface Design

A key design choice `pypomp` is it relys heavily on the JAX numerical backend. Unlike the R package `pomp`, where users typically provide POMP model components in C Snippets for acceleration, `pypomp` requires model components to be written as JAX-compatible Python functions. These functions are then compiled and vectorized by JAX tools such as `jit` and `vmap`. This design leads to several important interface features:

- **Strict argument requirements for compilation and vectorization:** JAX's `jit` compiler transforms the user-supplied component functions (`rinit`, `rproc`, `dmeas`, `rmeas`) into efficient machine code, while `vmap` efficiently run them over thousands of particles via vectorization of arguments. To ensure the compatibility with JAX's compilation and vectorization system, each component function must follow the expected input types and order, otherwise compilation would fail.

- **PRNG random key policy**: To ensure the reproducibility of randomness in POMP models under `pypomp`, the public API accept an optional `jax.random.PRNGkay`, which is explicityly passed through constructors and methods. Keys are internally split when it is needed. Unlike the R setting, where randomness can be controlled globally or by seed chunks, in JAX, random keys only be explicitly passed through funcitons

- **Consistent shapes and sizes handling**: model parameters, even multidimensional, are stored as flattened dictionaries. Consequently, JAX can uniformly process parameters, thereby maintaining consistency in particle propagation.

Later section will demonstrate how the JAX-based design supports further inference methods.

[Question: more introductions on JAX?]

7.8. Panel POMP class

POMP Methods in pypomp

In this section, we describe the core inference methods currently implemented in `pypomp`, including :

- `Paricle Filter` (Sequential Monte Carlo, written in `pfilter()`): A standard sequential Monte Carlo algorithm for likelihood evaluation and state estimation, forming the basis for most inference methods in POMP models.
- `Measurement-off-policy` Particle Filter (MOP(α), written in `mop()`): A recently proposed SMC method (Tan *et al.* 2024) that evaluates the likelihood at one parameter value while obtaining resampling decisions from another, adjusting via discounted off-parameter measurement weights.
- `Iterated Filtering` (IF2, `mif()`): A classical IF2 algorithm (Ionides, Nguyen, Atchadé, Stoev, and King 2015) for likelihood-based parameter inference that maximizes the likelihood via particle filtering.
- `Iterated Filtering with Automatic Differentiation` (`train()`): A recently proposed AD-based algorithm (Tan *et al.* 2024) that incorporates MOP(α), the differentiable particle filter, to enable efficient gradient-based parameter inference for maximum likelihood estimation.

A key feature of the above POMP inference methods lies in the **plug-and-play property**

(Ionides *et al.* 2006), meaning that inference algorithms can be implemented without requiring explicit evaluation of the transition density of the latent process. Instead, it suffices for the user to provide a simulator of the latent process (`rproc`), initial state distribution (`rinit`), and observation measurement model (`dmeas`, `rmeas`). This property enables POMP methods to be widely applied to complex mechanistic models where transition densities are intractable.

In `pypomp`, the plug-and-play design is fully preserved: users only need to provide component functions compliant with JAX requirements, which can be directly plugged in inference methods like`pfilter()`, `mop()`, `mif()`, and `train()`. The package combines the generality of plug-and-play modeling with the efficiency of JAX compilation and vectorization.

Unlike the R family of POMP packages, some `Pomp` class methods including `pfilter()`, `mif()` and `train()` yield results by modifying the object in place instead of returning new objects. All of results are stored a list under `LG_obj.results_history`, which is an attibute under `Pomp` class object `LG_obj`. Each element in the list correponds to one method call. Each element includes results such as the log-likelihood and parameter estimates when applicable as well as the inputs used for the function call, so it is easy to keep track of how the results were calculated. If multiple parameter sets are supplied in a list as an argument, the method evaluates at each set and the results for each are stored.

```
LG_obj.pfilter(J = 100,
               reps = 5,
               key = subkey)
LG_obj.mif(sigmas = 0.02,
           sigmas_init = 0.1,
           M = 2,
           a = 0.5,
           J = 100,
           key = subkey)

print(LG_obj.results_history)
```

```
[{'method': 'pfilter', 'logLiks': <xarray.DataArray (theta: 5, replicate: 5)> Size: 100B
Array([[ -96.25428 ,  -94.48256 ,  -98.0167  ,  -96.89331 , -101.386505],
       [ -88.01761 ,  -89.463234,  -86.35478 ,  -88.52014 ,  -86.4893  ],
       [ -85.702576,  -87.60687 ,  -86.09397 ,  -86.86352 ,  -87.17774 ],
       [ -99.95984 ,  -98.52388 ,  -98.849236, -100.11646 , -105.14761 ],
       [ -86.5121  ,  -83.150444,  -85.58832 ,  -83.39534 ,  -85.051895]],      dtype=floa
Dimensions without coordinates: theta, replicate, 'theta': [{'A11': 0.963512122631073, 'A1
0         NaN  0.963512 -0.178802  0.203705  1.069749  0.999111  0.000000
1   -0.000000  0.698513 -0.182381  0.123838  0.744760  0.631224  0.124299
2 -121.995293  0.755329 -0.215463  0.001247  0.647478  0.414623 -0.004155

        C21       C22       Q11           Q12       Q21       Q22       R11  \
0  0.000000  1.054883  0.010472  9.363539e-07  0.000001  0.010878  0.102497
1  0.058696  0.766573  0.108757 -1.344804e-01  0.110075  0.104514  0.163150
2  0.277807  0.756554  0.241079 -4.330555e-01  0.187088  0.177275  0.140986

        R12       R21       R22
```

```
0  0.010768  0.009182  0.096302
1 -0.186850  0.177904  0.091796
2 -0.228250  0.192902  0.098360  ,        logLik       A11       A12       A21       A22
0         NaN  0.966490 -0.178802  0.214531  0.982279  0.941170  0.000000
1   -0.000000  0.618959 -0.214863  0.095818  0.897657  0.893280  0.124688
2 -130.906647  0.551545 -0.415750  0.296858  1.093511  0.748129  0.614655

        C21       C22       Q11       Q12       Q21       Q22       R11  \
0  0.000000  1.077328  0.009964  0.000001  0.000001  0.009380  0.109075
1  0.264847  0.825057  0.105006 -0.391712  0.310784  0.124249  0.130566
2  0.225912  0.589995  0.116228 -0.170694  0.207179  0.119729  0.128406

        R12       R21       R22
0  0.009379  0.010691  0.093214
1  0.056420 -0.040017  0.093769
2  0.224661 -0.211174  0.137349  ,        logLik       A11       A12       A21       A22
0         NaN  0.967444 -0.178802  0.185848  0.989572  1.003363  0.000000
1   -0.000000  0.575949 -0.118584  0.208007  0.832811  0.877054  0.285515
2 -143.576355  0.246772 -0.114797  0.448115  0.910574  0.739520 -0.015389

        C21       C22       Q11           Q12       Q21       Q22       R11  \
0  0.000000  1.003444  0.009594  9.344108e-07  0.000001  0.010617  0.100243
1  0.339936  0.869348  0.189332 -2.465630e-01  0.038902  0.241033  0.205867
2  0.225607  0.831877  0.246554 -2.449578e-01  0.164394  0.086019  0.115632

        R12       R21       R22
0  0.010192  0.010793  0.098955
1  0.275044 -0.130057  0.197181
2  0.365310 -0.340070  0.135814  ,        logLik       A11       A12       A21       A22
0         NaN  1.015698 -0.178802  0.192220  1.073113  0.973013  0.000000
1   -0.000000  0.754714 -0.186093  0.206044  0.762122  0.990731 -0.095371
2 -135.249161  0.380205 -0.160751  0.157198  0.753371  0.601139  0.095075

        C21       C22       Q11           Q12       Q21       Q22       R11  \
0  0.000000  0.916460  0.009097  9.765389e-07  0.000001  0.009699  0.101522
1 -0.030712  0.857645  0.165851  2.412775e-01 -0.177483  0.106149  0.287129
2 -0.130519  0.807980  0.139537  4.800732e-01 -0.429105  0.103709  0.106721

        R12       R21       R22
0  0.009060  0.010467  0.095675
1  0.027864 -0.120701  0.110245
2 -0.112830  0.089776  0.078192  ,        logLik       A11       A12       A21       A22
0         NaN  0.977909 -0.178802  0.195180  0.928942  1.055157  0.000000
1   -0.000000  0.822491 -0.259999  0.467303  0.823400  0.682049  0.014131
2 -136.957916  0.958484 -0.221754  0.131998  0.716926  0.722765  0.110022

        C21       C22       Q11       Q12       Q21       Q22       R11  \
```

```
0  0.000000  0.939434  0.010325  0.000001  0.000001  0.010019  0.108067
1 -0.055267  0.735370  0.102880  0.018784  0.042280  0.177021  0.112855
2  0.164151  0.699523  0.068404 -0.077313  0.045861  0.097577  0.113749

        R12       R21       R22
0  0.010244  0.010489  0.096365
1 -0.114201  0.083839  0.067392
2 -0.098525  0.052682  0.090189  ], 'theta': [{'A11': 0.963512122631073, 'A12': -0.1788023
```

8. Particle Filter (pfilter)

NOTE: 1. purpose/role 2. implementation details in pypomp 3. outputs/results 4. remarks/highlights

The particle filter algorithm, referred to Algorithm 1, [Introduction (purpose/role) of pfilter]

Algorithm 1 Sequential Monte Carlo (SMC, or particle filter) in pypomp: `LG_obj.pfilter(J=J, reps=reps, key=key)`, where `LG_obj` is a class `Pomp` object with definitions for `rinit`, `rproc`, `dmeas`, `rmeas`, `ys`, and `theta`

Require: Simulator for $f_{X_n|X_{n-1}}(x_n \mid x_{n-1};\theta)$; Evaluator for $f_{Y_n|X_n}(y_n \mid x_n;\theta)$; Simulator for $f_{X_0}(x_0;\theta)$; Parameter θ; Data $y^*_{1:N}$; Number of particles J.

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

2: **for** $n = 1$ to N **do**

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

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

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

6: Resample indices $k_{1:J}$ with $\Pr[k_j = m] = \tilde{w}(n,m)$.

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

8: Compute conditional log likelihood:

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

9: **end for**

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

10: Complexity: $\mathcal{O}(J)$

In `pypomp`, the `pfilter()` functions is internally run in `pfilter_internal()` but wrapped up into a class method. It returns a `dict` type element updated inside the `LG_obj.result_history` attribute, containing the log-likelihoods, algorithm parameters used, as well as model diagonostic elements (conditional log-likelihood, effective sample size, filtered mean, and prediction mean) at each time included if their respective boolean flags are set to True. For example, suppose we run

```
LG_obj_2 = pp.Pomp(
    rinit=rinit,
    rproc=rproc,
    dmeas=dmeas,
    rmeas=rmeas,
    ys=ys,
    theta=theta_list,
    covars=None,
)

LG_obj_2.pfilter(J = 1000,
                 reps = 10,
                 key = subkey)

LG_obj_2.pfilter(J = 1000,
                 reps = 10,
                 key = subkey,
                 CLL = True,
                 ESS = True,
                 filter_mean = True,
                 prediction_mean = True)
```

where J is the number of particles used and *reps* is the number of particle filtering replicates to run for each para,eter set provided in the `Pomp` object or as an optional argument to `pfilter()`. Because `LG_obj2.result_history` begins as an empty list here when the model is constructed, the results are appended at `LG_obj_2.results_history[0]` and `LG_obj_2.results_history[1]` respectively. Both of these two dictionaries contain with the following items:

- `method`: The method that was run. In this case, `pfilter`.
- `logLiks`: A
- `theta`:
- `J`:
- `thresh`:
- `key`: The PRNG key used

Meanwhile, `LG_obj_2.results_history[1]` also contains the following itesm that are not contained in `LG_obj_2.results_history[0]` :

- `CLL`:
- `ESS`:
- `filter_mean`:
- `predict_mean`:

9. MOP

Algorithm 2 MOP(α): Measurement off-policy sequential Monte Carlo

Initialize filter particles: simulate $X_{0,j}^{F,\theta} \sim f_{X_0}(\cdot;\theta)$ for $j = 1{:}J$.
Initialize relative weights: $w_{0,j}^{F,\theta} = 1$ for $j = 1{:}J$.
for $n = 1$ to N **do**
  Simulate prediction particles: $X_{n,j}^{P,\theta} \sim f_{X_n|X_{n-1}}(\cdot \mid X_{n-1,j}^{F,\theta};\theta)$ for $j = 1{:}J$.
  Prediction weights with discounting: $w_{n,j}^{P,\theta} = \left(w_{n-1,j}^{F,\theta}\right)^{\alpha}$ for $j = 1{:}J$.
  Evaluate measurement density: $g_{n,j}^{\theta} = f_{Y_n|X_n}(y_n^* \mid X_{n,j}^{P,\theta};\theta)$ for $j = 1{:}J$.
  Conditional likelihood:
$$L_n^{\theta,\alpha} = \frac{\sum_{j=1}^{J} g_{n,j}^{\theta} w_{n,j}^{P,\theta}}{\sum_{j=1}^{J} w_{n,j}^{P,\theta}}.$$
  Conditional likelihood under ϕ:
$$L_n^{\phi} = \frac{1}{J}\sum_{m=1}^{J} g_{n,m}^{\phi}.$$
  Normalize weights: $\tilde{g}_{n,j}^{\phi} = \dfrac{g_{n,j}^{\phi}}{L_n^{\phi}}$ for $j = 1{:}J$.
  Resample indices $k_{1:J}$ with $\Pr[k_j = m] = \tilde{g}_{n,m}^{\phi}$.
  Resample particles: $X_{n,j}^{F,\theta} = X_{n,k_j}^{P,\theta}$ for $j = 1{:}J$.
  Filter weights corrected for resampling:
$$w_{n,j}^{FC,\theta} = w_{n,j}^{P,\theta} \times \frac{g_{n,j}^{\theta}}{g_{n,j}^{\phi}} \quad \text{for } j = 1{:}J.$$
  Resample filter weights: $w_{n,j}^{F,\theta} = w_{n,k_j}^{FC,\theta}$ for $j = 1{:}J$.
end for
Likelihood estimate: $L(\theta) = \prod_{n=1}^{N} L_n^{\theta,\alpha}$.

10. Iterated Filtering

Algorithm 3 Iterated Filtering (IF2)

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

for $m = 1$ to M **do**
  Initialize parameters: $\Theta^P_{0,j,i} \sim \mathrm{Normal}\big([\theta_{m-1}]_i,\ (Ca^m\sigma_i)^2\big)$ for $i \in 1{:}p$, $j \in 1{:}J$.
  Initialize states: simulate $X^F_{0,j} \sim f_{X_0}(\cdot\,;\Theta^P_{0,j})$ for $j = 1{:}J$.
  Initialize filter mean: $\bar{\theta}_0 = \theta_{m-1}$.
  Define $[V]_i = (C^2+1)a^{2m}\sigma_i^2$.
  for $n = 1$ to N **do**
    Perturb parameters: $\Theta^P_{n,j,i} \sim \mathrm{Normal}\big([\Theta^F_{n-1,j}]_i,\ (a^m\sigma_i)^2\big)$ for $i \notin I$, $j = 1{:}J$.
    Simulate prediction particles: $X^P_{n,j} \sim f_{X_n|X_{n-1}}(\cdot \mid X^F_{n-1,j};\Theta^P_{n,j})$ for $j = 1{:}J$.
    Evaluate weights: $w(n,j) = f_{Y_n|X_n}(y^*_n \mid X^P_{n,j};\Theta^P_{n,j})$ for $j = 1{:}J$.
    Normalize weights: $\tilde{w}(n,j) = \dfrac{w(n,j)}{\sum_{u=1}^J w(n,u)}$.
    Resample indices $k_{1:J}$ with $\Pr[k_u = j] = \tilde{w}(n,j)$.
    Resample particles: $X^F_{n,j} = X^P_{n,k_j}$ and $\Theta^F_{n,j} = \Theta^P_{n,k_j}$ for $j = 1{:}J$.
    Filter mean: $[\bar{\theta}_n]_i = \sum_{j=1}^J \tilde{w}(n,j)[\Theta^P_{n,j}]_i$ for $i \notin I$.
    Prediction variance: $[V_{n+1}]_i = (a^m\sigma_i)^2 + \sum_{j=1}^J \tilde{w}(n,j)\big([\Theta^P_{n,j}]_i - [\bar{\theta}_n]_i\big)^2$ for $i \notin I$.
  end for
  Update non-IVPs: $[\theta_m]_i = [\theta_{m-1}]_i + [V]_i \sum_{n=1}^N \big([\bar{\theta}_n]_i - [\theta_{m-1}]_i\big)$ for $i \notin I$.
  Update IVPs: $[\theta_m]_i = \frac{1}{J}\sum_{j=1}^J [\Theta^F_{L,j}]_i$ for $i \in I$.
end for

Ensure: Monte Carlo maximum likelihood estimate θ_M.

11. Iterated Firltering with Automatic Differentiation

Algorithm 4 IFAD: Iterated Filtering with Automatic Differentiation

Require: Number of particles J, timesteps N, IF2 cooling schedule η_m, MOP-α discounting parameter α, initial parameter θ_0, iteration index $m = 0$.

1: Run IF2 until initial "convergence" under cooling schedule η_m, or for a fixed number of iterations, to obtain $\{\Theta_j, j = 1, \ldots, J\}$.

2: Set $\theta_m := \frac{1}{J}\sum_{j=1}^{J} \Theta_j$.

3: **while** procedure not converged **do**

4: Run Algorithm 2 (MOP-α filter) to obtain $\hat{\ell}(\theta_m)$.

5: Obtain gradient and Hessian:

$$g(\theta_m) = \nabla_{\theta_m}(-\hat{\ell}(\theta_m)), \quad H(\theta_m) \quad \text{s.t. } \lambda_{\min}(H(\theta_m)) > c.$$

6: Update parameter:

$$\theta_{m+1} := \theta_m - \eta_m H(\theta_m)^{-1} g(\theta_m).$$

7: Set $m := m + 1$.

8: **end while**

Ensure: Return $\hat{\theta} := \theta_m$.

Data Analysis with pypomp

This section demonstrates: - Log-likelihood profiling - GPU benchmarking - Conditional log-likelihood residuals

Discussion

References

Abkemeier A, Chen J, Ionides E, Wheeler J, Tan K (2024). "Pypomp."

Asfaw K, Park J, King AA, Ionides EL (2024). "spatPomp: An R Package for Spatiotemporal Partially Observed Markov Process Models." *Journal of Open Source Software*, **9**(104), 7008. ISSN 2475-9066. doi:10.21105/joss.07008.

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

Blackwood JC, Streicker DG, Altizer S, Rohani P (2013). "Resolving the roles of immunity, pathogenesis, and immigration for rabies persistence in vampire bats." *Proceedings of the National Academy of Sciences of the United States of America*, **110**(51), 20837–20842. doi:10.1073/pnas.1308817110. URL https://doi.org/10.1073/pnas.1308817110.

Bradbury J, Frostig R, Hawkins P, Johnson MJ, Leary C, Dougal Maclaurin, Necula G, Paszke A, VanderPlas a, Skye Wanderman-Milne, Zhang Q (2018). "JAX: Composable Transformations of Python+NumPy Programs."

Bretó C (2014). "On idiosyncratic stochasticity of financial leverage effects." *Statistics & Probability Letters*, **91**, 20–26. doi:10.1016/j.spl.2014.04.003. URL https://doi.org/10.1016/j.spl.2014.04.003.

Bretó C, Wheeler J, King AA, Ionides EL (2025). "panelPomp: Analysis of Panel Data via Partially Observed Markov Processes in R." doi:10.48550/arXiv.2410.07934. 2410.07934.

Fox SJ, Lachmann M, Tec M, Pasco R, Woody S, Du Z, Wang X, Ingle TA, Javan E, Dahan M, Gaither K, Escott ME, Adler SI, Johnston SC, Scott JG, Meyers LA (2022). "Real-time pandemic surveillance using hospital admissions and mobility data." *Proceedings of the National Academy of Sciences*, **119**(7), e2111870119. doi:10.1073/pnas.2111870119. URL https://doi.org/10.1073/pnas.2111870119.

He D, Ionides EL, King AA (2010). "Plug-and-Play Inference for Disease Dynamics: Measles in Large and Small Populations as a Case Study." *Journal of The Royal Society Interface*, **7**(43), 271–283. ISSN 1742-5689, 1742-5662. doi:10.1098/rsif.2009.0151.

Ionides EL, Breto C, King AA (2006). "Inference for Nonlinear Dynamical Systems." *Proceedings of the National Academy of Sciences*, **103**(49), 18438–18443. ISSN 0027-8424, 1091-6490. doi:10.1073/pnas.0603181103.

Ionides EL, Breto C, Park J, Smith RA, King AA (2017). "Monte Carlo Profile Confidence Intervals for Dynamic Systems." *Journal of The Royal Society Interface*, **14**(132), 20170126. ISSN 1742-5689, 1742-5662. doi:10.1098/rsif.2017.0126.

Ionides EL, Nguyen D, Atchadé Y, Stoev S, King AA (2015). "Inference for Dynamic and Latent Variable Models via Iterated, Perturbed Bayes Maps." *Proceedings of the National Academy of Sciences*, **112**(3), 719–724. ISSN 0027-8424, 1091-6490. doi:10.1073/pnas.1410597112.

King AA, Ionides EL, Pascual M, Bouma MJ (2008). "Inapparent Infections and Cholera Dynamics." *Nature*, **454**(7206), 877–880. ISSN 1476-4687. doi:10.1038/nature07084.

King AA, Nguyen D, Ionides EL (2016). "Statistical Inference for Partially Observed Markov Processes via the *R* Package **Pomp**." *Journal of Statistical Software*, **69**(12). ISSN 1548-7660. doi:10.18637/jss.v069.i12.

Korevaar H, Metcalf CJ, Grenfell BT (2020). "Structure, Space and Size: Competing Drivers of Variation in Urban and Rural Measles Transmission." *Journal of The Royal Society Interface*, **17**(168), 20200010. doi:10.1098/rsif.2020.0010.

Marino JAJ, Peacor SD, Bunnell DB, Vanderploeg HA, Pothoven SA, Elgin AK, Bence JR, Jiao J, Ionides EL (2019). "Evaluating consumptive and nonconsumptive predator effects on prey density using field time-series data." *Ecology*, **100**(3), e02583. doi:10.1002/ecy.2583. URL https://doi.org/10.1002/ecy.2583.

Mietchen MS, Clancey E, McMichael C, Lofgren ET (2024). "Estimating SARS-CoV-2 transmission parameters between coinciding outbreaks in a university population and the surrounding community." doi:10.1101/2024.01.10.24301116. URL https://doi.org/10.1101/2024.01.10.24301116.

Sun W (2024). "Model Based Inference of Stochastic Volatility via Iterated Filtering."

Tan K, Hooker G, Ionides EL (2024). "Accelerated Inference for Partially Observed Markov Processes Using Automatic Differentiation." doi:10.48550/arXiv.2407.03085. 2407.03085.

Wen L, Yin Y, Li Q, Peng Z, He D (2024). "Modeling the co-circulation of influenza and COVID-19 in Hong Kong, China." *Advances in Continuous and Discrete Models*, **2024**(1), 1–9. doi:10.1186/s13662-024-03830-7. Article 32, URL https://doi.org/10.1186/s13662-024-03830-7.

Affiliation:

Aaron J. Abkemeier*
Department of Statistics
E-mail: aaronabk@umich.edu

Jun Chen*
Department of Statistics
E-mail: chenjunc@umich.edu

Kevin Tan
Department of Statistics
E-mail: kevtan@wharton.upenn.edu

Jesse Wheeler
Department of Mathematics
E-mail: jessewheeler@isu.edu

Bo Yang
Department of Marketing
E-mail: ybb@umich.edu

Kunyang He
Department of Statistics
E-mail: kunyanghe@umich.edu

Jonathan Terhorst
Department of Statistics
E-mail: jonth@umich.edu

Aaron A. King

Edward L. Ionides
Department of Statistics
E-mail: ionides@umich.edu

*These authors contributed equally

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
MMMMMM YYYY, Volume VV, Issue II *Submitted:* yyyy-mm-dd
doi:10.18637/jss.v000.i00 *Accepted:* yyyy-mm-dd