Lesson 3: Exercises on Likelihood-based Inference for POMP Models

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

2025-12-23

Table of contents I

Table of contents II

Table of contents III

This document contains worked solutions to the exercises from Lesson 3 on likelihood-based inference for POMP models, implemented using `pypomp`.

Import Required Packages

```
import jax.numpy as jnp
import jax
import pandas as pd
import numpy as np
import pypomp as pp
import matplotlib.pyplot as plt
import time
from scipy import stats
```

Load Data and Build Model

```
# 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"])
```

Helper Functions

```
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)
    lam = jax.random.gamma(key_g, k) * (mu / k)
    return jax.random.poisson(key_p, lam)
```

SIR Model Components

```
def rinit(theta_, key, covars, t0):
    """Initial state simulator for SIR model."""
    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
    return {"S": S0, "I": I0, "R": R0, "H": H0}

def rproc(X_, theta_, key, covars, t, dt):
    """Process simulator for SIR model."""
    S, I, R, H = X_["S"], X_["I"], X_["R"], X_["H"]
    Beta = theta_["Beta"]
    mu_IR = theta_["mu_IR"]
    N = theta_["N"]

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

    key_SI, key_IR = jax.random.split(key)
    dN_SI = jax.random.binomial(key_SI, n=S.astype(jnp.int32), p=p_SI)
    dN_IR = jax.random.binomial(key_IR, n=I.astype(jnp.int32), p=p_IR)
```

Create POMP Object

```
theta = {
    "Beta": 15.0,
    "mu_IR": 0.5,
    "N": 38000.0,
    "eta": 0.06,
    "rho": 0.5,
    "k": 10.0
}

statenames = ["S", "I", "R", "H"]

measSIR = pp.Pomp(
    rinit=rinit,
    rproc=rproc,
    dmeas=dmeas,
    rmeas=rmeas,
    ys=ys,
    theta=theta,
    statenames=statenames,
    t0=0.0,
    nstep=7,
    accumvars=(3,),
    ydim=1,
    covars=None
```

Problem Statement

What is the difference between a likelihood **slice** and a **profile**? What is the consequence of this difference for the statistical interpretation of these plots? How should you decide whether to compute a profile or a slice?

Solution I

Likelihood Slice:

A likelihood slice fixes all parameters except one at specified values and varies only that one parameter. For example, a slice in the β direction at our current parameter values:

$$\ell_{\text{slice}}(\beta) = \ell(\beta, \mu_{IR}^0, \eta^0, \rho^0, k^0, N^0)$$

Profile Likelihood:

A profile likelihood **optimizes** over all other parameters for each value of the parameter of interest:

$$\ell^{\text{profile}}(\beta) = \max_{\mu_{IR}, \eta, \rho, k, N} \ell(\beta, \mu_{IR}, \eta, \rho, k, N)$$

Solution II

Consequences for Statistical Interpretation:

1. **Slices** show the likelihood surface conditional on fixed values of other parameters. They do not account for the fact that other parameters might be adjusted.
2. **Profiles** provide valid confidence intervals via Wilks' theorem. The profile likelihood accounts for uncertainty in nuisance parameters.
3. A slice can **underestimate** how good the likelihood can be at a parameter value, because it doesn't optimize over other parameters.

Solution III

When to Use Which:

Use Slice When	Use Profile When
Quick exploration	Formal inference
Understanding local geometry	Constructing confidence intervals
Checking parameter sensitivity	Model comparison
Computational resources limited	Results need to be published

Solution IV

Profiles are computationally much more expensive (require optimization at each point) but are necessary for proper statistical inference.

Problem Statement

- How much computer processing time does a particle filter take?
- How does this scale with the number of particles?

Form a conjecture based upon your understanding of the algorithm. Test your conjecture by running a sequence of particle filter operations, with increasing numbers of particles (J), measuring the time taken for each one. Plot and interpret your results.

Conjecture I

Theoretical Analysis:

The particle filter algorithm consists of:

1. **Initialize**: $O(J)$ operations to initialize J particles
2. **For each time step** $n = 1, \ldots, N$**:**
   - **Predict**: Simulate each particle forward: $O(J)$
   - **Weight**: Compute weights for each particle: $O(J)$
   - **Resample**: Sample J particles with replacement: $O(J)$

Total complexity: $O(N \times J)$

Conjecture II

Conjecture: The computational time should scale **linearly** with the number of particles J.

With JAX, there may be: - Some fixed overhead from JIT compilation - Potentially better-than-linear scaling for small J due to vectorization - Linear scaling for large J

Testing the Conjecture I

Testing the Conjecture II

```
def time_pfilter(pomp_obj, J, n_reps=3):
    """Time how long a particle filter takes."""
    times = []
    for rep in range(n_reps):
        key = jax.random.key(rep + int(J))
        start = time.time()
        pomp_obj.pfilter(key=key, J=J, reps=1)
        # Block until computation is complete (important for JAX timing)
        result = pomp_obj.results_history.last()
        _ = result.logLiks.values  # Force computation
        elapsed = time.time() - start
        times.append(elapsed)
    return np.mean(times), np.std(times)

# Test different numbers of particles
J_values = [100, 500, 1000, 2000, 5000, 10000, 20000, 50000]

timing_results = []
for J in J_values:
    mean_time, std_time = time_pfilter(measSIR, J, n_reps=3)
    timing_results.append({
        'J': J,
        'time_mean': mean_time,
        'time_std': std_time
```

Interpretation

The results confirm that:

1. **Linear scaling**: Time scales linearly with the number of particles (as expected from $O(N \times J)$ complexity)
2. **Fixed overhead**: There is a small fixed cost for JIT compilation and setup
3. **Marginal cost**: The marginal cost is approximately constant per particle per time step
4. **Practical implication**: Doubling the number of particles approximately doubles the computation time

Problem Statement I

Set up a likelihood evaluation for the measles model, choosing the numbers of particles and replications so that your evaluation takes approximately one minute on your machine.

- Provide a Monte Carlo standard error for your estimate.
- Comment on the bias of your estimate.

Solution I

First, let's estimate how many replications we can do in one minute:

```
# Time a single evaluation with moderate J
J_test = 10000
key = jax.random.key(999)

start = time.time()
measSIR.pfilter(key=key, J=J_test, reps=1)
result = measSIR.results_history.last()
_ = result.logLiks.values  # Force computation
test_time = time.time() - start

print(f"Time for J={J_test}: {test_time:.3f} seconds")

# Estimate how many reps in 60 seconds
target_time = 60  # seconds
estimated_reps = int(target_time / test_time)
print(f"Estimated replications in {target_time}s: {estimated_reps}")
```

```
Time for J=10000: 6.033 seconds
Estimated replications in 60s: 9
```

Solution II

```
# Run the likelihood estimation
J = 10000
n_reps = min(50, max(10, estimated_reps))  # Reasonable range

print(f"Running {n_reps} replicates with J={J}...")
start_time = time.time()

key = jax.random.key(1000)
measSIR.pfilter(key=key, J=J, reps=n_reps)

# Get results
result = measSIR.results_history.last()
logliks = result.logLiks.values[0, :]  # All replicates

total_time = time.time() - start_time
print(f"Total time: {total_time:.1f} seconds")
```

```
Running 10 replicates with J=10000...
Total time: 28.6 seconds
```

Solution III

```
# Compute estimates using pypomp's utility functions
ll_array = np.array(logliks)

# Simple mean and SE
mean_ll = np.mean(ll_array)
se_ll = np.std(ll_array, ddof=1) / np.sqrt(n_reps)

# logmeanexp estimate (less biased for likelihood)
lme_ll = pp.logmeanexp(ll_array)
lme_se = pp.logmeanexp_se(ll_array)

print(f"\nResults:")
print(f"  Number of particles: {J}")
print(f"  Number of replications: {n_reps}")
print(f"  Simple mean log-likelihood: {mean_ll:.4f} (SE: {se_ll:.4f})")
print(f"  logmeanexp estimate: {lme_ll:.4f} (SE: {lme_se:.4f})")
```

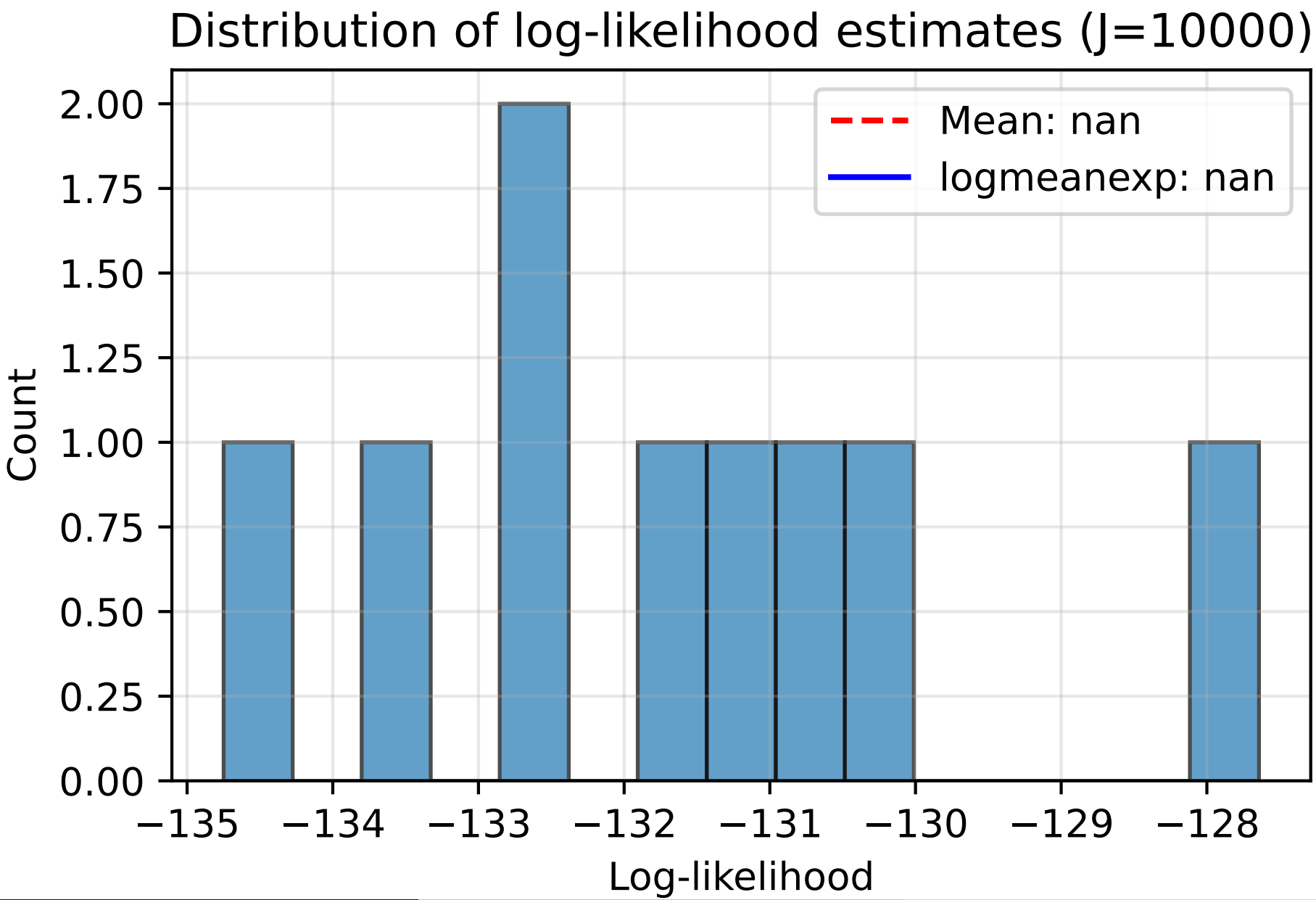
Solution IV

```
Results:
  Number of particles: 10000
  Number of replications: 10
  Simple mean log-likelihood: nan (SE: nan)
  logmeanexp estimate: nan (SE: nan)
```

Solution V

```
# Visualize the distribution of log-likelihoods
fig, ax = plt.subplots(figsize=(5, 3.5))
ax.hist(ll_array, bins=15, edgecolor='black', alpha=0.7)
ax.axvline(mean_ll, color='red', linestyle='--',
           label=f'Mean: {mean_ll:.2f}')
ax.axvline(lme_ll, color='blue', linestyle='-',
           label=f'logmeanexp: {lme_ll:.2f}')
ax.set(xlabel='Log-likelihood', ylabel='Count',
       title=f'Distribution of log-likelihood estimates (J={J})')
ax.legend()
ax.grid(alpha=0.3)
plt.tight_layout()
plt.show()
```

Solution VI


Distribution of log-likelihood estimates (J=10000)
Mean: nan
logmeanexp: nan
Count
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−135
−134
−133
−132
−131
−130
−129
−128
Log-likelihood

Comment on Bias I

The particle filter provides:

- An **unbiased** estimate of the likelihood $\mathcal{L}(\theta)$
- A **biased** estimate of the log-likelihood $\ell(\theta)$

This is because:

$$\mathbb{E}[\log \hat{\mathcal{L}}] \neq \log \mathbb{E}[\hat{\mathcal{L}}] = \log \mathcal{L}$$

By Jensen's inequality, since $\log$ is concave:

$$\mathbb{E}[\log \hat{\mathcal{L}}] \leq \log \mathbb{E}[\hat{\mathcal{L}}]$$

The bias is negative — the simple average of log-likelihoods underestimates the true log-likelihood.

Comment on Bias II

The `logmeanexp` function helps:

Instead of computing $\frac{1}{n}\sum_i \log \hat{\mathcal{L}}_i$, we compute:

$$\log\left(\frac{1}{n}\sum_i \hat{\mathcal{L}}_i\right) = \log\left(\frac{1}{n}\sum_i e^{\log \hat{\mathcal{L}}_i}\right)$$

This averages the likelihoods (unbiased) and then takes the log, reducing bias.

Comment on Bias III

Reducing bias:

1. **Increase particles (J)**: More particles $\rightarrow$ lower variance $\rightarrow$ less bias
2. **Use logmeanexp**: Average likelihoods instead of log-likelihoods
3. **More replications**: Better estimate of the mean

The difference between `mean_ll` and `lme_ll` in our results illustrates this bias.

Problem Statement

Compute several likelihood slices in the η direction.

Solution I

Solution II

```
def compute_likelihood_slice(param_name, param_values, base_theta,
                             J=5000, n_reps=3):
    """Compute log-likelihood for a slice of parameter values."""
    results = []
    for val in param_values:
        theta_test = base_theta.copy()
        theta_test[param_name] = val

        pomp_test = pp.Pomp(
            rinit=rinit, rproc=rproc, dmeas=dmeas, rmeas=rmeas,
            ys=ys, theta=theta_test, statenames=statenames,
            t0=0.0, nstep=7, accumvars=(3,), ydim=1, covars=None
        )

        key = jax.random.key(int(val * 10000))
        pomp_test.pfilter(key=key, J=J, reps=n_reps)

        pf_result = pomp_test.results_history.last()
        logliks_arr = pf_result.logLiks.values[0, :]

        results.append({
            param_name: val,
            'loglik': pp.logmeanexp(logliks_arr),
            'loglik_se': pp.logmeanexp_se(logliks_arr)
```

Interpretation

The slices reveal:

1. The optimal η depends on the value of β
2. There is a ridge-like relationship between η and β
3. This suggests these parameters may be partially **non-identifiable** or have a **functional relationship**

Problem Statement

Compute a slice of the likelihood in the β-η plane.

Solution I

Solution II

```
def compute_2d_surface(param1_name, param1_vals, param2_name, param2_vals,
                       base_theta, J=2000, n_reps=2):
    """Compute 2D log-likelihood surface."""
    results = []

    for v1 in param1_vals:
        for v2 in param2_vals:
            theta_test = base_theta.copy()
            theta_test[param1_name] = v1
            theta_test[param2_name] = v2

            pomp_test = pp.Pomp(
                rinit=rinit, rproc=rproc, dmeas=dmeas, rmeas=rmeas,
                ys=ys, theta=theta_test, statenames=statenames,
                t0=0.0, nstep=7, accumvars=(3,), ydim=1, covars=None
            )

            key = jax.random.key(int(v1 * 1000) + int(v2 * 10000))
            pomp_test.pfilter(key=key, J=J, reps=n_reps)

            pf_result = pomp_test.results_history.last()
            logliks_arr = pf_result.logLiks.values[0, :]

            results.append({
```

Interpretation I

The 2D likelihood surface reveals several important features:

1. **Wedge-shaped relationship**: The likelihood forms a ridge running diagonally through the parameter space, indicating a trade-off between β and η.
2. **Parameter correlation**: Higher β (transmission rate) tends to be associated with lower η (initial susceptibles) at the maximum likelihood.
3. **Non-identifiability**: The elongated ridge suggests that β and η are not fully identifiable from these data alone — many combinations give similar likelihoods.

Interpretation II

4. **Monte Carlo noise**: Despite the noise in individual likelihood evaluations, the major structure of the surface is clearly visible.
5. **Optimization challenges**: The wedge shape means that gradient-based optimization may converge slowly along the ridge direction.

These features are **typical** of epidemiological models and motivate: - Using profile likelihoods for inference - Fixing some parameters at known values - Employing specialized optimization algorithms (like iterated filtering)

Key Takeaways

1. **Slices vs Profiles**: Slices fix other parameters; profiles optimize them. Profiles are needed for valid statistical inference.
2. **Computational scaling**: Particle filter time scales linearly with number of particles J.
3. **Bias in log-likelihood**: Particle filter estimates of log-likelihood are negatively biased; use `logmeanexp` and large J to reduce bias.
4. **2D surfaces**: Reveal relationships between parameters and potential identifiability issues.

References I