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

Aaron J. Abkemeier* Jun Chen* Kevin Tan Jesse Wheeler
Bo Yang Kunyang He Jonathan Terhorst Aaron A. King
Edward L. Ionides *These authors contributed equally

Methods for fitting and evaluating partially observed Markov process (POMP) mechanistic models are useful in a variety of fields such as epidemiology, ecology, and finance, but they are limited by their computational demands. In response, we introduce **pypomp**, a high-performance Python package for statistical inference using POMP models. Built to handle complex stochastic dynamic systems, it implements plug-and-play algorithms for likelihood-based inference including sequential Monte Carlo and iterated filtering. By utilizing parallel processing on graphical processing unit (GPU) hardware, **pypomp** is able to run these algorithms faster and more efficiently than its predecessor, the R package **pomp**. Models that took weeks to fit and evaluate in **pomp** now take only days in **pypomp**, facilitating the use of previously impractical models. The package offers a comprehensive interface that streamlines the entire pipeline of model construction, fitting, and analysis, allowing practitioners to develop and test large, complex models with minimal implementation overhead.

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 et al. 2024; Fox et al. 2022; Wen et al. 2024), ecology (Auger-Méthé et al. 2021; Marino et al. 2019; Blackwood et al. 2013), finance (Bretó 2014), and other domains.

[Existing package discussion]

The POMP package ecosystem 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.

As it turns out, POMP methods can be sped up considerably. Many of the processes involved in POMP methods 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 (E. L. Ionides et al. 2017). Graphics Processing Units (GPUs) are well-suited for such operations. However, the existing family of POMP packages (**pomp**, **panelPomp**, and **spatPomp** Asfaw et al. (2024)) only runs on CPUs.

[introducing AD/JIT/GPU hardware + pypomp]

As the demand grows for scalable and parallelizable inference algorithms, there is an increasing need for an accelerated framework for POMP modeling framework. The Python package pypomp (Abkemeier et al. 2024) addresses this by integrating **automatic differentiation (AD)**, **just-in-time (JIT) compilation**, and **hardware acceleration** through JAX (Bradbury et al. 2018), a high-performance numerical computing library that supports hardware acceleration and vectorization. AD techniques enable efficient and accurate derivative computation by systematically applying the chain rule to fundamental operations. Recent works have extended AD for gradient estimation using particle filtering, yielding a class of methods termed automatic differentiation particle filters (ADPF). Building on these advancements, pypomp provides AD-enabled gradient estimation for inference for POMP models within a plug-and-play framework (E. L. Ionides, Breto, and King 2006), where users specify dynamic models solely through simulators of latent state trajectories, rather than evaluating the transition density of the latent Markov process. JAX's JIT compilation further accelerates repeated inference and evaluation by converting Python functions into machine code at runtime. Finally, JAX's GPU support enables efficient scaling of inference methods. Together, these features make `pypomp` more than a port of the `R` package `pomp`. Instead, it establishes an independent, modernized Python platform for building, simulating and fitting fast and flexible POMP models, aiming to serve two user groups: scientists who need fast, plug-and-play likelihood-based inference for nonlinear dynamic systems, as well as method developers who want a clean, differentiable platform for experimenting with new particle-filter and Bayesian algorithms.

[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.

[add discussion of panelpomp into introduction]

Motivation for pypomp

[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.]

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.

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 (E. L. 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** 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.

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, Hooker, and Ionides 2024).

Summary of key features

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

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

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

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.

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 we observe the process at discrete time $n = 1, \dots, N$, with initial time 0. Data $Y_{1:N}$ are modeled as noisy, incomplete, and indirect observations of a latent (unobserved) Markov procss $\{X_{0:N}\}$ at the corresponding time. A POMP model is defined by three building components, each corresponding to user-supplied Python functions in `pypomp`:

1. initial density: $f_{X_0}(x_0; \theta)$, implemented via `rinit`, generates draws of the initial states X_0.
2. transition density: $f_{X_n|X_{n-1}}(x_n \mid x_{n-1}; \theta)$, implemented via `rproc`, propagates the evolution of latent states.
3. measurement density: $f_{Y_n|X_n}(y_n \mid x_n; \theta)$, implemented via `dmeas`, evaluates or simulates observations conditional on the latent states.

Here, we hold an conditional independence assumption that, given X_n, the observation Y_n is independent of all other variables. Then, 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:

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

The marginal likelihood of the observations is $\mathcal{L}(\theta) = f_{Y_{1:N}}(y_{1:N}; \theta) = \int f_{X_{0:N},Y_{1:N}}(x_{0:N}, y_{1:N}; \theta)\, dx_{0:N}$. Such POMP models can include all the features desired by (Bjørnstad and Grenfell 2001). 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.

Implementations of POMP models in pypomp

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.

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

Constructor	Attributes	Type	Description / Mathematical representation
Pomp	`rinit`	`RInit`	simulate initial states $X_0 \sim f_{X_0}(x_0; \theta)$
	`rproc`	`RProc`	simulate state transitions $X_n \sim f_{X_n\|X_{n-1}}(x_n \mid x_{n-1}; \theta)$
	`rmeas`	`RMeas`	simulate observations $Y_n \sim f_{Y_n\|X_n}(y_n \mid x_n; \theta)$
	`dmeas`	`DMeas`	evaluate measurement density $f_{Y_n\|X_n}(y_n \mid x_n; \theta)$
	`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`	`dict`	parameters θ
	`par_trans`	`Partrans`	parameter transformation object
	`results_hisotry`	`list`	History of the results for POMP methods run on the object

Constructor	Attributes	Type	Description / Mathematical representation
	`fresh_key`	`jax.Array`	A fresh, unused key is stored in the attribute after running a method on the object with a key argument

where `RInit`, `RProc`, `DMeas`, and `RMeas` are sub-constructors that specify the stochastic mechanisms of the model—the initial state simulator, state process transition, measurement density, and measurement simulator, respectively. The user provides each component as a plain Python function and `pypomp` then wraps into a JAX-compatible class instance. These wrapped components are passed as arguments when initializing the main `Pomp` object. The `ParTrans` sub-constructor defines parameter transformations between the specific estimation space and the natural parameter space.

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.

```python
import pypomp as pp
import pandas as pd
import jax
import jax.numpy as jnp
from functools import partial


def get_thetas(theta):
    A = jnp.array(
      [theta["A11"], theta["A12"], theta["A21"], theta["A22"]]
    ).reshape(2, 2)
    C = jnp.array(
      [theta["C11"], theta["C12"], theta["C21"], theta["C22"]]
    ).reshape(2, 2)
    Q = jnp.array(
      [theta["Q11"], theta["Q12"], theta["Q21"], theta["Q22"]]
    ).reshape(2, 2)
    R = jnp.array(
      [theta["R11"], theta["R12"], theta["R21"], theta["R22"]]
    ).reshape(2, 2)
    return A, C, Q, R


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


rw_sd = pp.RWSigma(
  sigmas={
    "A11": 0.02, "A12": 0.02,
    "A21": 0.02, "A22": 0.02,
    "C11": 0.02, "C12": 0.02,
    "C21": 0.02, "C22": 0.02,
    "Q11": 0.02, "Q12": 0.02,
    "Q21": 0.02, "Q22": 0.02,
    "R11": 0.02, "R12": 0.02,
    "R21": 0.02, "R22": 0.02,
  },
  init_names=[],
)

key = jax.random.key(1)
```

Model Components

We refer to model components describing initialization, transfer, or measurement processes as the model mechanics, including `rinit`, `rproc`, `dmeas`, and `rmeas`. Users must define these mechanics 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 arguments must be included, the function does not need to utilize all of them. We include here an example defining the model mechanics for the linear gaussian model. In practice, at least one of `dmeas` or `rmeas` must be provided, while definitions for `rinit` and `rproc` are always required.

```
def rinit(theta_, key, covars=None, t0=None):
    """Initial state process simulator for the linear Gaussian model"""
    A, C, Q, R = get_thetas(theta_)
    result = jax.random.multivariate_normal(key=key, mean=jnp.array([0, 0]), cov=Q)
    return {"X1": result[0], "X2": result[1]}


def rproc(X_, theta_, key, covars=None, t=None, dt=None):
    """Process simulator for the linear Gaussian model"""
    A, C, Q, R = get_thetas(theta_)
    X_array = jnp.array([X_["X1"], X_["X2"]])
    result = jax.random.multivariate_normal(key=key, mean=A @ X_array, cov=Q)
    return {"X1": result[0], "X2": result[1]}


def dmeas(Y_, X_, theta_, covars=None, t=None):
    """Measurement model distribution for the linear Gaussian model"""
    A, C, Q, R = get_thetas(theta_)
    X_array = jnp.array([X_["X1"], X_["X2"]])
    Y_array = jnp.array([Y_["Y1"], Y_["Y2"]])
    return jax.scipy.stats.multivariate_normal.logpdf(Y_array, X_array, R)

def rmeas(X_, theta_, key, covars=None, t=None):
    """Measurement simulator for the linear Gaussian model"""
    A, C, Q, R = get_thetas(theta_)
    X_array = jnp.array([X_["X1"], X_["X2"]])
    return jax.random.multivariate_normal(key=key, mean=C @ X_array, cov=R)
```

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.

```
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.key(1)
key, subkey = jax.random.split(key)
theta_list = pp.Pomp.sample_params(param_bounds, n, subkey)
```

Covariates

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 covariate time points may differ from the observation times, so the `Pomp` object automatically interpolates covariate values at the observation times. The linear gaussian model does not involve any covariates, and an example using covariates is given in the Data Analysis Section.

POMP Object Construction

We do not have real data in this `LG` example, so we generate our own. To make this example cleaner, we 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.

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

key, subkey = jax.random.split(key)
_, ys = pp.LG(T=T).simulate(key=subkey)
ys = ys.rename(columns={"obs_0": "Y1", "obs_1": "Y2"})
ys = ys[["time", "Y1", "Y2"]]
ys.set_index("time", inplace=True)

LG_obj = pp.Pomp(
    rinit=rinit,
    rproc=rproc,
    dmeas=dmeas,
    rmeas=rmeas,
    t0=0.0,
    nstep=1,
    ydim=2,
    ys=ys,
    theta=theta_list,
    statenames=["X1", "X2"],
)

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 0x327e3b0b0>
<pypomp.model_struct.RProc object at 0x327bacfb0>
<pypomp.model_struct.DMeas object at 0x327bce960>
<pypomp.model_struct.RMeas object at 0x3272ac9e0>
PompParameters(n_replicates=5, n_params=16)
            Y1        Y2
time
1.0  -0.115930  0.724815
2.0  -0.353621  0.230780
3.0   0.079373 -0.154176
4.0   0.148403 -0.297087
5.0   0.655207  0.529257
```

Premade models:

Beyond the linear gaussian model, pypomp includes several ready-to-use model constructors that serve as examples for what users can do:

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 et al. 2008).
4. `UKMeasles.Pomp()` — the measles district model from He et al. (He, Ionides, and King 2010), extended to the Korevaar et al. dataset (Korevaar, Metcalf, and Grenfell 2020). Panel and spatial variants (PanelPOMP/SpatPOMP style) are planned.

These examples demonstrate correct model specification. If a user's model throws an error or runs slowly, they may compare its components to a premade model to make improvements.

Panel POMP class

POMP Methods in pypomp

In this section, we describe the core elementary and 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 (optional) state estimation, forming the basis for most inference methods in POMP models.
- `Differentiated Measurement-off-policy` Particle Filter (DMOP(α), written in `mop()`): A recently proposed SMC method (Tan, Hooker, and Ionides 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 (Edward L. Ionides et al. 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, Hooker, and Ionides 2024) that leverages DMOP(α) perform maximum likelihood estimation with gradient descent methods.

It also contains other class funcions for better model diagnostics after running the above elmentary and inference mthods:

- `simulate()` performs simulations of the POMP model, i.e. it samples from joint distributions of latent states and observations.
- `traces()` returns the full trace of log-likelihoods and parameters from the entire result history.
- `results(idx)` returns the results of the method run at the given index.
- `time()` summarizes the execution times of methods run

- `prune()` replaces self.theta with a list of top n thetas based on the most recent available log-likelihood estimates.
- `plot_traces()` generates the plots containing parameter and log-likelihood traces from the entire result history.
- `print_summary()` prints a summary of the POMP object.

A key feature of the above POMP inference methods lies in the **plug-and-play property** (E. L. Ionides, Breto, and King 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 the latent process simulator (`rproc`), the initial state simulator (`rinit`), and the observation density (`dmeas`). 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 the 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. Meanwhile, each following method in `pypomp` that takes a key as an argument stores an unused child key under `LG_obj.fresh_key` that later method calls can use by default when a key argument is not given. This design prevents repeated use of the same key across a sequence of method calls and ensures proper randomness.

```
LG_obj.pfilter(J=100, reps=5, key=subkey)
LG_obj.mif(rw_sd=rw_sd, M=2, a=0.5, J=100, key=subkey)

print(LG_obj.results_history.traces())
```

```
   replicate  iteration   method       logLik       A11       A12       A21  \
0          0          0  pfilter   -99.785549  0.963512 -0.178802  0.203705
1          0          0      mif          NaN  0.963512 -0.178802  0.203705
2          0          1      mif  -146.906250  0.572848 -0.199666  0.192313
3          0          2      mif  -142.255066  0.610309 -0.159505  0.255666
4          1          0  pfilter   -90.157137  0.966490 -0.178802  0.214532
5          1          0      mif          NaN  0.966490 -0.178802  0.214531
```

```
6           1          1      mif -135.346619  0.863753 -0.176641  0.332267
7           1          2      mif -141.794006  0.688705 -0.040529  0.255167
8           2          0  pfilter  -90.953700  0.967444 -0.178802  0.185848
9           2          0      mif         NaN  0.967444 -0.178802  0.185848
10          2          1      mif -134.741959  0.816369 -0.055617  0.242442
11          2          2      mif -134.361496  0.669066 -0.267495  0.387243
12          3          0  pfilter -104.294093  1.015698 -0.178802  0.192220
13          3          0      mif         NaN  1.015698 -0.178802  0.192220
14          3          1      mif -149.708771  1.041893 -0.030668  0.088754
15          3          2      mif -155.627655  0.903259 -0.012985  0.219679
16          4          0  pfilter  -88.790386  0.977908 -0.178802  0.195180
17          4          0      mif         NaN  0.977909 -0.178802  0.195180
18          4          1      mif -141.853638  1.065276 -0.229773  0.152513
19          4          2      mif -136.760620  0.841766 -0.171315  0.332422

         A22       C11       C12       C21       C22       Q11           Q12  \
0   1.069749  0.999112  0.000000  0.000000  1.054883  0.010472  9.363539e-07
1   1.069749  0.999111  0.000000  0.000000  1.054883  0.010472  9.363539e-07
2   0.886104  0.902826  0.031704 -0.335078  0.976379  0.269876  2.102747e-01
3   1.008311  0.885925 -0.210804 -0.548692  0.966376  0.147663  9.209296e-02
4   0.982279  0.941170  0.000000  0.000000  1.077328  0.009964  1.079856e-06
5   0.982279  0.941170  0.000000  0.000000  1.077328  0.009964  1.079856e-06
6   0.840223  0.484592  0.188556 -0.165521  0.911229  0.128907 -1.154891e-01
7   0.663210  0.372712  0.445420 -0.128935  0.461554  0.155705 -4.652517e-02
8   0.989572  1.003363  0.000000  0.000000  1.003444  0.009594  9.344107e-07
9   0.989572  1.003363  0.000000  0.000000  1.003444  0.009594  9.344108e-07
10  0.874546  0.593635 -0.429084  0.259972  0.827221  0.115433 -9.864584e-02
11  0.915659  0.459554 -0.345782 -0.086212  1.027752  0.149203 -1.380723e-01
12  1.073113  0.973013  0.000000  0.000000  0.916460  0.009097  9.765388e-07
13  1.073113  0.973013  0.000000  0.000000  0.916460  0.009097  9.765389e-07
14  1.174014  0.607510 -0.044455 -0.164548  0.789968  0.250420  1.784301e-01
15  0.831207  0.677350 -0.022509 -0.268272  0.663113  0.198001  2.027629e-01
16  0.928942  1.055156  0.000000  0.000000  0.939434  0.010325  1.084500e-06
17  0.928942  1.055157  0.000000  0.000000  0.939434  0.010325  1.084500e-06
18  0.807170  0.922964  0.257845 -0.087839  1.086084  0.208441 -4.319310e-04
19  0.645019  1.313488 -0.042610 -0.147641  1.075309  0.107748  1.334490e-01

         Q21       Q22       R11       R12       R21       R22
0   0.000001  0.010878  0.102497  0.010768  0.009182  0.096302
1   0.000001  0.010878  0.102497  0.010768  0.009182  0.096302
2  -0.111698  0.086543  0.261754  0.024031 -0.139018  0.146297
3  -0.050992  0.117316  0.231509 -0.081919  0.060413  0.142251
4   0.000001  0.009380  0.109075  0.009379  0.010691  0.093214
```

```
5   0.000001  0.009380  0.109075  0.009379  0.010691  0.093214
6   0.090095  0.118782  0.176969 -0.049828 -0.056426  0.120035
7   0.098540  0.196946  0.223285  0.040767 -0.183336  0.166994
8   0.000001  0.010617  0.100243  0.010192  0.010793  0.098955
9   0.000001  0.010617  0.100243  0.010192  0.010793  0.098955
10  0.115726  0.146333  0.124413  0.038974 -0.094679  0.079910
11  0.154053  0.135989  0.143612  0.081676 -0.160226  0.150676
12  0.000001  0.009699  0.101522  0.009060  0.010467  0.095675
13  0.000001  0.009699  0.101522  0.009060  0.010467  0.095675
14 -0.108098  0.119053  0.190852 -0.230511  0.106878  0.195054
15 -0.235946  0.062398  0.172400 -0.437604  0.376313  0.118888
16  0.000001  0.010019  0.108067  0.010244  0.010489  0.096365
17  0.000001  0.010019  0.108067  0.010244  0.010489  0.096365
18 -0.001807  0.136363  0.168800 -0.118241  0.113117  0.184493
19 -0.185023  0.199876  0.102027 -0.040129  0.103583  0.148904
```

Particle Filter (pfilter)

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

The particle filter algorithm (sequential Monte Carlo, SMC) (Arulampalam et al. (2002), King, Nguyen, and Ionides (2016)) is the core classical likelihood evaluation tool in POMP model inference. It uses Monte Carlo techniques to sequentially estimate the integrals in the prediction and filtering recursions, and produce an unbiased Monte Carlo estimate of the likelihood of the observed data under a parameter set. The likelihood estimate is essential for other parameter inference algorithms such as iterative filtering. In `pypomp`, the basic particle filter is implemented under `pfilter()`. The algorithm is summarized as Algorithm 1,

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

```
Require Simulator for $\text{process}_n(\cdot|x_n;\theta)$; Evaluator for $f_{Y_n|X_n}(y_n \mid x_n;\theta)$; Simulator for $f_{X_0}(x_0;\theta)$; Parameter $\theta$; Data $y^*_{1:N}$; Number of particles $J$.
Initialize particles: $X^{F,\theta}_{0,j} \sim f_{X_0}(\cdot;\theta)$ for $j = 1{:}J$.
For $n = 1, ..., N$:
    Simulate prediction particles: $X^{P,\theta}_{n,j} \sim \text{process}_n(\cdot|x_n;\theta)$ for $j = 1{:}J$.
    Evaluate weights and measurement density: $w(n,j) = f_{Y_n|X_n}(y^*_n \mid X^{P,\theta}_{n,j};\theta)$ for $j = 1{:}J$.
    Normalize weights $\tilde{w}(n,j) = \dfrac{w(n,j)}{\sum_{m=1}^{J} w(n,m)}$ for $j = 1 : J$.
    Select resample indices $k_{1:J}$ with $P(k_j = m) = \tilde{w}(n,m)$.
    Obtain resampled particles $X^{F,\theta}_{n,j} = X^{P,\theta}_{n,k_j}$ for $j = 1{:}J$.
    Compute conditional log likelihood:
        $$\hat{\ell}_{n|1:n-1} = \log\left(\frac{1}{J}\sum_{m=1}^{J} w(n,m)\right).$$
Return Log likelihood estimate $\hat{\ell}(\theta) = \sum_{n=1}^{N}\hat{\ell}_{n|1:n-1}$; filter samples $X^{F,\theta}_{n,1:J}$ for $n = 1{:}N$.
Complexity: $\mathcal{O}(J)$
```

In `pypomp`, the `pfilter()` functions is internally run in a JIT-compiled kernel `_vmapped_pfilter_internal2`, which applies `jax.vmap` over replicates and sharded parameters on `theta`. This enables the parallel execution on CPU/GPU and ensures the computational efficiency. In implementation, the weights and likelihood are stored in a log scale, preventing underflow. There are optional arguments (`CLL`, `ESS`, `filter_mean`, `prediction_mean`) control wheter additional diagnostic arrays are allocated and updated. The algorithm works for flexible and extendable models through the user-specified `Pomp` class components (`rinit`, `rproc`, `dmeas`, `reams`), number of particles `J` as well as other parameters. Each call generates a `PompPFilterResult` object, which is then appended to `object.results_history` attribute, containing the execution time, evaluated log-likelihoods, algorithm parameters used, as well as model diagnostic elements at each time included if their respective boolean flags are set to True. Here `PompPFilterResults` class is derived from `PompBaseResult` and provides standardized storage and methods for `pfilter()` results. This return design supports multi-stage inference workflow and ensures reproducibility. For example, suppose we run

```
LG_obj_2 = pp.Pomp(
    rinit=rinit,
    rproc=rproc,
    dmeas=dmeas,
    rmeas=rmeas,
    t0=0.0,
    nstep=1,
    ydim=2,
    ys=ys,
    theta=theta_list,
    statenames=["X1", "X2"],
)

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 parameter set provided in the `Pomp` object or as an optional argument to `pfilter()`. Because `LG_obj2.results_history` begins as an empty list here when the model is constructed, the `PompPFilterResult` object results are appended at `LG_obj_2.results_history[0]` and `LG_obj_2.results_history[1]` respectively. Both of these two objects contain the following attributes which are not None:

- `method`: The method that was run. In this case, `"pfilter"`.
- `execution time`: The execution time if `track_time = True`
- `key`: The RNG key used.
- `theta`: The list of parameter sets used.
- `logLiks`: An `xarray.DataArray` of particle filter log-likelihood estimates. Evaluated log-likelihoods returned from the internal kernel are reshaped into `(n_theta, reps)` based on the number of parameter sets and replicates.
- `J`: The number of particles used.
- `reps`: The number of replicates.
- `thresh`: Threshold value for resampling used, which is 0 by default.

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

- `CLL`: An `xarray.DataArray` containing the conditional loglikelihood over observed time steps (`theta`, `replicate`, `time`).
- `ESS`: An `xarray.DataArray` containing the effective sample size over observed time steps (`theta`, `replicate`, `time`).
- `filter_mean`: An `xarray.DataArray` containing the filtered mean over observed time steps (`theta`, `replicate`, `time`, `state`).
- `prediction_mean`: An `xarray.DataArray` containing the prediction mean over observed time steps. (`theta`, `replicate`, `time`, `state`).

The result `PompPFilterResult` also provides the following functions for further analysis: - `to_dataframe()` provides the summary data frame of per-parameter-set log-likelihood estimates via log-mean-exp trick with Monte Carlo standard error. - `traces()` returns traces data frame with one row per replicate. - `print_summary` prints brief readable summary for the results with top 5 evaluated log-likelihood results.

A key design is that all runs across multiple parameter sets and replications are stored in one `results_history` list under the corresponding `Pomp` object. This avoids object duplication, and increases reproducibility by facilitating pickling `Pomp` objects obtained from many different search strategies, loading them into a new Python session, and then comparing the results to see how they vary with different algorithmic parameters.

Differentiated Measurement Off-Parameter (DMOP) particle filter

([TO BE REWRITTEN, FOLLOWING ISSUE 2])

[Motivation for DMOP] Classical particle filters for POMP models produce unbiased Monte Carlo estimates, but are non-differentiable in model parameters θ, due to the discrete random sampling of particles. This prevents the wide use of AD and gradient-based optimization in POMP dynamical systems. The differentiable measurement off-parameter particle filter (DMOP-α, Tan, Hooker, and Ionides (2024)) provides a solution: it treats a *differentiable* likelihood estimator produced by a variant of the standard particle filter as an objective and apply AD to obtain $\nabla_\theta \widehat{\ell}(\theta)$ for likelihood-based optimal parameter search. The derivative estimate from DMOP-α is essential for further stochastic gradient inference methods, such as iterated filtering with automatic differentiation (IFAD), where it employs stochastic gradient descent using DMOP-α. A tunning dicount factor $\alpha \in [0, 1]$ controls the bias-variance trade-off of the gradient estimator: larger α may result in lower bias and higher variance, while smaller α can reduce the variance with the cost of increasing bias. In implementation, DMOP-α does not require heavy diagnostics or multiple replicates, and doesn't require time tracking function. It only needs a stable, differentiable of the (log-) likelihood to feed into `jax.grad` and `jax.value_and_grad`. The pseudocode is shown in 2

Algorithm 2 DMOP-α in `pypomp`: `jax.value_and_grad(LG_obj.mop(J=J, reps=reps, key=key, alpha=alpha))`. `LG_obj` is a `Pomp` object with components `rinit`, `rproc`, `dmeas`, `ys`, `theta`.

```
Require Simulator $\text{process}_n(\cdot|x_n;\theta)$; Evaluator for $f_{Y_n|X_n}(y_n^*|x_n,\theta)$; Simulator for $f_{X_0}(\cdot;\theta)$;
    Evaluation parameter $\theta$; Data $y_{1:N}^*$; Number of particles J; Discount factor $\alpha$.
Initialize particles $X_{0,j}^{F,\theta} \sim f_{X_0}(\cdot\,;\theta)$, weights $w_{0,j}^{F,\theta} = 1$ for $j = 1 : J$.
For $n = 1, ..., N$:
    Simulate prediction particles, $X_{n,j}^{P,\theta} \sim \text{process}_n(\,\cdot\,|X_{n-1,j}^{F,\theta};\theta)$ for $j = 1 : J$.
    Accumulate discounted prediction weights, $w_{n,j}^{P,\theta} = (w_{n-1,j}^{F,\theta})^{\alpha}$.
    Evaluate measurement density, $g_{n,j}^{\theta} = f_{Y_n|X_n}(y_n^*|X_{n,j}^{P,\theta}\,;\theta)$ for $j = 1 : J$.
    Compute conditional likelihood in version B: $L_n^{B,\theta,\alpha} = \sum_{j=1}^{J} g_{n,j}^{\theta}\, w_{n,j}^{P,\theta} \Big/ \sum_{j=1}^{J} w_{n,j}^{P,\theta}$.
    Select resampling indices $k_{1:J}$ with $P(k_j = m) \propto g_{n,m}^{\theta}$.
    Obtain resampled particles $X_{n,j}^{F,\theta} = X_{n,k_j}^{P,\theta}$.
    Calculate corrected weights $w_{n,j}^{F,\theta} = w_{n,k_j}^{P,\theta}\, g_{n,k_j}^{\theta} \Big/ \texttt{stop\_gradient}(g_{n,k_j}^{\theta})$.
    Compute considtional likelihood in version A: $L_n^{A,\theta,\alpha} = L_n^{\phi} \cdot \sum_{j=1}^{J} w_{n,j}^{F,\theta} \Big/ \sum_{j=1}^{J} w_{n,j}^{P,\theta}$.
Forward Pass: AD constructs a computation graph to evaluate the likelihood estimate
    $\hat{L}^A(\theta) = \prod_{n=1}^{N} L_n^{A,\theta,\alpha}$ or $\hat{L}^B(\theta) = \prod_{n=1}^{N} L_n^{B,\theta,\alpha}$.
Backward Pass: AD propagates gradients through the computation graph and evaluates
    gradient estimate $\hat{\mathscr{D}}^A(\theta)$ or $\hat{\mathscr{D}}^B(\theta)$
Return: likelihood estimate $\hat{L}^A(\theta)$ or $\hat{L}^B(\theta)$, and gradient estimate $\hat{\mathscr{D}}^A(\theta)$ or $\hat{\mathscr{D}}^B(\theta)$.
```

The key building block of the DMOP-α is the MOP-α algorithm, a variant of the standard particle filter designed to make the discontinuous Monte Carlo resampling process independent of the target parameter θ. It evaluates the likelihood at the evaluation parameter θ but draws resampling indices using weights computed at another baseline behavior parameter ϕ. This renders the likelihood estimates differentiable w.r.t. θ, further enabling AD.

In DMOP-α, the same idea is implemented equivalently by setting $\phi \equiv$ `stop_gradient`(θ) in JAX. This prevent its input from being differentiated. The filtering process therefore runs $\theta = \phi$, assuming the evaluation parameter and the bahavior parameters have identical values. This avoids the need to explicitly introduce new parameters while preserving the differentiable property. Consequently, the likelihood evaluation of DMOP is identical to that of the standard particle filter, and it does not require implement a fully separate MOP-α procedure. This design simplifies the algorithm and still ensures the valid likelihood and gradient estimator, while maintaining differentiability with respect to θ. That is, DMOP computes the derivative of the likelihood estimate obtained by MOP when $\theta = \phi$. The pseudocode of MOP is shown in 3.

Algorithm 3 MOP(α), Measurement off-policy sequential Monte Carlo

```
Require Simulator for process$_n(\cdot|x_n;\theta)$; Evaluator for $f_{Y_n|X_n}(y_n \mid x_n;\theta)$; Simulator for $f_{X_0}(x_0;\theta)$; Evaluation Parameter $\theta$; Behavior Parameter $\phi$; Data $y^*_{1:N}$; Number of particles $J$.
Initialize filter particles $X^{F,\theta}_{0,j} \sim f_{X_0}(\cdot;\theta)$ and $w^{F,\theta}_{0,j} = 1$ for $j = 1{:}J$.
For $n = 1, ..., N$:
    Simulate prediction particles: $X^{P,\theta}_{n,j} \sim f_{X_n|X_{n-1}}(\cdot \mid X^{F,\theta}_{n-1,j};\theta)$ for $j = 1{:}J$.
    Accumulate discounted Prediction weight: $w^{P,\theta}_{n,j} = \left(w^{F,\theta}_{n-1,j}\right)^{\alpha}$ for $j = 1{:}J$.
    Evaluate measurement density: $g^{\theta}_{n,j} = f_{Y_n|X_n}(y^*_n \mid X^{P,\theta}_{n,j};\theta)$ for $j = 1{:}J$.
    Compute conditional likelihood under $\theta$ in version $B$: $L^{B,\theta,\alpha}_n = \sum_{j=1}^{J} g^{\theta}_{n,j} w^{P,\theta}_{n,j} \Big/ \sum_{j=1}^{J} w^{P,\theta}_{n,j}$.
    Compute conditional likelihood under $\phi$:$L^{\phi}_n = \frac{1}{J}\sum_{m=1}^{J} g^{\phi}_{n,m}$.
    Select resample indices $k_{1:J}$ with $P(k_j = m) \propto \tilde{g}^{\phi}_{n,m}$.
    Obtain resampled particles $X^{F,\theta}_{n,j} = X^{P,\theta}_{n,k_j}$ for $j = 1{:}J$.
    Calculate resampled corrected weights:$w^{F,\theta}_{n,j} = w^{P,\theta}_{n,j} \times \frac{g^{\theta}_{n,j}}{g^{\phi}_{n,j}}$ for $j = 1{:}J$.
    Compute conditional likelihood under $\theta$ in version $A$: $L^{A,\theta,\alpha}_n = L^{\phi}_n \cdot \sum_{j=1}^{J} w^{F,\theta}_{n,j} \Big/ \sum_{j=1}^{J} w^{P,\theta}_{n,j}$.
Return: Likelihood estimate: $\hat{L}(\theta) = \prod_{n=1}^{N} L^{A,\theta,\alpha}_n$ or $\hat{L}(\theta) = \prod_{n=1}^{N} L^{B,\theta,\alpha}_n$.
```

In `pypomp`, DMOP-α is implemented by wrapping the JAX-compiled `mop()` function inside JAX AD. It is a `Pomp` class method calling an internal `_mop_internal()`, which uses static arguments and `jax.lax.fori_loop` for efficiency, and stores weights and likelihoods in log scale to prevent underflow. Systematic sampling is involved for the resampling process. Users can directly apply `jax.grad` and `jax.value_and_grad` to `mop()` to obtain gradients.

In practice, the `mop()` provides an auto-differentiable alternative to `pfilter()`, as a lightweight version. Unlike `pfilter()`, which provides the choice of multiple replicates and diagnostic summaries, `mop()` in `pypomp` focuses purely on producing differentiable likelihood estimates. This makes the algorithm suitable for AD and stochastic gradient descent algorithms in accelerated iterated filtering algorithms by setting the function as input of `jax.value_and_grad()` and `jax.grad()`. Beyond a well-constructed `Pomp` object, the method requires only the number of particles `J`, the evaluation parameter `theta`, a PRNG key, and the cooling factor α. Results are stored in the `results_history` attribute as a list of `jax.Array` objects, one for each parameter set, representing the estimated log-likelihood of the observations. For example, we run

```
LG_obj_3 = pp.Pomp(
    rinit=rinit,
    rproc=rproc,
    dmeas=dmeas,
    rmeas=rmeas,
    t0=0.0,
    nstep=1,
    ydim=2,
    ys=ys,
    theta=theta_list,
    statenames=["X1", "X2"],
)

LG_obj_3.mop(J=1000, key=subkey, alpha=0.97)
```

```
[Array(-99.67439, dtype=float32),
 Array(-89.77343, dtype=float32),
 Array(-91.195496, dtype=float32),
 Array(-103.54038, dtype=float32),
 Array(-88.92766, dtype=float32)]
```

Generally, the `result_history` only updates:

- log-likelihood(s): a list of `jax.Array` containing the estimated log-likelihood(s) of the observed data given the model parameters. Always a list, even if only one theta is provided.

This design highlights `mop()` as a lightweight but essential building block for accelarated gradient-based inference methods within `pypomp`. By combining `jax.grad` and `jax.value_and_grad` around `mop()`, the package effectively implements the DMOP algorithm, enabling stochastic-gradient updates for likelihood-based inference on POMP models.

Iterated Filtering

Iterated filtering (IF2) provides a Monte Carlo approach to estimate parameters by combining particle filtering with sequential random-walk perturbations and a cooling schedule to approximate maximum likelihood estimates when the likelihood is analytically intractable but can be unbiasedly estimated through sequential Monte Carlo methods. Compared with a plain particle filter or MOP method, IF2 not only evaluates the likelihoods but also systematically updates parameters across iterations. IF2 enables systematic exploration of the parameter space in early iterations via perturbation and gradual stabilization around high-likelihood regions when the perturbation variance decreases. The IF2 algorithm contains M iterations, each using J particles. At iteration m, parameters are perturbed by a Gaussian random walk with standard deviations scaled by a cooling factor. At each observation time, the likelihood evaluations and resampling are performed and parameter estimates are obtained. The perturbation magnitudes are progressively reduced across iterations, resulting in convergence to a Monte Carlo approximation of the maximum likelihood estimate, which is an optimal parameter estimate.

The pseudocode for IF2 algorithm is as follows:

Algorithm 4 Iterated Filtering (IF2)

```
Require: Starting parameter $\theta_0$; 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: $[V_{n+1}]_i = (a^m\sigma_i)^2 + \sum_{j=1}^{J} \tilde{w}(n,j)\left([\Theta^P_{n,j}]_i - [\bar{\theta}_n]_i\right)^2$ for $i \notin I$.
    end for
    Update non-IVPs: $[\theta_m]_i = [\theta_{m-1}]_i + [V]_i \sum_{n=1}^{N} \left([\bar{\theta}_n]_i - [\theta_{m-1}]_i\right)$ 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 $\theta_M$.
```

In `pypomp`, the IF2 algorithm is implemented in the method `mif()`. It follows the structure of the IF2 implementation in the R package `pomp`, while introducing JAX-based vectorization and compilation for improved computational performance.

The argument `rw_sd` specifies the random walk standard deviations for parameter perturbations. The perturbation scales are encapsulated in a `RWSigma` object, which is passed to the method through the `rw_sd`. The `RWSigma` object stores a mapping from parameter names to non-negative perturbation values and distinguishes between initial-value parameters (IVPs) and non-initial parameters, allowing different perturbation behavior at the initial time point and at subsequent time steps. The float-type cooling rate `a` specifies the decay of perturbation magnitudes across iterations. Together, `rw_sd` and `a` jointly control the trade-off between global exploration and local convergence of the parameter estimates.

The `mif()` updates the interal state of the `Pomp` object directly. In particular, `LG_obj.theta` is replaced with the parameter estimate from the end of the last iteration. The traces of log-likelihoods and parameter estimates are stored in `LG_obj.results_history` as a `xr.DataArray` object. This array has dimentions `(replicate, iteration, variable)`, where the `variable` dimension includes the log-likelihood estimations and all POMP model parameters. For reproducibility, each call updates and stores a child key into `LG_obj.fresh_key`, allowing subfequent method calls to safely omit an explicit assignment on `key` argument.

Each call generates a `PompMIFResult` object, another dataclass derived from `PompBaseResult`. The object is appended to `object.results_history` attribute, containing the IF2 trace outputs, algorithm settings and execution context:

- `method`: The method that was run. In this case, `"mif"`.
- `execution time`: The execution time if `track_time = True`
- `key`: The RNG key used.
- `theta`: The list of parameter sets used.
- `traces_da`: An `xarray.DataArray` of IF2 trace outputs. The array is indexed by `(replicate, iteration, variable)`, where `variable` includes the estimated loglikelihood and all model parameters.
- `J`: The number of particles used.
- `M`: The number of IF2 iterations.
- `rw_sd`: an `RWSigma` object representing the random walk pertubation specification.
- `a`: Cooling fraction.
- `thresh`: Threshold value for resampling used, which is 0 by default.

Similar to `PompPFilterResult` class, `PompMIFResult` object contains the `to_dataframe()`, `traces()`, and `print_summary()` as helper functions

```
key, subkey = jax.random.split(key)

LG_obj_4 = pp.Pomp(
    rinit=rinit,
    rproc=rproc,
    dmeas=dmeas,
    rmeas=rmeas,
    t0=0.0,
    nstep=1,
    ydim=2,
    ys=ys,
    theta=theta_list,
    statenames=["X1", "X2"],
)

rw_sd = pp.RWSigma(
        sigmas={
            "A11": 0.02,
            "A12": 0.02,
            "A21": 0.02,
            "A22": 0.02,
            "C11": 0.02,
            "C12": 0.02,
            "C21": 0.02,
            "C22": 0.02,
            "Q11": 0.02,
            "Q12": 0.02,
            "Q21": 0.02,
            "Q22": 0.02,
            "R11": 0.02,
            "R12": 0.02,
            "R21": 0.02,
            "R22": 0.0,
        },
        init_names=[],
    )

LG_obj_4.mif(
  rw_sd = rw_sd,
  J = 1000,
  M = 2,
  a = 0.5,
  key = subkey
)
```

The `pypomp` implementation of IF algorithm in `mif()` inherits the structure of R's `pomp` package while adding JAX compilation, providing a substantial performance improvements over R implementation.

Iterated Filtering with Automatic Differentiation

Iterated filtering with automatic differentiation (IFAD) (Tan, Hooker, and Ionides 2024) is a two-stage method that combines the complementary advantages of iterated filtering (IF2) and a new proposed MOP-α gradient method. The motivation is that IF2 is effective at global exploration of the parameter space and can quickly approach the neighborhood of the global maximum, but it struggles to achieve exact local convergence to the global maximum once it has reached a high-likelihood neighborhood. In contrast, by using the differentiable particle filter, the MOP-α gradient method can conduct local refinements via gradients (and optionally Hessians) in the parameter space, but shows weaker ability at conducting a global exploration (Tan, Hooker, and Ionides 2024). IFAD addresses this tradeoff by first running IF2 to approach the promising region and then swithcing to a gradient-based method local search to further refine the parameter estimates. IFAD improves robustness by combining exploration and exploitation through the two-stage work.

The Algorithm 5 outlines the IFAD workflow:

Algorithm 5 IFAD: Iterated Filtering with Automatic Differentiation

```
Require: Number of particles $J$, timesteps $N$, IF2 cooling schedule $\eta_m$, MOP-$\alpha$ discounting parameter $\alpha$, initial parameter $\theta_0$, iteration index $m = 0$.
Run IF2 until initial "convergence" under cooling schedule $\eta_m$, or for a fixed number of iterations, to obtain $\{\Theta_j, j = 1, \dots, J\}$.
Set $\theta_m := \frac{1}{J}\sum_{j=1}^{J} \Theta_j$.
while procedure not converged do
    Run Algorithm 3 (MOP-$\alpha$ filter) to obtain $\hat{\ell}(\theta_m)$.
    Obtain gradient and Hessian:
        $g(\theta_m) = \nabla_{\theta_m}\big(-\hat{\ell}(\theta_m)\big), \quad H(\theta_m) \quad \text{s.t. } \lambda_{\min}(H(\theta_m)) > c.$
    Update parameter:
        $\theta_{m+1} := \theta_m - \eta_m\, H(\theta_m)^{-1} g(\theta_m).$
    Set $m := m + 1$.
end while
Ensure: Return $\hat{\theta} := \theta_m$.
```

In `pypomp`, the MOP-α gradent local search stage is implemented by the `train()`, which utilizes JAX's automatic differentiation features to iteratively optimize MOP-α objective. While IFAD is not implemented as an independent function, users can always run `mif()` and `train()` to execute the two phases sequentially. For example, the following code first runs IF2 for M_1 iterations to obtain a high-likelihood starting point, and then applies Newton-type optimization for M_2 iterations to refine the estimate:

```
key, subkey = jax.random.split(key)
LG_obj_5 = pp.Pomp(
    rinit=rinit,
    rproc=rproc,
    dmeas=dmeas,
    rmeas=rmeas,
    t0=0.0,
    nstep=1,
    ydim=2,
    ys=ys,
    theta=theta_list,
    statenames=["X1", "X2"],
)
LG_obj_5.mif(
    rw_sd=rw_sd,
    J=1000,
    M=3,
    a=0.5,
    key=subkey
    )
LG_obj_5.train(
    J=1000,
    M=3,
    eta=0.0025, # learning rate
    optimizer="Newton"
)
```

In `pypomp`, `train()` serves as a gradient-based optimizer for exploring the parameter space. Each iteration follows the standard optimization working flow: starting from the current parameter estimate, it computes gradient (and Hessian) of the log-likelihood via `jax.grad()` and `jax.Hessian()`, then utilizes the choosen internal optimizer to update the direction, obtains the learning rate, and finally updates the parameters as well as evaluates the new likelihood under the updated parameters. `train()` supports several common gradient-based optimizers for exploring the parameter space, including Newton's method, weighted Newton's method, Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm, and stochastic gradient descent method (SGD). In addition, An optional Armijo backtracking line search method (Wills and Schön 2021) is available for adaptively determining the learning rate (step size) under stochastic Quasi-Newton methods.

In `train()`, there is a key argument `n_monitors`, controlling the number of log-likelihood values are evaluated during each training step. When setting `n_monitors=0`, the algorithm totally skip the evaluating process and conduct the gradient-only optimization. It is the fastest choice, but doesn't support the line search method for learning rate determination, as it utilizes the estimated log-likelihood as the current objective function value. `n_monitors=1` is a lightest monitoring form, where users can get a one log-likelihood evaluation per training step. Larger `n_monitors` reduces the log-likelihood estimate variance in particle filtering using Monte Carlo average at additional computational cost. There is another useful Boolean flag, `scale`, which controls whether the search direction is normalized to unit length. A normalized search direction can provide a more conservative update when the gradients or Hessians are unstable.

Similarly to `mif()`, each call to `train()` creates a `PompTrainResult` object, which contains the algorithmic settings and an `xarray.DataArray` trace `traces_da` with dimensions `(replicate, iteration, variable)`. The result object is appended into `results_history`.

The core highlight of IFAD in algorithmic implementation lies in its internal automatic differentiation and just-in-time compilation powered by JAX. Via JAX, gradients and Hessian matrices for the MOP-α log-likelihood function can be directly obtained without manual derivation, ensuring correctness, efficiency and operability. This also enables flexible use of advanced gradient-based optimizers in each training step, such as Newton's method, quasi-Newton methods, BFGS, gradient descent and stochastic line search. Additionally, JAX's GPU compilation capabilities enable these optimization processes to execute efficiently, making IFAD not only theoretically feasible but also capable of effectively handling large complex POMP models in practical algorithmic implementations.

Data Analysis with pypomp / Tutorials and data analysis examples

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

Discussion

Abkemeier, Aaron, Jun Chen, Edward Ionides, Jesse Wheeler, and Kevin Tan. 2024. "Pypomp."

Arulampalam, M. S., S. Maskell, N. Gordon, and T. Clapp. 2002. "A Tutorial on Particle Filters for Online Nonlinear/Non-Gaussian Bayesian Tracking." *IEEE Transactions on Signal Processing* 50 (2): 174–88. https://doi.org/10.1109/78.978374.

Asfaw, Kidus, Joonha Park, Aaron A. King, and Edward L. Ionides. 2024. "spatPomp: An R Package for Spatiotemporal Partially Observed Markov Process Models." *Journal of Open Source Software* 9 (104): 7008. https://doi.org/10.21105/joss.07008.

Auger-Méthé, Marie, Ken Newman, Diana Cole, Fanny Empacher, Rowenna Gryba, Aaron A. King, Vianey Leos-Barajas, et al. 2021. "A Guide to State-Space Modeling of Ecological Time Series." *Ecological Monographs* 91 (4): e01470. https://doi.org/10.1002/ecm.1470.

Bjørnstad, Ottar N., and Bryan T. Grenfell. 2001. "Noisy Clockwork: Time Series Analysis of Population Fluctuations in Animals." *Science* 293: 638–43. https://doi.org/10.1126/science.1062226.

Blackwood, J. C., D. G. Streicker, S. Altizer, and P. Rohani. 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): 20837–42. https://doi.org/10.1073/pnas.1308817110.

Bradbury, James, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, et al. 2018. "JAX: Composable Transformations of Python+NumPy Programs."

Bretó, Carles. 2014. "On Idiosyncratic Stochasticity of Financial Leverage Effects." *Statistics & Probability Letters* 91: 20–26. https://doi.org/10.1016/j.spl.2014.04.003.

Bretó, Carles, Jesse Wheeler, Aaron A. King, and Edward L. Ionides. 2025. "panelPomp: Analysis of Panel Data via Partially Observed Markov Processes in R." arXiv. https://doi.org/10.48550/arXiv.2410.07934.

Fox, S. J., M. Lachmann, M. Tec, R. Pasco, S. Woody, Z. Du, X. Wang, et al. 2022. "Real-Time Pandemic Surveillance Using Hospital Admissions and Mobility Data." *Proceedings of the National Academy of Sciences* 119 (7): e2111870119. https://doi.org/10.1073/pnas.2111870119.

He, Daihai, Edward L. Ionides, and Aaron A. King. 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): 271–83. https://doi.org/10.1098/rsif.2009.0151.

Ionides, E. L., C. Breto, and A. A. King. 2006. "Inference for Nonlinear Dynamical Systems." *Proceedings of the National Academy of Sciences* 103 (49): 18438–43. https://doi.org/10.1073/pnas.0603181103.

Ionides, E. L., C. Breto, J. Park, R. A. Smith, and A. A. King. 2017. "Monte Carlo Profile Confidence Intervals for Dynamic Systems." *Journal of The Royal Society Interface* 14 (132): 20170126. https://doi.org/10.1098/rsif.2017.0126.

Ionides, Edward L., Dao Nguyen, Yves Atchadé, Stilian Stoev, and Aaron A. King. 2015. "Inference for Dynamic and Latent Variable Models via Iterated, Perturbed Bayes Maps." *Proceedings of the National Academy of Sciences* 112 (3): 719–24. https://doi.org/10.1073/pnas.1410597112.

King, Aaron A., Edward L. Ionides, Mercedes Pascual, and Menno J. Bouma. 2008. "Inapparent Infections and Cholera Dynamics." *Nature* 454 (7206): 877–80. https://doi.org/10.1038/nature07084.

King, Aaron A., Dao Nguyen, and Edward L. Ionides. 2016. "Statistical Inference for Partially Observed Markov Processes via the *R* Package **Pomp**." *Journal of Statistical Software* 69 (12). https://doi.org/10.18637/jss.v069.i12.

Korevaar, Hannah, C. Jessica Metcalf, and Bryan T. Grenfell. 2020. "Structure, Space and Size: Competing Drivers of Variation in Urban and Rural Measles Transmission." *Journal of The Royal Society Interface* 17 (168): 20200010. https://doi.org/10.1098/rsif.2020.0010.

Marino, J. A. Jr., S. D. Peacor, D. B. Bunnell, H. A. Vanderploeg, S. A. Pothoven, A. K. Elgin, J. R. Bence, J. Jiao, and E. L. Ionides. 2019. "Evaluating Consumptive and Nonconsumptive Predator Effects on Prey Density Using Field Time-Series Data." *Ecology* 100 (3): e02583. https://doi.org/10.1002/ecy.2583.

Mietchen, Matthew S., Erin Clancey, Corrin McMichael, and Eric T. Lofgren. 2024. "Estimating SARS-CoV-2 Transmission Parameters Between Coinciding Outbreaks in a University Population and the Surrounding Community." https://doi.org/10.1101/2024.01.10.24301116.

Sun, Weizhe. 2024. "Model Based Inference of Stochastic Volatility via Iterated Filtering," April.

Tan, Kevin, Giles Hooker, and Edward L. Ionides. 2024. "Accelerated Inference for Partially Observed Markov Processes Using Automatic Differentiation." arXiv. https://doi.org/10.48550/arXiv.2407.03085.

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

Wills, Adrian G., and Thomas B. Schön. 2021. "Stochastic Quasi-Newton with Line-Search Regularisation." *Automatica* 127: 109503. https://doi.org/10.1016/j.automatica.2021.109503.