Lesson 4: Iterated Filtering — Principles and Practice

Aaron A. King Edward L. Ionides Kunyang He

Monday, March 23, 2026

Introduction

This tutorial covers likelihood estimation via the method of iterated filtering. It presupposes familiarity with building partially observed Markov process (POMP) objects in the `pypomp` package. This tutorial follows on from the topic of particle filtering (also known as sequential Monte Carlo) via `pfilter` in `pypomp`.

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.

Classification of statistical methods for POMP models

Many statistical methods have been proposed for inference on POMP models. The volume of research indicates both the importance and the difficulty of the problem. Let's start by considering three criteria to categorize inference methods:

- the plug-and-play property
- full-information or feature-based
- frequentist or Bayesian

Plug-and-play (also called simulation-based) methods

- Inference methodology that calls `rproc` 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".
- Historically, simulation-based meant simulating forward from initial conditions to the end of the time series. However, particle filtering methods instead consider each observation interval sequentially. They carry out multiple, carefully selected, simulations over each interval.

- Plug-and-play methods can call `dmeas`. A method that uses only `rproc` and `rmeas` is called "doubly plug-and-play".
- Two **non-plug-and-play** methods—expectation-maximization (EM) and Markov chain Monte Carlo (MCMC)—have theoretical convergence problems for nonlinear POMP models. The failures of these two workhorses of statistical computation have prompted development of alternative methodologies.

Full-information and feature-based methods

- **Full-information** methods are defined to be those based on the likelihood function for the full data (i.e., likelihood-based frequentist inference and Bayesian inference).
- **Feature-based** methods either consider a summary statistic (a function of the data) or work with an alternative to the likelihood.
- Asymptotically, full-information methods are statistically efficient and feature-based methods usually are not.

- In some cases, loss of statistical efficiency might be an acceptable tradeoff for advantages in computational efficiency.
- However:
  - Good low-dimensional summary statistics can be hard to find.
  - When using statistically inefficient methods, it can be hard to know how much information you are losing.
  - Intuition and scientific reasoning can be inadequate tools to derive informative low-dimensional summary statistics.

Bayesian and frequentist methods

- Plug-and-play Bayesian methods exist:
  - particle Markov chain Monte Carlo (PMCMC)
  - approximate Bayesian computation (ABC)
- Prior belief specification is both the strength and weakness of Bayesian methodology.
- The likelihood surface for nonlinear POMP models often contains nonlinear ridges and variations in curvature.

- These situations bring into question the appropriateness of independent priors derived from expert opinion on marginal distributions of parameters.
- They also are problematic for specification of "flat" or "uninformative" prior beliefs.
- Expert opinion can be treated as data for non-Bayesian analysis. However, our primary task is to identify the information in the data under investigation, so it can be helpful to use methods that do not force us to make our conclusions dependent on quantification of prior beliefs.

POMP inference methodologies

	Frequentist	Bayesian
Plug-and-play, Full-info	Iterated filtering	Particle MCMC
Plug-and-play, Feature	Simulated moments, synthetic likelihood	ABC, SL-based MCMC
Not plug-and-play, Full-info	EM, Kalman filter	MCMC
Not plug-and-play, Feature	Yule-Walker, extended Kalman filter	Extended Kalman filter

Full-information, plug-and-play, frequentist methods

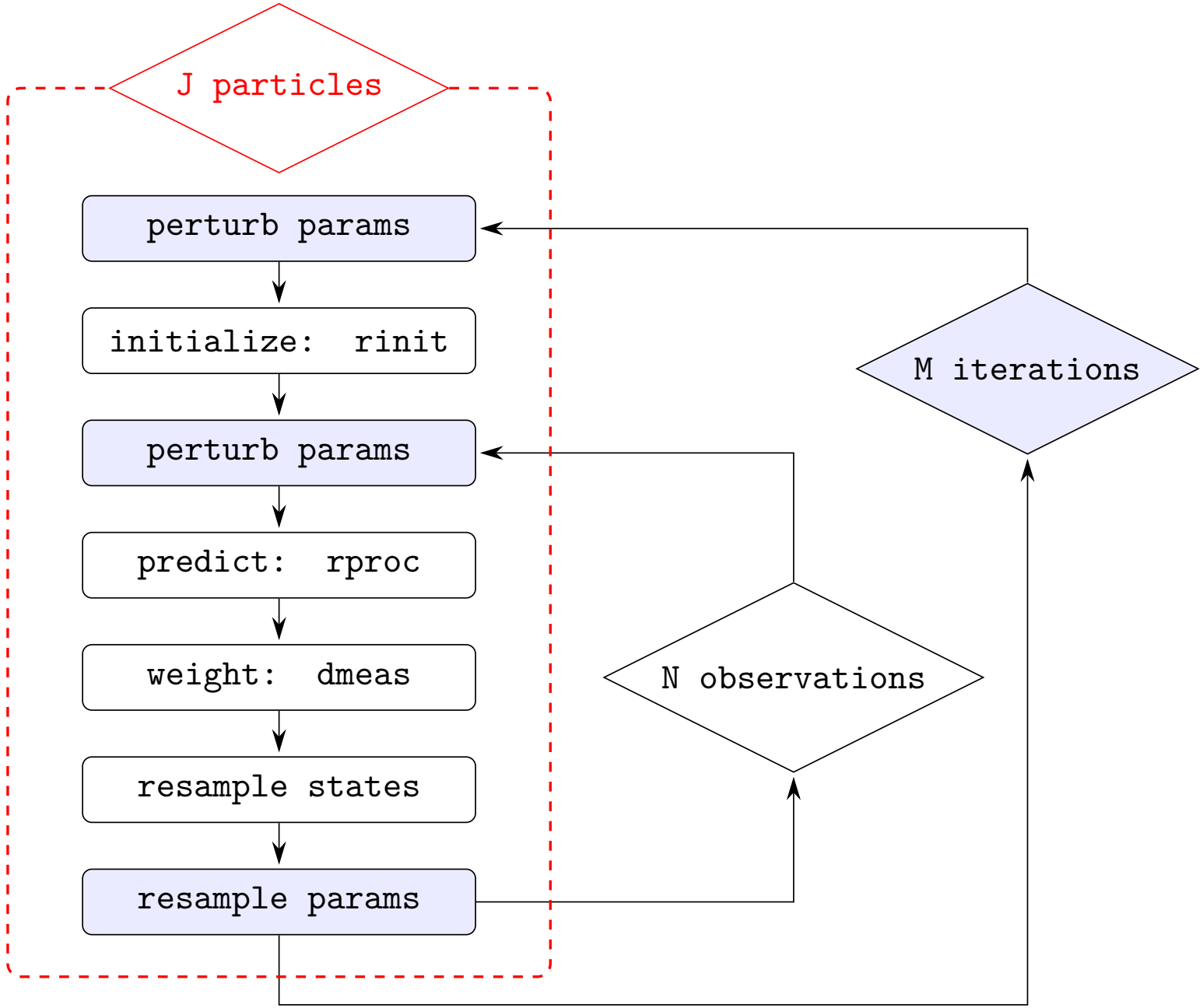
- Iterated filtering methods are the only currently available, full-information, plug-and-play, frequentist methods for POMP models.
- Iterated filtering methods have been shown to solve likelihood-based inference problems for epidemiological situations which are computationally intractable for available Bayesian methodology.

An iterated filtering algorithm (IF2)

We focus on the IF2 algorithm of Ionides et al. (2015). In this algorithm:

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

In theory, this procedure converges toward the region of parameter space maximizing the maximum likelihood. In practice, we can test this claim on examples.


J particles
perturb params
initialize: rinit
perturb params
predict: rproc
weight: dmeas
resample states
resample params
M iterations
N observations

IF2 algorithm pseudocode

Input:

- simulators for $f_{X_0}(x_0;\theta)$ and $f_{X_n|X_{n-1}}(x_n|x_{n-1};\theta)$
- evaluator for $f_{Y_n|X_n}(y_n|x_n;\theta)$
- data, $y^*_{1:N}$

Algorithmic parameters and corresponding `mif` arguments:

- number of iterations, M
- number of particles, J
- initial parameter swarm, $\{\Theta^0_j, j = 1,\dots,J\}$
- random walk standard deviation, `rw_sd`, squared to construct a diagonal variance matrix, V_n
- cooling fraction in 50 iterations, a

Output: final parameter swarm, $\{\Theta^M_j, j = 1,\dots,J\}$

IF2 algorithm pseudocode II

in practice, perturbations are Normal. In theory, they don't have to be.

Note: ignoring (H) and fixing a parameter Θ, the N loop is a particle filter.

Procedure:

1. For m in 1:M
2. $\quad \Theta^{F,m}_{0,j} \sim \mathcal{N}\left(\Theta^{m-1}_{j}, V_0\, a^{2m/50}\right)$ for j in 1:J
3. $\quad X^{F,m}_{0,j} \sim f_{X_0}(x_0; \Theta^{F,m}_{0,j})$ for j in 1:J
4. $\quad$ For n in 1:N
5. $\quad\quad \Theta^{P,m}_{n,j} \sim \mathcal{N}\left(\Theta^{F,m}_{n-1,j}, V_n\, a^{2m/50}\right)$ for j in 1:J ← prediction
6. $\quad\quad X^{P,m}_{n,j} \sim f_{X_n|X_{n-1}}(x_n|X^{F,m}_{n-1,j}; \Theta^{P,m}_{n,j})$ for j in 1:J
7. $\quad\quad w^{m}_{n,j} = f_{Y_n|X_n}(y^*_n|X^{P,m}_{n,j}; \Theta^{P,m}_{n,j})$ for j in 1:J ← weights
8. $\quad\quad$ Draw $k_{1:J}$ with $P[k_j = i] = w^m_{n,i} \Big/ \sum_{u=1}^{J} w^m_{n,u}$ ← resampling
9. $\quad\quad \Theta^{F,m}_{n,j} = \Theta^{P,m}_{n,k_j}$ and $X^{F,m}_{n,j} = X^{P,m}_{n,k_j}$ for j in 1:J
10. $\quad$ End For
11. $\quad$ Set $\Theta^{m}_{j} = \Theta^{F,m}_{N,j}$ for j in 1:J
12. End For

IF2 algorithm pseudocode III

Remarks:

- The N loop (lines 4 through 10) is a basic particle filter applied to a model with stochastic perturbations to the parameters.
- The M loop repeats this particle filter with decreasing perturbations.
- The superscript F in $\Theta_{n,j}^{F,m}$ and $X_{n,j}^{F,m}$ denote solutions to the *filtering problem*, with the particles $j = 1, \ldots, J$ providing a Monte Carlo representation of the conditional distribution at time n given data $y_{1:n}^*$ for filtering iteration m.
- The superscript P in $\Theta_{n,j}^{P,m}$ and $X_{n,j}^{P,m}$ denote solutions to the *prediction problem*.
- The *weight* $w_{n,j}^m$ gives the likelihood of the data at time n for particle j in filtering iteration m.

Analogy with evolution by natural selection

- The parameters characterize the **genotype**.
- The swarm of particles is a **population**.
- The likelihood, a measure of the compatibility between the parameters and the data, is the analogue of **fitness**.
- Each successive observation is a new **generation**.

- Since particles reproduce in each generation in proportion to their likelihood, the particle filter acts like **natural selection**.
- The artificial perturbations augment the "genetic" variance and therefore correspond to **mutation**.
- IF2 increases the **fitness** of the population of particles.
- However, because our scientific interest focuses on the model without the artificial perturbations, we decrease the intensity of the latter with successive iterations.

Setting up the model

```
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 pickle
import os
import time
from scipy.stats import chi2
from pypomp.random import (
    fast_approx_rbinom,
    fast_approx_rgamma,
    fast_approx_rpoisson,
)

cache_dir = "cache"
os.makedirs(cache_dir, exist_ok=True)
```

```
file = "Measles_Consett_1948.csv"
meas = (
    pd.read_csv(file)
    .loc[:, ["week", "cases"]]
    .rename(columns={"week": "time",
                     "cases": "reports"})
    .set_index("time")
    .astype(float)
)

ys = meas.copy()
ys.columns = pd.Index(["reports"])
```

Negative binomial functions

- These allow for overdispersed measurement distributions

```
def nbinom_logpmf(x, k, mu):
    x, k, mu = (jnp.asarray(v) for v in (x, k, mu))
    logp_zero = jnp.where(x == 0, 0.0, -jnp.inf)
    safe_mu = jnp.where(mu == 0.0, 1.0, mu)
    gammaln = jax.scipy.special.gammaln
    core = (gammaln(k + x) - gammaln(k)
            - 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):
    key_g, key_p = jax.random.split(key)
    lam = fast_approx_rgamma(key_g, k) * (mu / k)
    return fast_approx_rpoisson(key_p, lam)
```

SIR model components

```
def rinit(theta_, key, covars, t0):
    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):
    S = jnp.asarray(X_["S"])
    I = jnp.asarray(X_["I"])
    R = jnp.asarray(X_["R"])
    H = jnp.asarray(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 = fast_approx_rbinom(
        key_SI, jnp.float32(S), p_SI)
    dN_IR = fast_approx_rbinom(
        key_IR, jnp.float32(I), p_IR)

    return {"S": S - dN_SI,
            "I": I + dN_SI - dN_IR,
            "R": R + dN_IR,
            "H": H + dN_IR}
```

```
def dmeas(Y_, X_, theta_, covars, t):
    rho = theta_["rho"]
    k = theta_["k"]
    H = X_["H"]
    mu = rho * H
    return nbinom_logpmf(Y_["reports"], k, mu)

def rmeas(X_, theta_, key, covars, t):
    rho = theta_["rho"]
    k = theta_["k"]
    H = X_["H"]
    mu = rho * H
    reports = rnbinom(key, k, mu)
    return jnp.array([reports])
```

`ParTrans` ensures IF2 random walks respect parameter constraints. Parameter perturbations for IF2 are added on the **estimation scale**.

```
def to_est(theta):
    """Natural scale -> estimation scale."""
    return {
        "Beta": jnp.log(theta["Beta"]),
        "mu_IR": theta["mu_IR"],
        "N": theta["N"],
        "rho": jax.scipy.special.logit(theta["rho"]),
        "eta": jax.scipy.special.logit(theta["eta"]),
        "k": theta["k"],
    }

def from_est(theta):
    """Estimation scale -> natural scale."""
    return {
        "Beta": jnp.exp(theta["Beta"]),
        "mu_IR": theta["mu_IR"],
        "N": theta["N"],
        "rho": jax.scipy.special.expit(theta["rho"]),
        "eta": jax.scipy.special.expit(theta["eta"]),
        "k": theta["k"],
    }

par_trans = pp.ParTrans(
    to_est=to_est, from_est=from_est)
```

Setting up the estimation problem

Let's assume that the population size, N, is known accurately. We'll fix that parameter.

Let's revisit the assumption that the infectious period is 2 weeks, imagining that we have access to the results of household and clinical studies that have concluded that infected patients shed the virus for 3–4~da. We'll use these results to constrain the infectious period in our model to 3.5~da, i.e., $\mu_{IR} = 2\ \mathrm{wk}^{-1}$. We also fix $k = 10$. Later, we can relax our assumptions. We proceed to estimate β, η, and ρ.

Create POMP object

```
theta = {
    "Beta": 15.0,
    "mu_IR": 2.0,
    "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,
    par_trans=par_trans,
    t0=0.0, nstep=7,
    accumvars=("H",),
    ydim=1, covars=None
)
```

Testing the codes: filtering

Before engaging in iterated filtering, it is a good idea to check that the basic particle filter is working since we can't iterate something unless we can run it once!

```
key = jax.random.key(42)
measSIR.pfilter(key=key, J=5000, reps=1)
result = measSIR.results_history.last()
loglik = float(result.logLiks.values[0, 0])
print(f"Log-likelihood: {loglik:.1f}")
```

```
Log-likelihood: -288.8
```

Specifying random walk standard deviations

Since β will be estimated on the log scale, and ρ and η on the logit scale, we expect that perturbations of size 0.02 on the transformed scale will have a small but non-negligible effect. We fix `a=0.5`, so that after 50 IF2 iterations, the perturbations are reduced to half their original magnitudes.

```
rw_sd = pp.RWSigma(
    sigmas={
        "Beta": 0.02,
        "mu_IR": 0.0,
        "N": 0.0,
        "rho": 0.02,
        "eta": 0.02,
        "k": 0.0
    },
    init_names=["eta"]
)
```

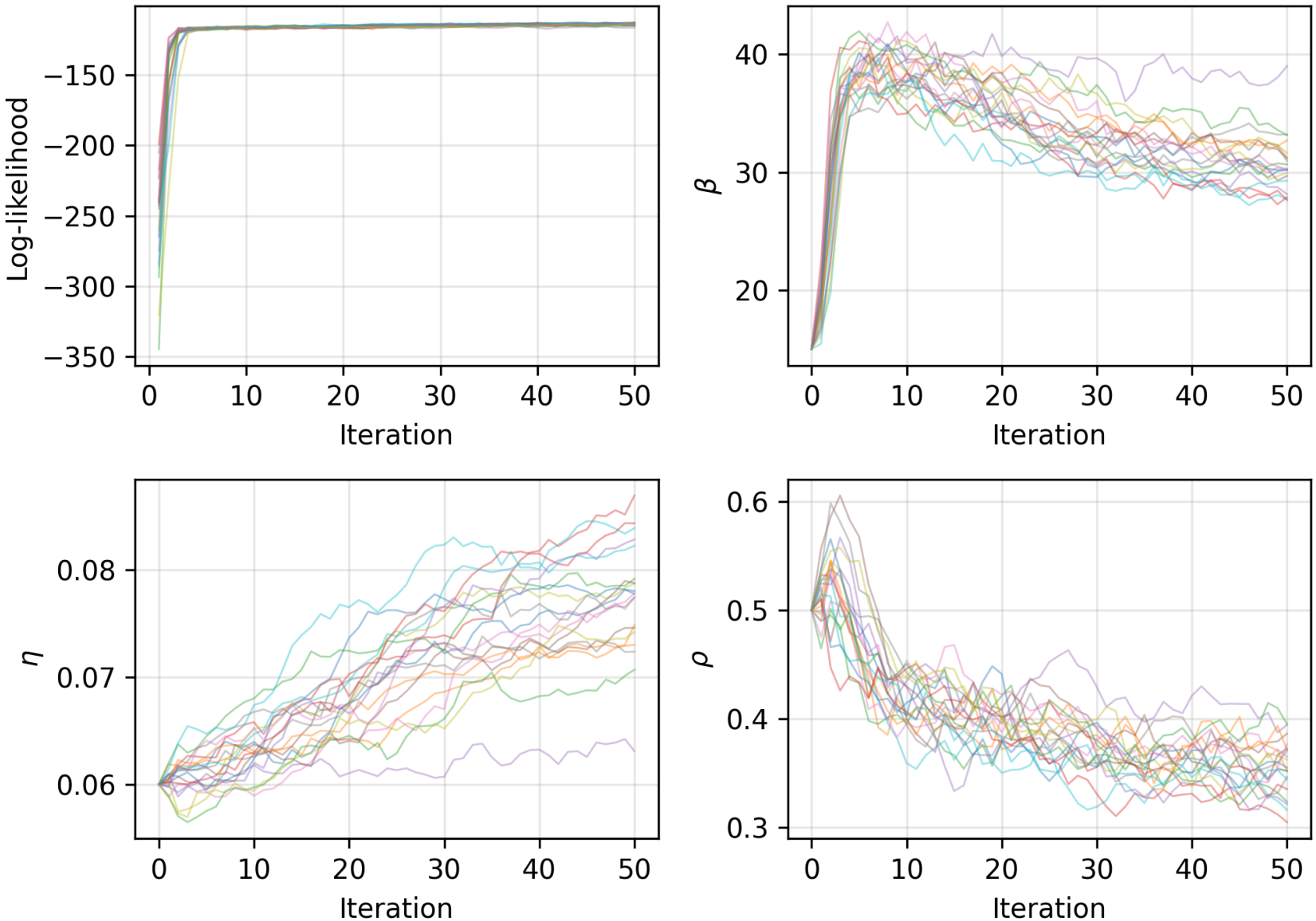
- The **initial value parameter** (IVP) η, designated by `init_names`, affects the dynamics only at time t_0, so it is perturbed only at time t_0.

A local search of the likelihood surface

```
cache_file = cache_dir + "/local-mif.pkl"
if os.path.exists(cache_file):
    with open(cache_file, 'rb') as f:
        mif_result = pickle.load(f)
else:
    # 20 parallel runs from same starting point
    n_local = 20
    theta_list = [theta.copy() for _ in range(n_local)]
    mod_local = pp.Pomp(
        rinit=rinit, rproc=rproc,
        dmeas=dmeas, rmeas=rmeas,
        ys=ys, theta=theta_list,
        statenames=statenames,
        par_trans=par_trans,
        t0=0.0, nstep=7,
        accumvars=("H",),
        ydim=1, covars=None
    )
    key = jax.random.key(482947940)
    mod_local.mif(
        J=2000, M=50,
        rw_sd=rw_sd, a=0.5, key=key
    )
    mif_result = mod_local.results_history.last()
    with open(cache_file, 'wb') as f:
        pickle.dump(mif_result, f)
```

Iterated filtering diagnostics

```
traces_da = mif_result.traces_da
```



- We see that the likelihood increases as the iterations proceed, though there is considerable variability due to (a) the poorness of our starting guess and (b) the stochastic nature of this Monte Carlo algorithm.
- We see movement in the parameters, though variability remains.

Estimating the likelihood

Although the filtering carried out by `mif` in the final filtering iteration generates an approximation to the likelihood at the resulting point estimate, this is not good enough for reliable inference.

- Partly, this is because parameter perturbations are applied in the last filtering iteration, so that the likelihood reported by `mif` is not identical to that of the model of interest.
- Partly, this is because `mif` is usually carried out with fewer particles than are needed for a good likelihood evaluation.

Therefore, we evaluate the likelihood, together with a standard error, using replicated particle filters at each point estimate.

```
Best log-likelihood: -112.2
(SE: 0.0)
```

A global search of the likelihood surface

- When carrying out parameter estimation for dynamic systems, we need to specify beginning values for both the dynamic system (in the state space) and the parameters (in the parameter space).
- To avoid confusion, we use the term "initial values" to refer to the state of the system at t_0 and "starting values" to refer to the point in parameter space at which a search is initialized.
- Practical parameter estimation involves trying many starting values for the parameters.
- One way to approach this is to choose a large box in parameter space that contains all remotely sensible parameter vectors.
- If an estimation method gives stable conclusions with starting values drawn randomly from this box, this gives some confidence that an adequate global search has been carried out.

For our measles model, a box containing reasonable parameter values might be $\beta \in (5, 80)$, $\rho \in (0.2, 0.9)$, $\eta \in (0, 1)$.

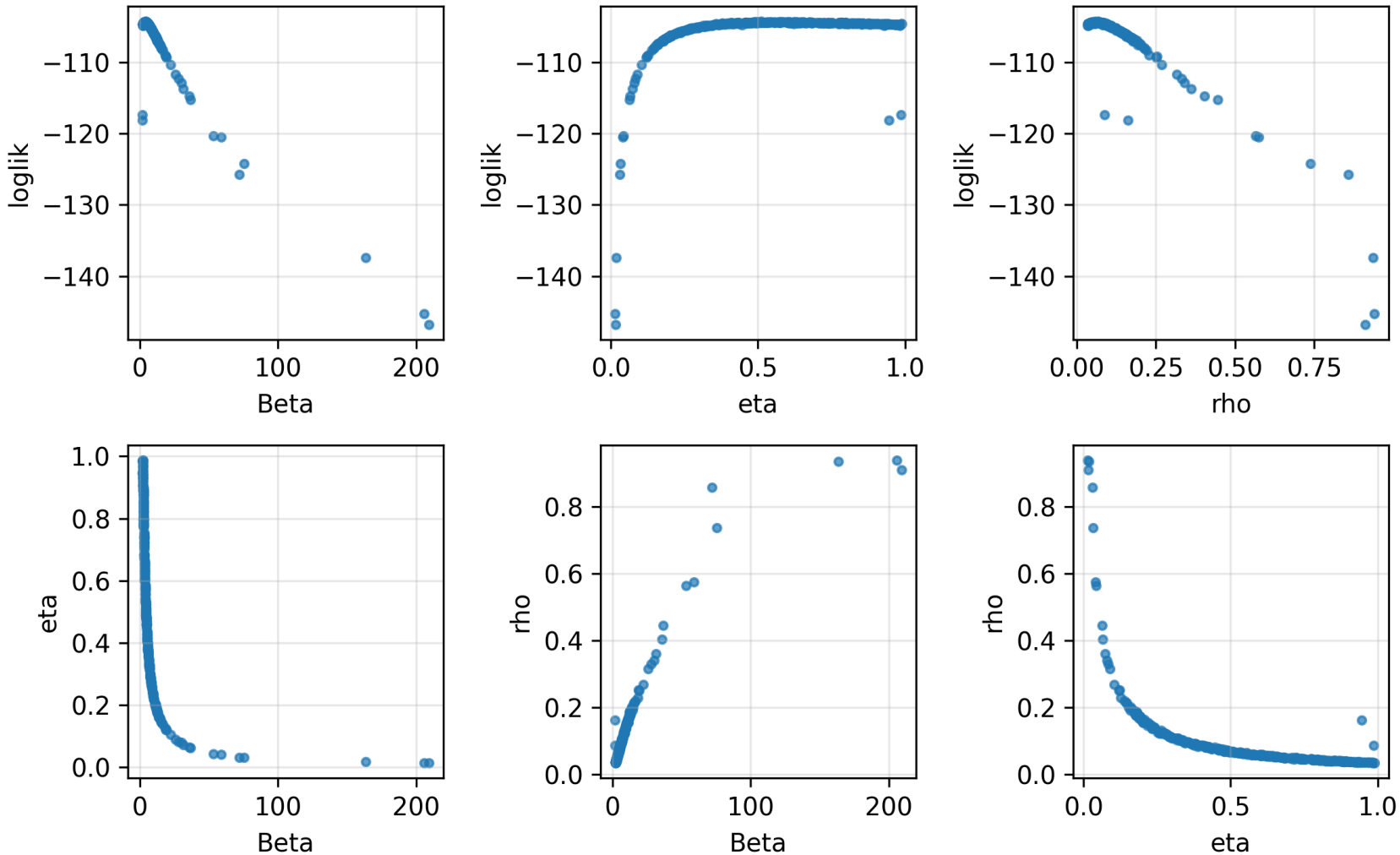
In `pypomp`, passing a list of parameter dicts to `Pomp` causes `mif` and `pfilter` to run all replicates **in parallel on the GPU**.

The full source code is online. In summary,

1. Cache to re-run only when results are absent.
2. Set up a starting box of parameter values.
3. Make multiple searches from each starting value.
4. Make multiple log-likelihood evaluations at each ending value.
5. Save all searches as `results_df`. The best is the candidate MLE.

```
Best log-likelihood: -104.3
```

Visualizing the global search

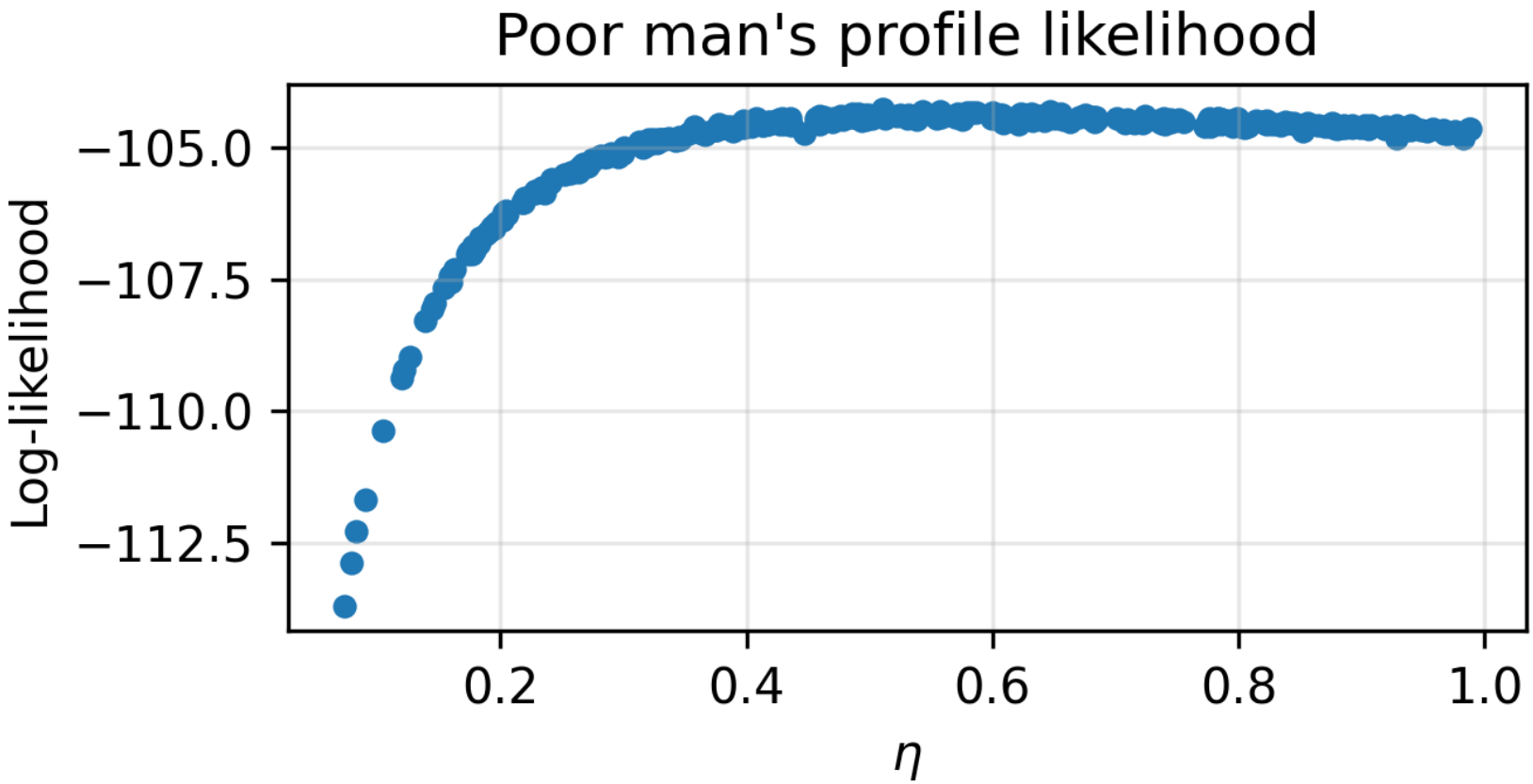


- We see that optimization attempts from diverse remote starting points converge on a particular region in parameter space.
- The estimates have comparable likelihoods, despite their considerable variability.
- This gives us some confidence in our maximization procedure.

The scatter plot of estimates for a specific parameter against their log-likelihood is a **poor man's profile**.

- All searches obtain log-likelihoods below the true profile log-likelihood function. Why?

The envelope containing all search results approximates the profile.

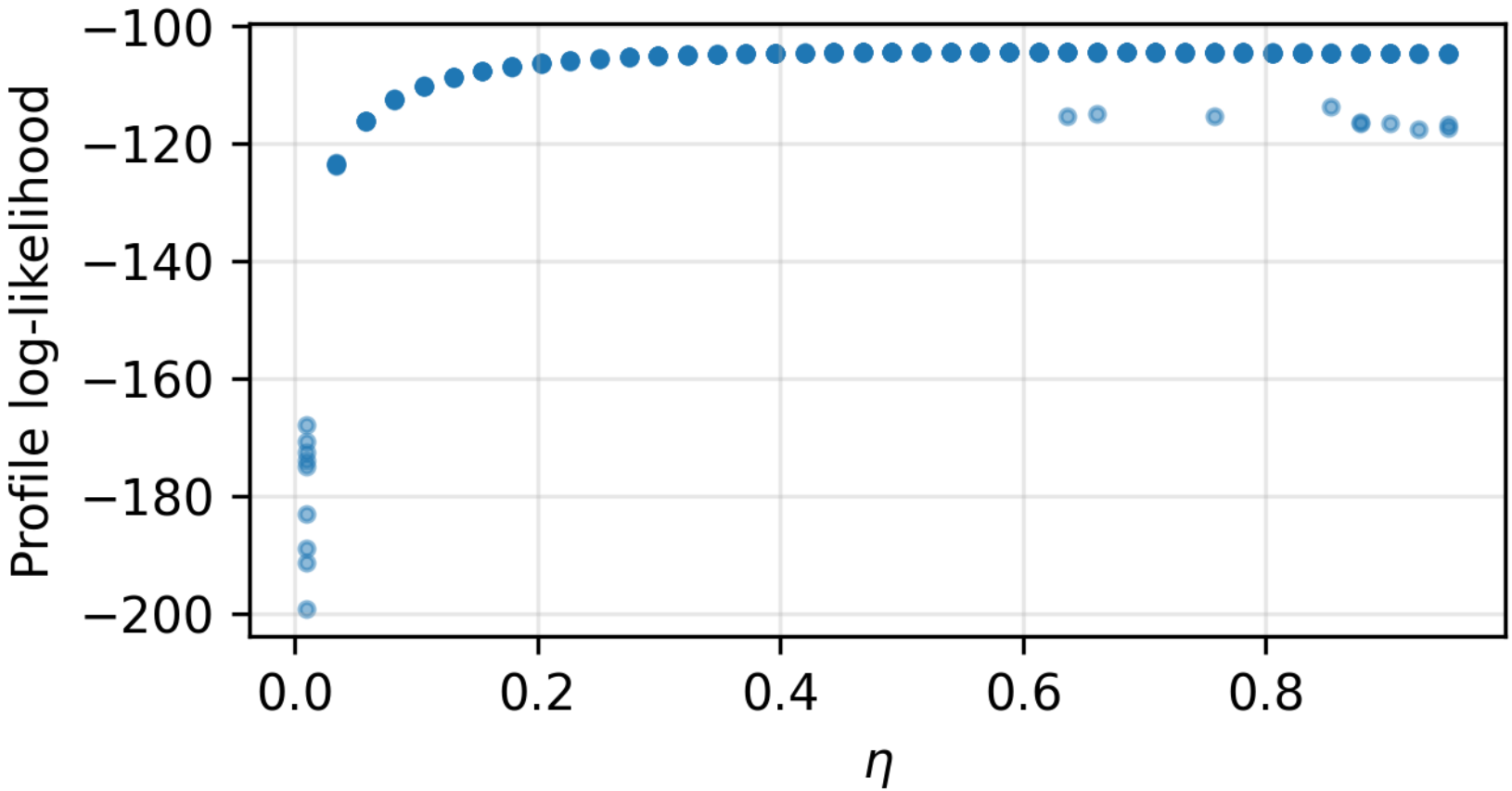


Profile likelihood over η

- The curvature displayed in the upper envelope of the above plot suggests that there is indeed information in the data with respect to the susceptible fraction, η.
- To solidify this evidence, let's compute a profile likelihood over this parameter.
- Recall that this means determining, for each value of η, the best likelihood that the model can achieve.
- To do this, we'll first bound the uncertainty by putting a box around the highest-likelihood estimates we've found so far.
- Within this box, we'll choose some random starting points, for each of several values of η.
- For each grid point, we launch multiple starts **in parallel**. We keep η fixed by setting its random walk standard deviation to zero.

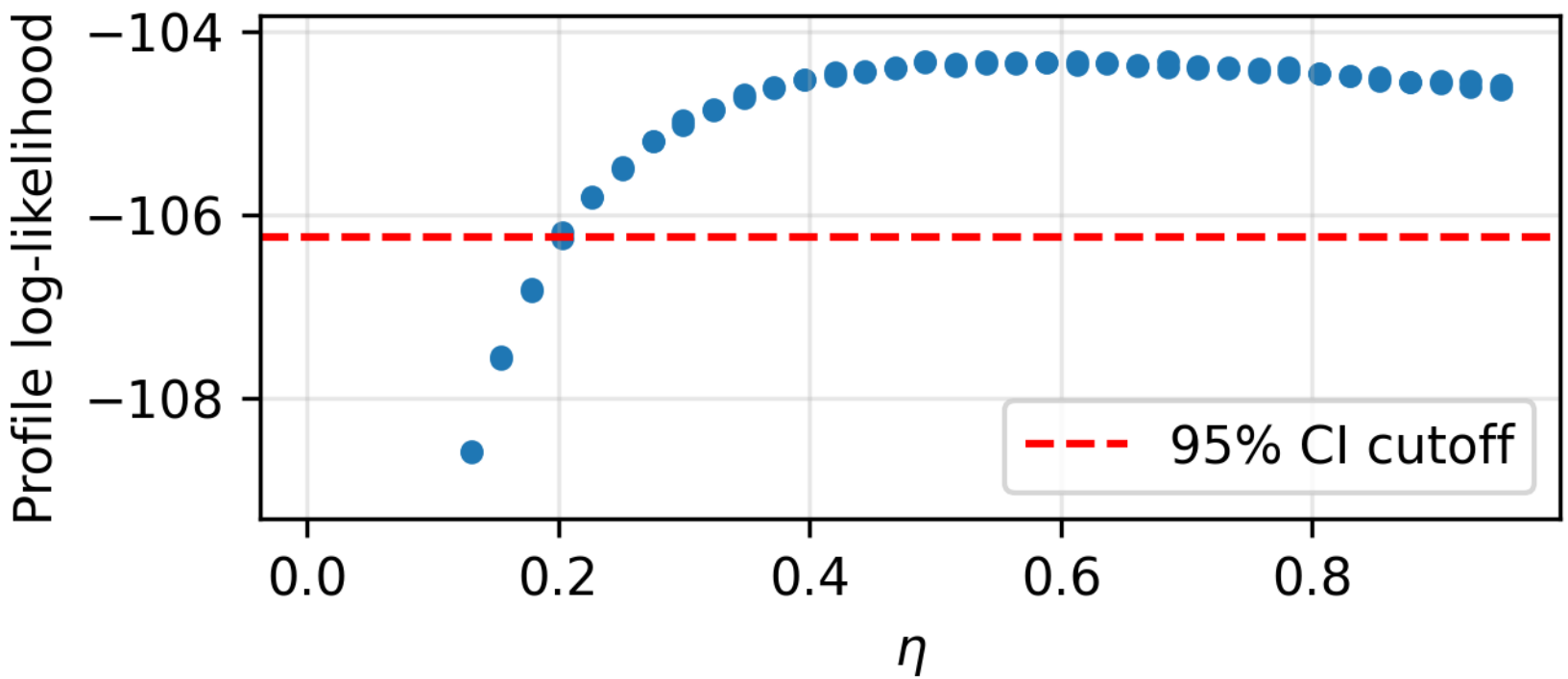
Visualizing profile likelihood

Plotting just the results of the profile calculation reveals that, while some of the IF2 runs either become "stuck" on local minima or run out of opportunity to reach the heights of the likelihood surface, many of the runs converge on high likelihoods.



Focusing on just the top of the surface shows that, in fact, one is able to estimate η using these data. In the following plot, the cutoff for the 95% confidence interval (CI) is shown.

```
max_ll = profile_eta_df['loglik'].max()
ci_cutoff = max_ll - 0.5 * chi2.ppf(0.95, df=1)

top = (profile_eta_df
    .groupby(profile_eta_df['eta'].round(5))
    .apply(lambda g: g.nlargest(2, 'loglik'),
           include_groups=False)
    .reset_index(drop=True))
```

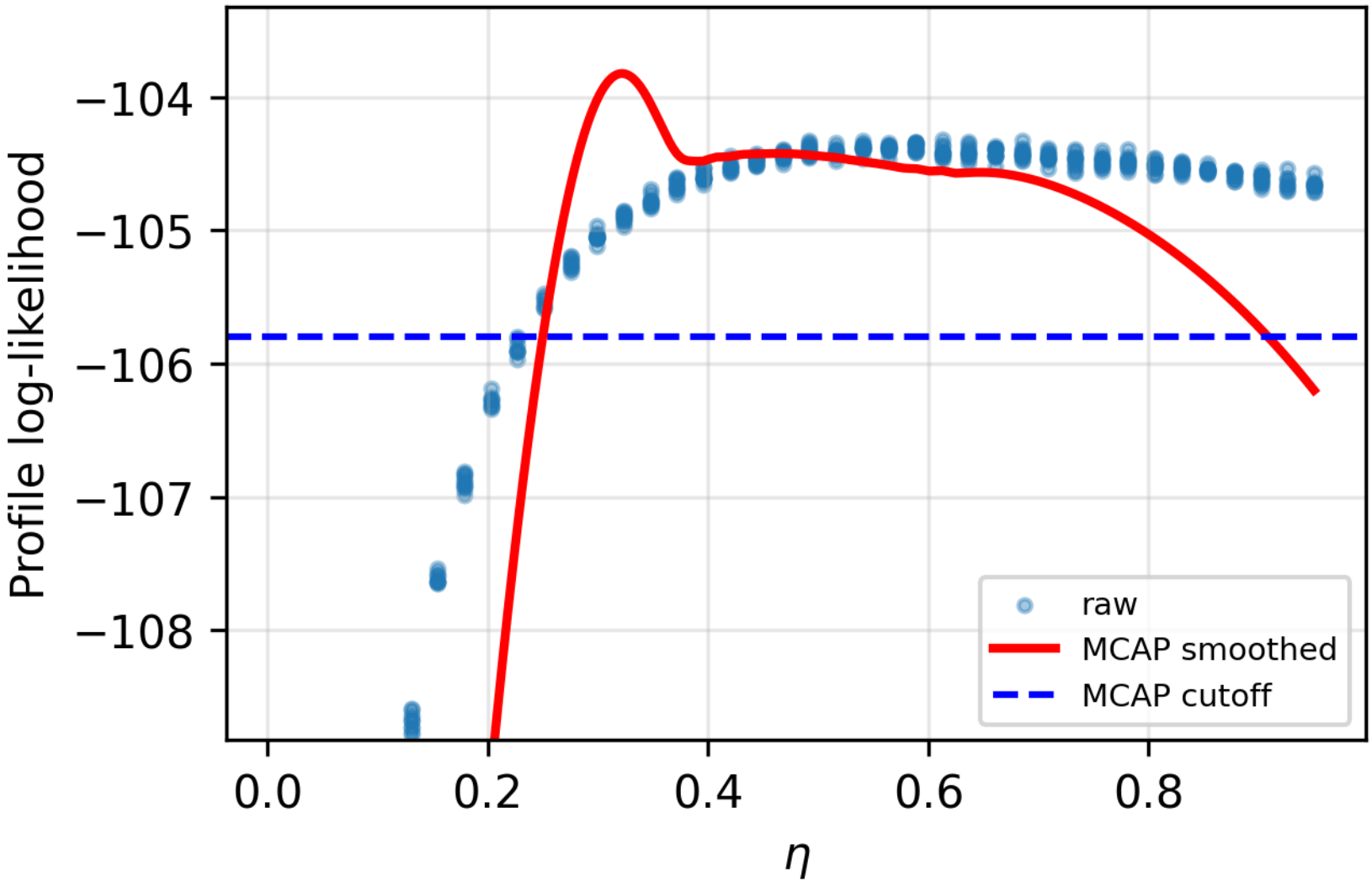


Monte Carlo Adjusted Profile (MCAP)

The mcap function in pypomp constructs a profile likelihood confidence interval that accounts for Monte Carlo error:

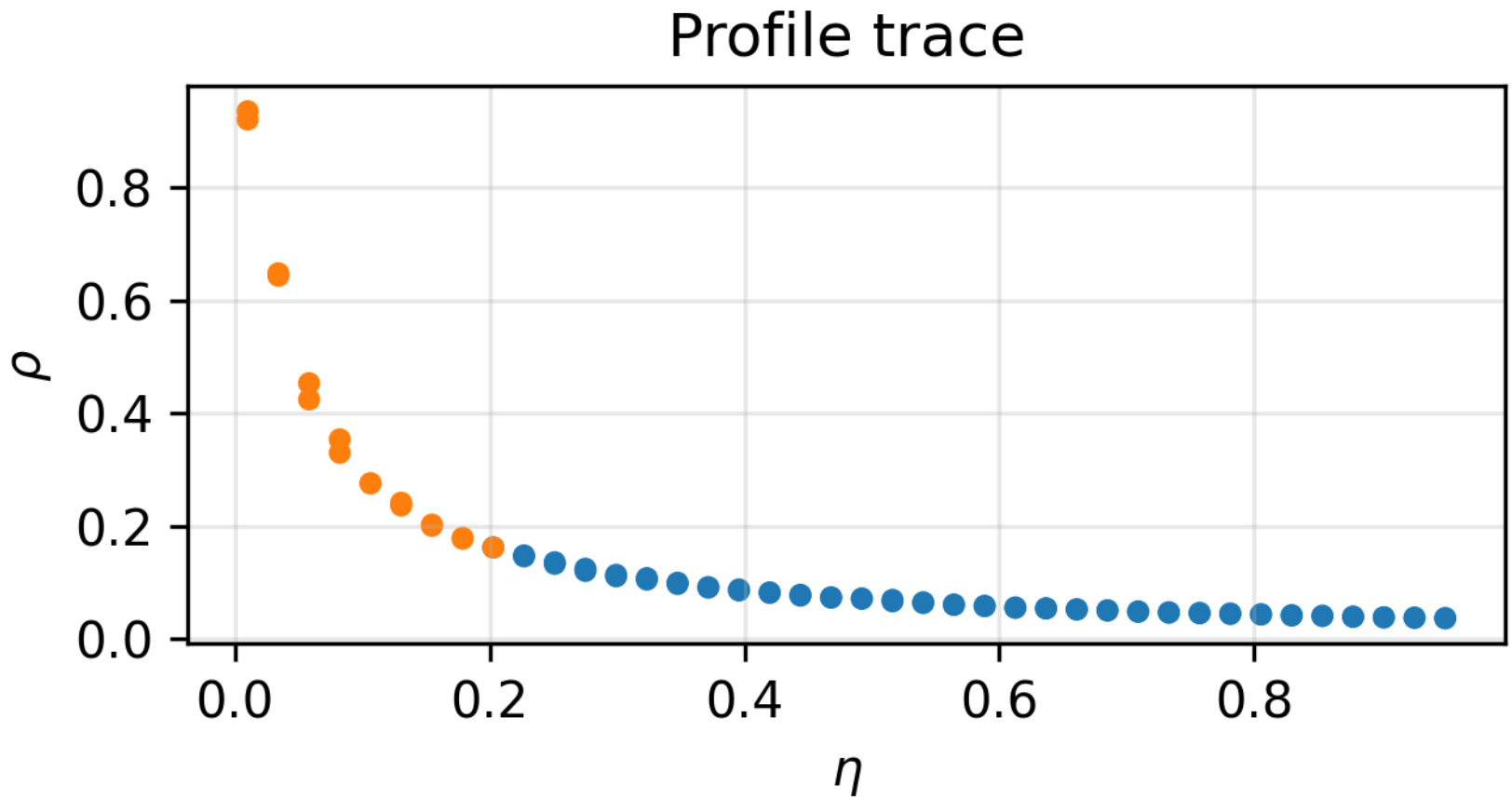
```
mcap_result = pp.mcap(
    parameter=profile_eta_df['eta'].values,
    loglik=profile_eta_df['loglik'].values,
    level=0.95, span=0.75)

print(f"MLE of eta: {mcap_result.mle:.4f}")
print(f"95% CI: ({mcap_result.ci[0]:.4f},"
      f" {mcap_result.ci[1]:.4f})")
```

```
MLE of eta: 0.3215
95% CI: (0.2499, 0.9077)
```


Profile log-likelihood
−104
−105
−106
−107
−108
0.0
0.2
0.4
0.6
0.8
η
raw
MCAP smoothed
MCAP cutoff

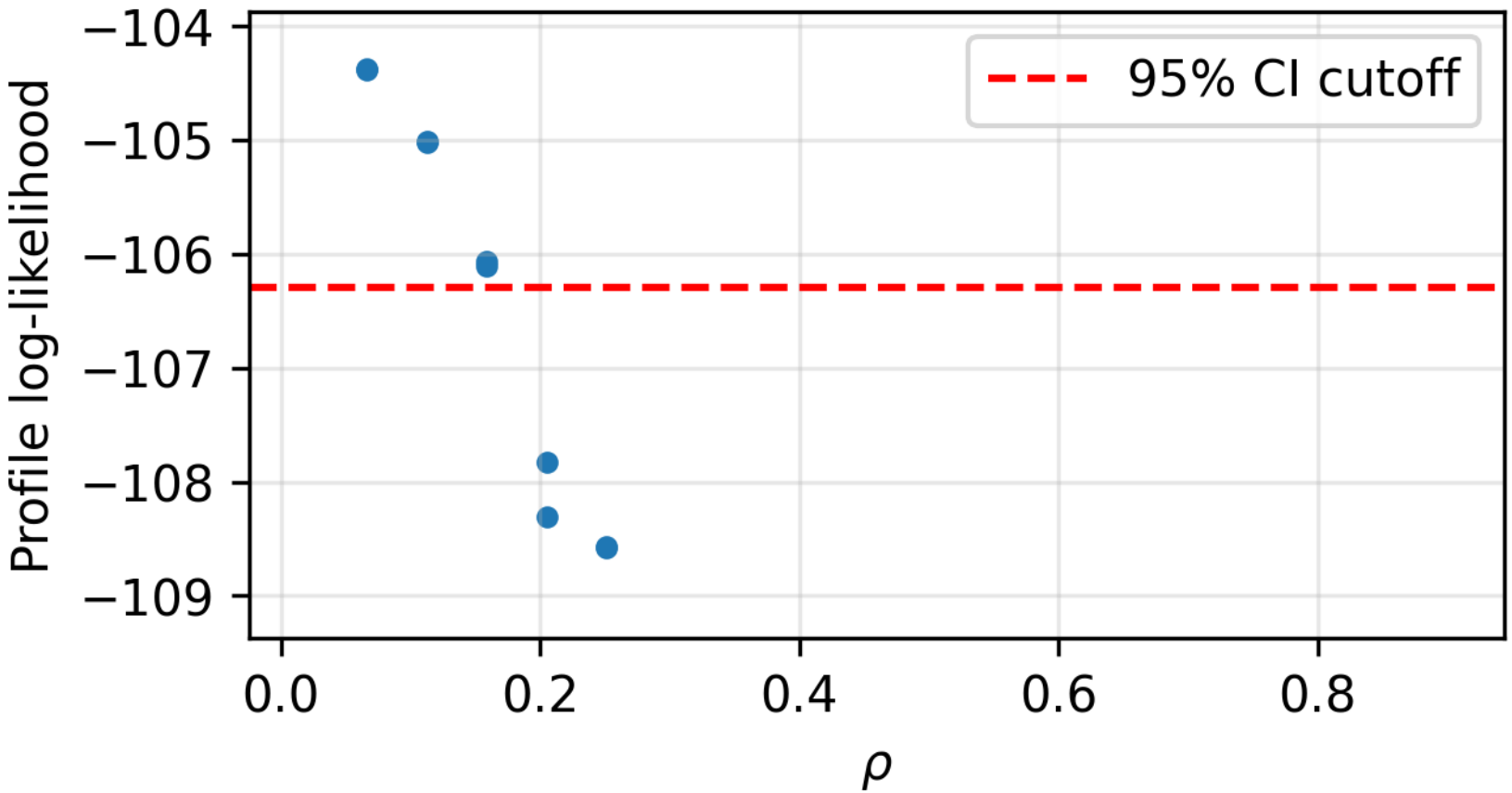
Profile trace

As one varies η across the profile, the model compensates by adjusting the other parameters. It can be very instructive to understand how the model does this. For example, how does the reporting efficiency, ρ, change as η is varied? We can plot ρ vs η across the profile. This is called a **profile trace**.



Profile over ρ

While the profile trace above is suggestive about the range of ρ, to confirm this we should construct a proper profile likelihood over ρ. We initialize the IF2 computations at points we have already established have high likelihoods.



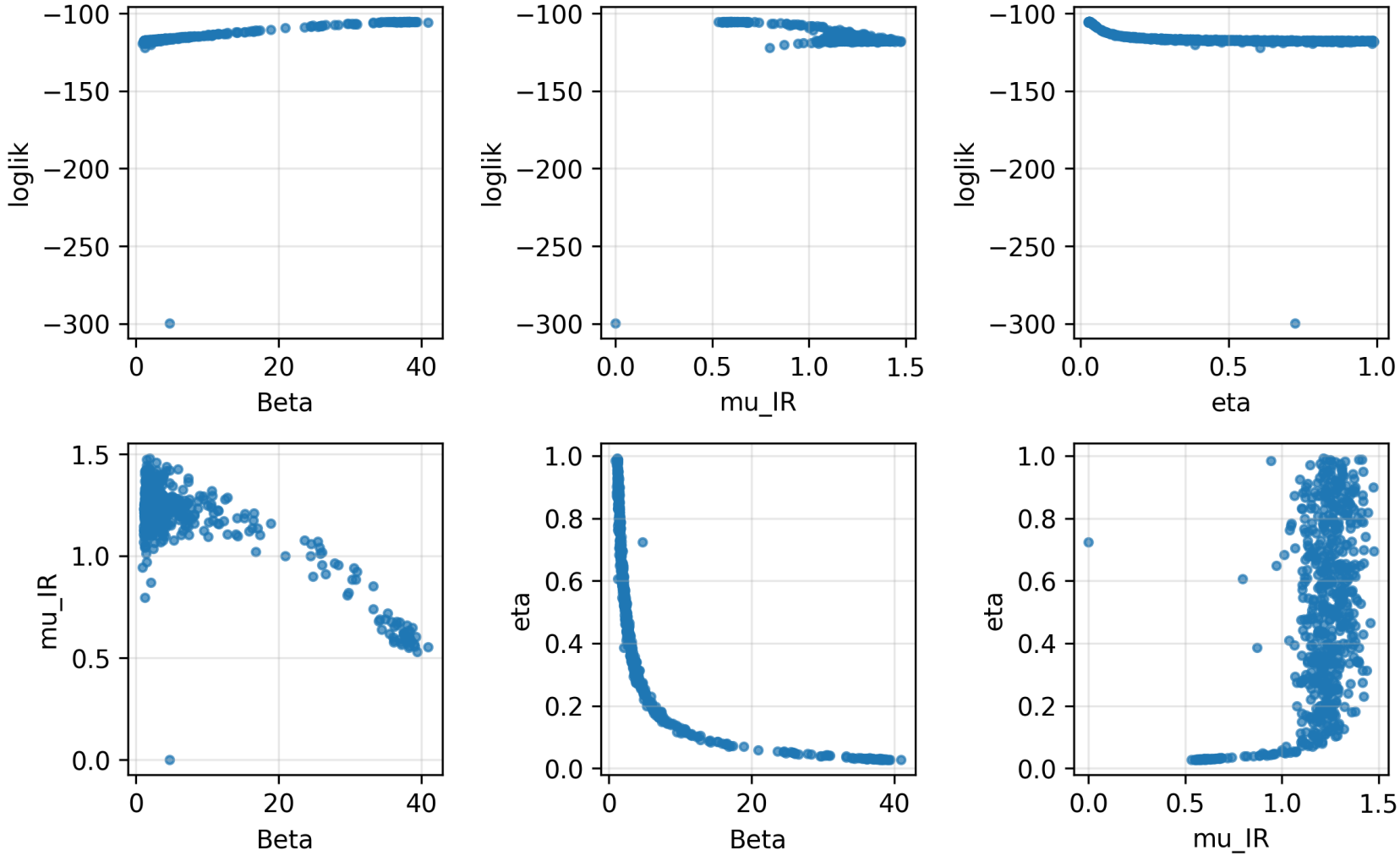
Parameter estimates as model predictions

- The estimated parameters are one kind of model prediction. When we can estimate parameters using other data, we can test these predictions.
- In the case of a highly contagious, immunizing childhood infection such as measles, we can obtain an estimate of the reporting efficiency, ρ, by simply regressing cumulative cases on cumulative births over many years.
- When we do this for Consett, we see that the reporting efficiency is roughly 60%.
- Since such a value makes the outbreak data quite unlikely, the prediction does not appear to be borne out.
- We can conclude that one or more of our model assumptions is inconsistent with the data.

Searching in another direction

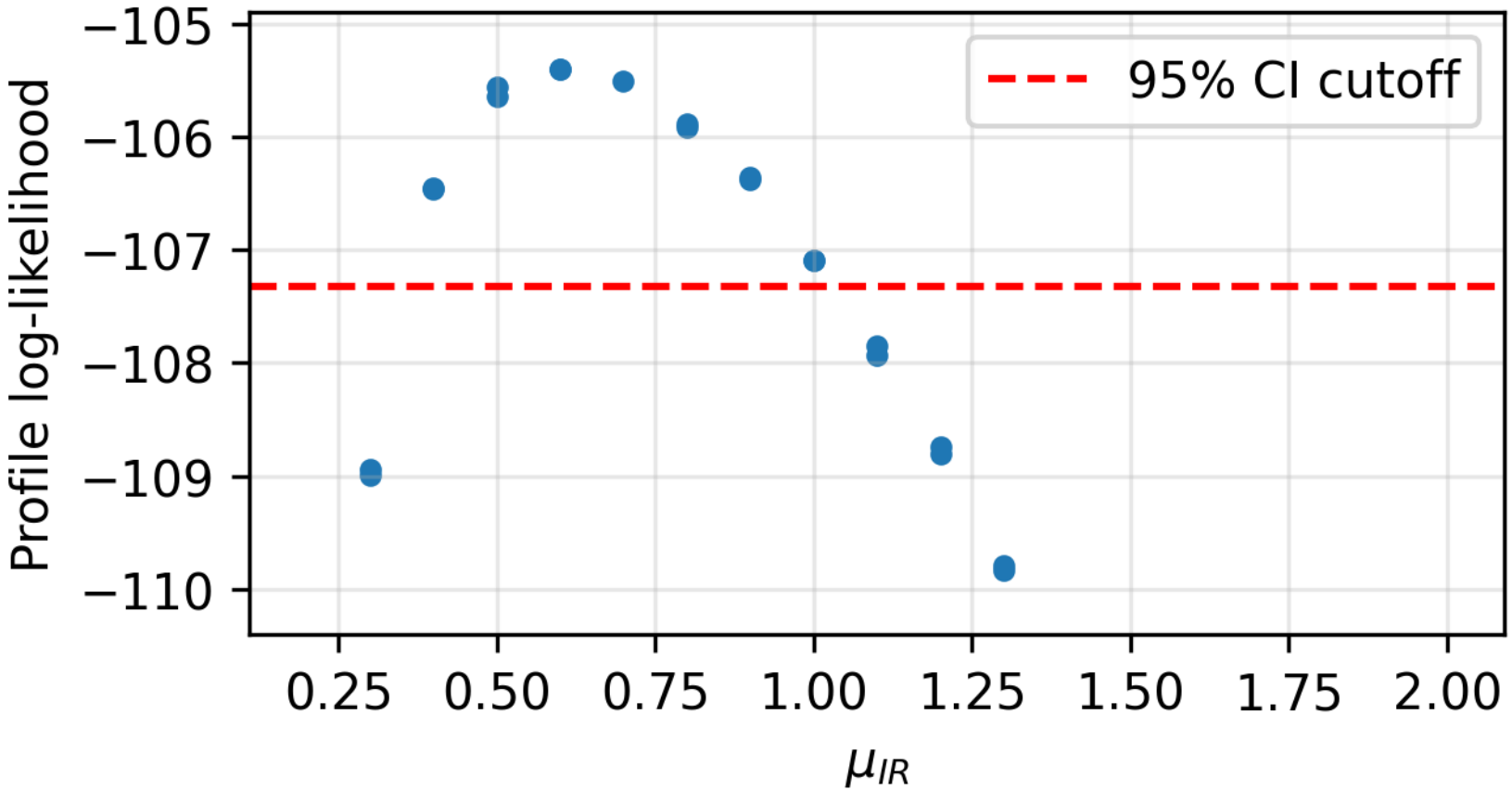
- Let's revisit our assumption about the infectious period. Indeed, it would not be surprising were we to find that the *effective* infectious period, at the population scale, were somewhat shorter than the *clinical* infectious period.
- For example, confinement of patients should reduce contact rates, and might therefore curtail the effective infectious period.
- To investigate this, we'll relax our assumption about the value of μ_{IR}.
- We will estimate the model under the assumption that $\rho = 0.6$, but without making assumptions about the duration of the infectious period.

Best log-likelihood: -105.4



Profile over μ_{IR} (with $\rho = 0.6$ fixed)

To make inferences about μ_{IR}, we compute a profile likelihood with $\rho = 0.6$ fixed.



- This suggests that $\rho = 0.6$ is consistent only with smaller values of μ_{IR}, and hence *longer* infectious periods than are possible if the duration of shedding is actually less than one week.
- Thus the model is incapable of reconciling both an infectious period of less than one week and a reporting rate of 60%.
- *What structural changes to the model might we make to improve its ability to explain the data?*

Exercises

Exercise 4.1. *Fitting the SEIR model.* In this exercise, you will estimate the parameters and likelihood of the SEIR model you implemented in the earlier lessons by following the template above. Purely for the sake of simplicity, you may assume that the values of μ_{IR} and k are known. To do this efficiently, we will make use of a system of *run-levels*. At each run-level, we will select some number of particles (J), number of IF2 iterations (M), and number of starting guesses, to achieve a particular result.

(A) First, conduct a local search and compute the likelihood at the end of each `mif` run. Track the time used and compute the amount of time used per IF2 iteration per 1000 particles.

(B) At run-level 1, we want a quick calculation that verifies that the codes are working as expected. Choose a number of IF2 iterations so that you can do a very crude "global search" that will complete in two or three minutes.

(C) At run-level 2, we want a computation that gives us some results we can begin to interpret, but that is still as quick as possible.

(D) Run-level 3 is intended for final or near-final results.

(E) How does the SEIR model compare with the SIR model?

Exercise 4.2. *Fitting all parameters of the SIR model.* In all of the foregoing, we have assumed a fixed value of the dispersion parameter, k, of the negative binomial measurement model. We've also fixed one or the other of μ_{IR}, η. Now attempt to estimate all the parameters simultaneously. To accomplish this, use the same system of run-levels as in the previous Exercise. How much is the fit improved? How has the model's explanation of the data changed?

Exercise 4.3. *Construct a profile likelihood.* How strong is the evidence about the contact rate, β, given this model and data? Use `mif` to construct a profile likelihood. Due to time constraints, you may be able to compute only a preliminary version.

It is also possible to profile over the basic reproduction number, $R_0 = \beta/\mu_{IR}$. Is this more or less well determined than β for this model and data?

Exercise 4.4. *Checking the source code.* Does the code implement the model described? It can be surprisingly hard to make sure that the written equations and the code are perfectly matched. Papers should be written to be readable, and therefore people rarely choose to clutter papers with numerical details which they hope and believe are scientifically irrelevant. (a) What problems can arise due to the conflict between readability and reproducibility? (b) What solutions are available?

Exercise 4.5. *Beware errors in `rproc`.* Suppose that there is an error in the coding of `rproc` and suppose that plug-and-play statistical methodology is used to infer parameters. As a conscientious researcher, you carry out a simulation study to check the soundness of your inference methodology on this model. To do this, you use `simulate` to generate realizations from the fitted model and you check that your parameter inference procedure recovers the known parameters, up to some statistical error. (a) Will this procedure help to identify the error in `rproc`? (b) If not, how might you debug `rproc`? (c) What research practices help minimize the risk of errors in simulation code?

Exercise 4.6. *Choosing the algorithmic settings for IF2.* Experiment with the algorithmic settings (`J`, `M`, `rw_sd`, `a`) on the Consett measles example. How sensitive are the results to these choices?

Summary

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. Parameter transformations via `ParTrans(to_est, from_est)` ensure the random walk respects constraints (e.g., `log` for positive parameters, `logit` for probabilities).

4. `pypomp` API for IF2: `mif(J, M, rw_sd, a, key)` runs IF2, `RWSigma(sigmas={...}, init_names=[...])` specifies perturbations, `results_history.last().traces_da` gets parameter traces.

5. Passing a **list of parameter dicts** to `Pomp` runs all replicates in parallel on the GPU.

6. Always evaluate likelihood at IF2 endpoints with `pfilter`. Use `mcap` for Monte Carlo-adjusted profile confidence intervals.

License and acknowledgments

- This lesson is prepared for the Simulation-based Inference for Epidemiological Dynamics module.
- The materials build on previous versions of this course and related courses.
-

References I

Ionides, E. L., Nguyen, D., Atchadé, Y., Stoev, S., and King, A. A. (2015). Inference for dynamic and latent variable models via iterated, perturbed Bayes maps. *Proc Natl Acad Sci*, 112(3):719–724.