Lesson 4: Iterated Filtering - Principles and Practice

Aaron A. King Edward L. Ionides Translated in pypomp by Kunyang He

2025-12-23

Table of contents I

Table of contents II

This lesson covers likelihood estimation via the method of iterated filtering (IF2). It presupposes familiarity with building POMP models and particle filtering from Lessons 2 and 3.

Learning Objectives

1. To review the available options for inference on POMP models, to put iterated filtering in context.
2. To understand how iterated filtering algorithms carry out repeated particle filtering operations, with randomly perturbed parameter values, in order to maximize the likelihood.
3. To gain experience carrying out statistical investigations using iterated filtering in a relatively simple situation: fitting an SIR model to data from a measles outbreak.

Plug-and-Play Methods I

- Inference methodology that calls `rprocess` but not `dprocess` is said to be **plug-and-play**.
- All popular modern Monte Carlo methods for POMP models are in this category.
- "Simulation-based" is equivalent to "plug-and-play".

Plug-and-Play Methods II

- Plug-and-play methods can call `dmeasure`. A method that uses only `rprocess` and `rmeasure` is called "doubly plug-and-play".
- Two non-plug-and-play methods—**EM** and **MCMC**—have theoretical convergence problems for nonlinear POMP models.

POMP Inference Methodologies

	Frequentist	Bayesian
Plug-and-play, Full-info	Iterated filtering	Particle MCMC
Plug-and-play, Feature	Simulated moments	ABC
Not plug-and-play, Full-info	EM, Kalman filter	MCMC

Full-Information, Plug-and-Play, Frequentist Methods I

- **Iterated filtering methods** are the only currently available, full-information, plug-and-play, frequentist methods for POMP models.

In the **IF2 algorithm**:

1. Each iteration consists of a particle filter, carried out with the parameter vector, for each particle, doing a random walk.
2. At the end of the time series, the collection of parameter vectors is recycled as starting parameters for the next iteration.
3. The random-walk variance decreases at each iteration.

Analogy with Natural Selection I

- The **parameters** characterize the genotype
- The **swarm of particles** is a population
- The **likelihood** is the analogue of **fitness**
- Each successive observation is a **new generation**

Analogy with Natural Selection II

- Since particles reproduce in proportion to their likelihood, the particle filter acts like **natural selection**
- The artificial perturbations correspond to **mutation**
- IF2 increases the fitness of the population of particles

Import Required Packages

```
import jax.numpy as jnp
import jax
import pandas as pd
import numpy as np
from pypomp import Pomp, RWSigma
from pypomp.util import logmeanexp, logmeanexp_se
import matplotlib.pyplot as plt
import time
```

Load Data and Build Model I

```
# Download and prepare data
meas = (pd.read_csv(
          "https://kingaa.github.io/sbied/stochsim/Measles_Consett_1948.csv")
          .loc[:, ["week", "cases"]]
          .rename(columns={"week": "time", "cases": "reports"})
          .set_index("time")
          .astype(float))

ys = meas.copy()
ys.columns = pd.Index(["reports"])
print(f"Data shape: {ys.shape}")
print(ys.head())
```

```
Data shape: (53, 1)
      reports
time
1         0.0
2         0.0
3         2.0
```

Load Data and Build Model II

```
4           0.0
5           3.0
```

Load Data and Build Model III

```
# Helper functions for negative binomial
def nbinom_logpmf(x, k, mu):
    """Log PMF of NegBin(k, mu) that is robust when mu == 0."""
    x = jnp.asarray(x)
    k = jnp.asarray(k)
    mu = jnp.asarray(mu)
    logp_zero = jnp.where(x == 0, 0.0, -jnp.inf)
    safe_mu = jnp.where(mu == 0.0, 1.0, mu)
    core = (jax.scipy.special.gammaln(k + x)
            - jax.scipy.special.gammaln(k)
            - jax.scipy.special.gammaln(x + 1)
            + k * jnp.log(k / (k + safe_mu))
            + x * jnp.log(safe_mu / (k + safe_mu)))
    return jnp.where(mu == 0.0, logp_zero, core)

def rnbinom(key, k, mu):
    """Sample from NegBin(k, mu) via Gamma-Poisson mixture."""
    key_g, key_p = jax.random.split(key)
    safe_mu = jnp.maximum(mu, 1e-10)
    lam = jax.random.gamma(key_g, k) * (safe_mu / k)
    return jax.random.poisson(key_p, lam)
```

Load Data and Build Model IV

```python
# SIR model components
def rinit(theta_, key, covars, t0):
    """Initialize the SIR state."""
    N = theta_["N"]
    eta = theta_["eta"]
    S0 = jnp.round(N * eta)
    I0 = 1.0
    R0 = jnp.round(N * (1 - eta)) - 1.0
    H0 = 0.0  # Accumulator for incidence
    return {"S": S0, "I": I0, "R": R0, "H": H0}

def rproc(X_, theta_, key, covars, t, dt):
    """SIR process model with Euler-binomial transitions."""
    S, I, R, H = X_["S"], X_["I"], X_["R"], X_["H"]
    Beta = theta_["Beta"]
    mu_IR = theta_["mu_IR"]
    N = theta_["N"]

    # Transition probabilities
    p_SI = 1.0 - jnp.exp(-Beta * I / N * dt)
    p_IR = 1.0 - jnp.exp(-mu_IR * dt)

    # Binomial transitions
    key_SI, key_IR = jax.random.split(key)
```

Specifying Random Walk SDs I

In pypomp, random walk standard deviations for IF2 are specified using the `RWSigma` class:

```
# Define random walk standard deviations
# init_names specifies "initial value parameters" (IVPs)
# These are perturbed only at the initial time
# IMPORTANT: ALL parameters must be listed in sigmas dict
# Use 0.0 for parameters you want to keep fixed

rw_sd = RWSigma(
    sigmas={
        "Beta": 0.02,    # Random walk SD for Beta
        "mu_IR": 0.0,    # Fixed - no perturbation
        "N": 0.0,        # Fixed - no perturbation
        "rho": 0.02,     # Random walk SD for rho
        "eta": 0.02,     # Random walk SD for eta (IVP)
        "k": 0.0         # Fixed - no perturbation
    },
    init_names=["eta"]  # eta is an initial value parameter
)
```

Specifying Random Walk SDs II

This is analogous to `rw.sd=rw_sd(Beta=0.02, rho=0.02, eta=ivp(0.02))` in R-pomp.

Local Search I

```
# Run IF2 from our starting point
key = jax.random.key(482947940)

# a=0.5 means perturbations are reduced to half after 50 iterations
measSIR.mif(
    J=2000,          # Number of particles
    M=50,            # Number of IF2 iterations
    rw_sd=rw_sd,     # Random walk SDs
    a=0.5,           # Cooling fraction
    key=key
)

mif_result = measSIR.results_history[-1]  # Use [-1] to get last result
print(f"Execution time: {mif_result.execution_time:.2f} seconds")
```

```
Execution time: 109.33 seconds
```

Local Search II

```
# Get traces - the traces are stored in traces_da (xarray DataArray)
# traces_da has dimensions: ('replicate', 'iteration', 'variable')
traces_da = mif_result.traces_da

# Select the first replicate (index 0)
# Since we only have one starting point, replicate=0
traces_rep0 = traces_da.isel(replicate=0)

iterations = traces_rep0.coords['iteration'].values
variables = traces_rep0.coords['variable'].values

# Plot the traces
fig, axes = plt.subplots(2, 2, figsize=(8, 6))

ax = axes[0, 0]
if 'logLik' in variables:
    ax.plot(iterations, traces_rep0.sel(variable='logLik').values)
ax.set(xlabel='Iteration', ylabel='Log-likelihood')
ax.grid(alpha=0.3)

ax = axes[0, 1]
if 'Beta' in variables:
    ax.plot(iterations, traces_rep0.sel(variable='Beta').values)
ax.set(xlabel='Iteration', ylabel=r'$\beta$')
```

Estimating the Likelihood I

The mif log-likelihood is not reliable for inference. Evaluate with pfilter:

```
# Get final parameters from mif
# theta is a PompParameters object (list of dicts)
final_theta = measSIR.theta

# Evaluate likelihood at the endpoint
key = jax.random.key(900242057)
measSIR.pfilter(key=key, J=5000, reps=10)

pf_result = measSIR.results_history[-1]
# logLiks is an xarray with shape (n_theta, reps)
logliks = pf_result.logLiks.values.flatten()

ll_est = logmeanexp(logliks)
ll_se = logmeanexp_se(logliks)

print(f"Log-likelihood: {ll_est:.2f} (SE: {ll_se:.2f})")
print(f"Final parameters:")
for k, v in measSIR.theta[0].items():
    print(f"  {k}: {v:.4f}")
```

Estimating the Likelihood II

```
Log-likelihood: -107.37 (SE: 0.03)
Final parameters:
  Beta: 13.6921
  mu_IR: 2.0000
  N: 38000.0000
  eta: 0.1626
  rho: 0.2047
  k: 10.0000
```

Running Multiple IF2 Searches I

Running Multiple IF2 Searches II

```
def run_mif_from_start(theta_start, seed):
    """Run MIF from a given starting point."""
    pomp_obj = Pomp(
        rinit=rinit, rproc=rproc, dmeas=dmeas, rmeas=rmeas,
        ys=ys, theta=theta_start, statenames=statenames,
        t0=0.0, dt=1/7, accumvars=(3,), ydim=1  # accumvars=(3,) for H
    )

    key = jax.random.key(seed)
    pomp_obj.mif(J=2000, M=50, rw_sd=rw_sd, a=0.5, key=key)

    # Evaluate likelihood at final parameters
    key = jax.random.key(seed + 1000)
    pomp_obj.pfilter(key=key, J=5000, reps=5)

    pf_result = pomp_obj.results_history[-1]
    logliks = pf_result.logLiks.values.flatten()

    # Get final parameters from theta (PompParameters object)
    final_params = pomp_obj.theta[0]  # First (and only) parameter set

    return {
        **final_params,
        'loglik': logmeanexp(logliks),
```

Computing a Profile I

To profile over eta, we fix eta and optimize over other parameters:

Computing a Profile II

```
def compute_profile_point(eta_val, seed):
    """Compute one point of the profile likelihood over eta."""
    theta_start = theta.copy()
    theta_start["eta"] = eta_val
    theta_start["Beta"] = np.random.uniform(10, 40)
    theta_start["rho"] = np.random.uniform(0.1, 0.6)

    pomp_obj = Pomp(
        rinit=rinit, rproc=rproc, dmeas=dmeas, rmeas=rmeas,
        ys=ys, theta=theta_start, statenames=statenames,
        t0=0.0, dt=1/7, accumvars=(3,), ydim=1  # accumvars=(3,) for H
    )

    # eta NOT in sigmas with non-zero value, so it stays fixed during MIF
    # ALL parameters must be listed in sigmas
    rw_sd_profile = RWSigma(
        sigmas={
            "Beta": 0.02,
            "mu_IR": 0.0,
            "N": 0.0,
            "rho": 0.02,
            "eta": 0.0,   # Fixed for profiling
            "k": 0.0
        },
```

Exercise 4.1: Fitting the SEIR Model

Estimate SEIR model parameters using IF2. Track computation time and use run-levels.

Exercise 4.2: Fitting All Parameters

Estimate all SIR parameters including k. How does the fit improve?

Exercise 4.3: Profile over Beta

Construct profile likelihood over Beta and R0. Which is better determined?

Exercise 4.4: Checking Source Code

Does your code implement the described model? What are solutions for reproducibility?

Exercise 4.5: Debugging rprocess

How would you identify and fix errors in rprocess?

Summary I

1. **IF2** is the primary full-information, plug-and-play, frequentist method for POMP models
2. The algorithm perturbs parameters with decreasing random walks and resamples both states AND parameters
3. **pypomp API for IF2:**
   - `mif(J, M, rw_sd, a, key)` runs IF2
   - `RWSigma(sigmas={...}, init_names=[...])` specifies perturbations
   - `results_history[-1].traces_da` gets parameter traces as xarray
4. Always evaluate likelihood at MIF endpoints with `pfilter`
5. Access final parameters via `pomp.theta[0]` (PompParameters is a list of dicts)

Summary II

6. **Key corrections:**
   - `accumvars` must be a tuple of integer indices, e.g., `(3,)`, not `["H"]`
   - `RWSigma` requires **ALL parameters** in `sigmas` dict - use `0.0` for fixed params
   - `traces_da` has dimensions `('replicate', 'iteration', 'variable')` - use `.isel(replicate=0)` to select
   - `rmeas` should return a JAX array with shape `(ydim,)`

References I