Chapter 12. Practical likelihood-based inference for POMP models

Objectives

1. Understanding the simplest **particle filter** and how it enables Monte Carlo solution of the POMP filtering and prediction recursions and computation of a Monte Carlo evaluation of the likelihood.
2. Using the particle filter to visualize and exploring likelihood surfaces
3. Understanding how iterated filtering algorithms carry out repeated particle filtering operations, with randomly perturbed parameter values, in order to maximize the likelihood.
4. Carrying out likelihood-based inferences for dynamic models using simulation-based statistical methodology in the R package `pomp`, demonstrated by fitting an SIR model to a boarding school flu outbreak.

Indirect specification of the statistical model via a simulation procedure

- For simple statistical models, we may describe the model by explicitly writing the density function $f_{Y_{1:N}}(y_{1:N}\,;\theta)$. One may then ask how to simulate a random variable $Y_{1:N} \sim f_{Y_{1:N}}(y_{1:N}\,;\theta)$.
- For many dynamic models it is convenient to define the model via a procedure to simulate the random variable $Y_{1:N}$. This implicitly defines the corresponding density $f_{Y_{1:N}}(y_{1:N}\,;\theta)$. For a complicated simulation procedure, it may be difficult or impossible to write down $f_{Y_{1:N}}(y_{1:N}\,;\theta)$ exactly.
- It is important for us to bear in mind that the likelihood function exists even when we don't know what it is! We can still talk about the likelihood function, and develop numerical methods that take advantage of its statistical properties.

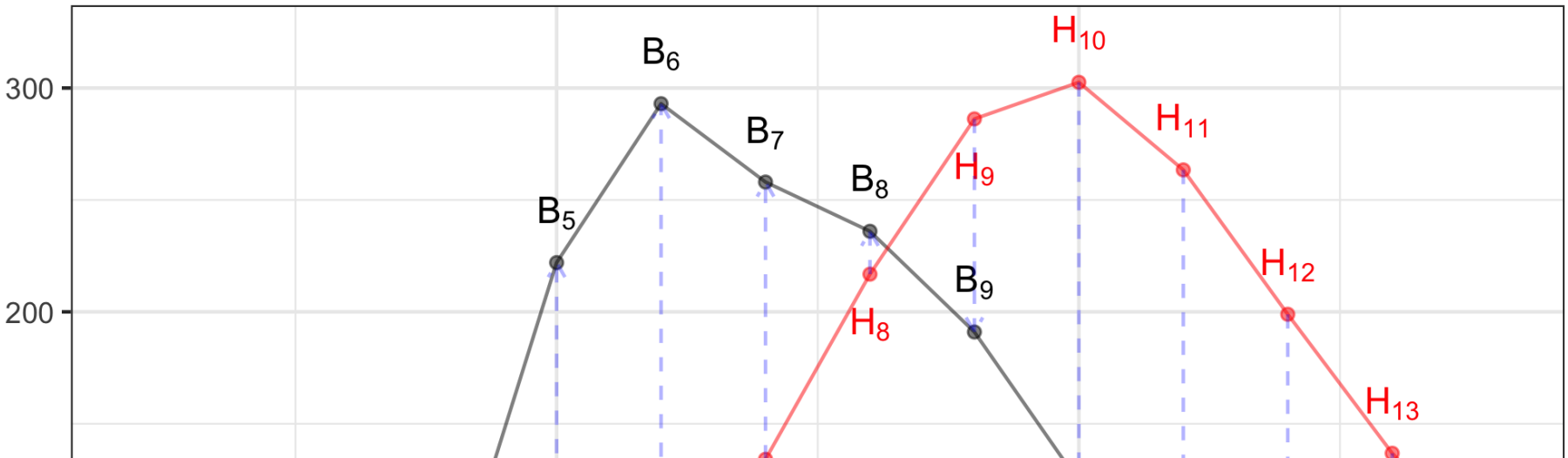
Special case: a deterministic unobserved state process

- Suppose $X_n = x_n(\theta)$ is a known function of θ for each n. What is the likelihood?
- Since the distribution of the observable random variable, Y_n, depends only on X_n and θ, and since, in particular Y_m and Y_n are independent given X_m and X_n, we have

$$\mathcal{L}(\theta) = \prod_n f_{Y_n|X_n}(y_n^* \,|\, x_n(\theta)\,;\theta)$$

or

$$\ell(\theta) = \log \mathcal{L}(\theta) = \sum_n \log f_{Y_n|X_n}(y_n^* \,|\, x_n(\theta)\,;\theta).$$



An ineffective method: Likelihood by direct simulation

To motivate the particle filter, we first introduce a simpler method which usually does not work on anything but very short time series. We calculate:

$$\begin{aligned}
\mathcal{L}(\theta) &= f_{Y_{1:N}}(y_{1:N}\,;\theta) \\
&= \int_{x_{0:N}} f_{X_0}(x_0\,;\theta) \prod_{n=1}^{N} f_{Y_n|X_n}(y_n \mid x_n\,;\theta)\, f_{X_n|X_{n-1}}(x_n|x_{n-1}\,;\theta)\, dx_{0:N} \\
&= \mathbb{E}\left[\prod_{n=1}^{N} f_{Y_n|X_n}(y_n \mid X_n\,;\theta)\right] \\
&\approx \frac{1}{J}\sum_{j=1}^{J}\prod_{n=1}^{N} f_{Y_n|X_n}(y_n \mid X_{n,j}\,;\theta)
\end{aligned}$$

where we have J independent simulated trajectories $\{X_{nj}, n = 1, \ldots, N\}$, and $\approx$ is justified by the laws of large numbers.

Question 12.1. Why is this approach ineffective if the time series is not very short?

The particle filter

- Fortunately, we can compute the likelihood for a POMP model much more efficiently by using Monte Carlo representations of the prediction and filtering recursions.
- This gives the **particle filter** algorithm, also known as sequential Monte Carlo (SMC):

1. Suppose $X^F_{n-1,j}$, $j = 1, \ldots, J$ is a set of J points drawn from the filtering distribution at time $n-1$.
2. We obtain a sample $X^P_{n,j}$ of points drawn from the prediction distribution at time t by simply simulating the process model:

$$X^P_{n,j} \sim \text{process}(X^F_{n-1,j}, \theta), \qquad j = 1, \ldots, J.$$

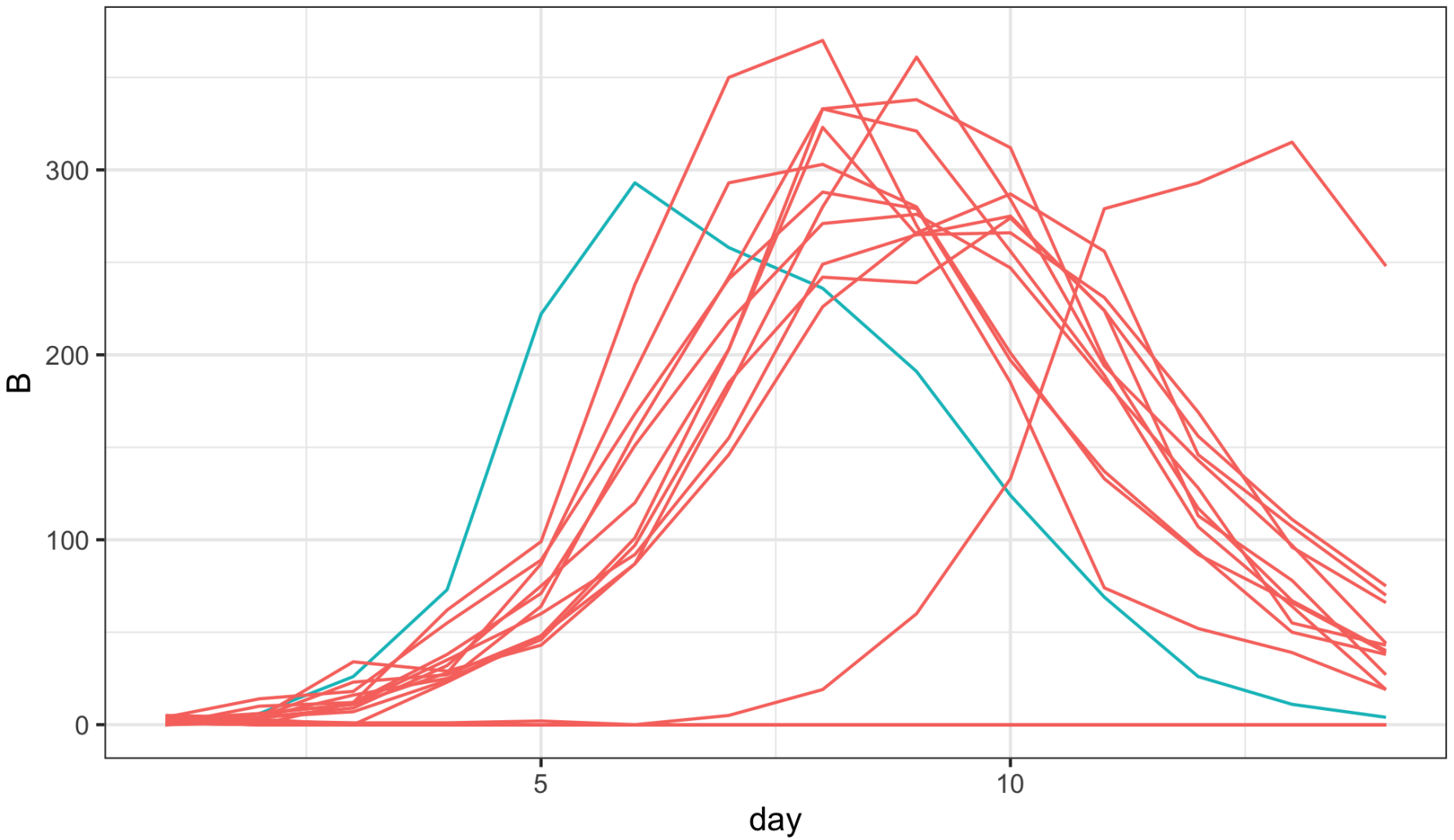
3. Having obtained $x^P_{n,j}$, we obtain a sample of points from the filtering distribution at time t_n by *resampling* from $\left\{X^P_{n,j}, j \in 1:J\right\}$ with weights

$$w_{n,j} = f_{Y_n|X_n}(y^*_n|X^P_{n,j}\,;\theta).$$

- The Monte Carlo principle tells us that the conditional likelihood

$$\mathcal{L}_n(\theta) \quad = \quad f_{Y_n|Y_{1:n-1}}(y^*_n|y^*_{1:n-1}\,;\theta)$$

```
sims <- simulate(sir,params=c(Beta=1.8,mu_IR=1,rho=0.9,N=2600),
  nsim=20,format="data.frame",include=TRUE)
ggplot(sims,mapping=aes(x=day,y=B,group=.id,color=.id=="data"))+
  geom_line()+guides(color=FALSE)
```



Sequential Monte Carlo in pomp

- Here, we'll get some practical experience with the particle filter, and the likelihood function, in the context of our influenza outbreak case study.
- In pomp, the basic particle filter is implemented in the command `pfilter`. We must choose the number of particles to use by setting the `Np` argument.

```
pf <- pfilter(sir,Np=5000,params=c(Beta=2,mu_IR=1,rho=0.8,N=2600))
logLik(pf)

## [1] -74.38111
```

- We can run a few particle filters to get an estimate of the Monte Carlo variability:

```
pf <- replicate(10,pfilter(sir,Np=5000,params=c(Beta=2,mu_IR=1,rho=
ll <- sapply(pf,logLik); ll

##  [1] -70.14517 -76.58557 -72.34135 -80.62329 -75.07345
##  [6] -74.11358 -76.50689 -76.52644 -75.68941 -69.65674
```

Some useful theoretical results for the particle filter

- A theoretical property of the particle filter is that it gives us an unbiased Monte Carlo estimate of the likelihood.
- This theoretical property, combined with Jensen's inequality and the observation that $\log(x)$ is a concave function, ensures that the average of the log likelihoods from many particle filter replications will have negative bias as a Monte Carlo estimator of the log likelihood.
- We've been careful to avoid this bias in the code above, by using `logmeanexp` to average the likelihood estimates on the natural scale not the logarithmic scale.

The graph of the likelihood function: The likelihood surface

- Intuitively, it can be helpful to think of the geometric surface defined by the likelihood function.
- If Θ is two-dimensional, then the surface $\ell(\theta)$ has features like a landscape: local maxima of $\ell(\theta)$ are peaks, local minima are valleys, peaks may be separated by a valley or may be joined by a ridge.
- Moving along a ridge, you may be able to go from one peak to the other without losing much elevation. Narrow ridges can be easy to fall off, and hard to get back on to.
- In higher dimensions, one can still think of peaks and valleys and ridges. However, as the dimension increases it quickly becomes hard to imagine the surface.
- To get an idea of what the likelihood surface looks like in the neighborhood of the default parameter set supplied by `sir`, we can construct a **likelihood slice**.
- We'll make slices in the β and μ_{IR} directions. Both slices will pass through the default parameter set.

```
sliceDesign(
  c(Beta=2,mu_IR=1,rho=0.8,N=2600),
  Beta=rep(seq(from=0.5,to=4,length=40),each=3),
  mu_IR=rep(seq(from=0.5,to=2,length=40),each=3)) -> p

# library(foreach)
library(doParallel)
registerDoParallel()

library(doRNG)
registerDoRNG(3899882)

foreach (theta=iter(p,"row"),
  .combine=rbind,.inorder=FALSE) %dopar% {
    pfilter(sir,params=unlist(theta),Np=5000) -> pf
    theta$loglik <- logLik(pf)
    theta
  } -> p
```

- The code above serves as a definition of what is a likelihood slice.

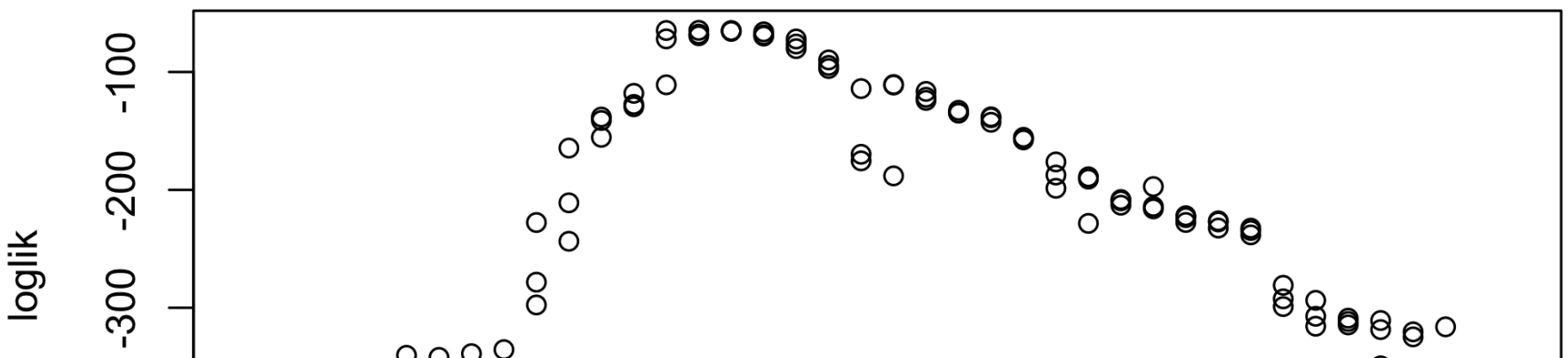
Question 12.2. Write down the definition of a likelihood slice in mathematical notation.

Question 12.3. Explain the difference between a likelihood slice and a likelihood profile,
(a) from a computational perspective.

(b) from the perspective of constructing confidence intervals and hypothesis tests.

- Note that the slice computation used the **foreach** package with the multicore backend (**doParallel**) to parallelize these computations.
- To ensure that we have high-quality random numbers in each parallel R session, we use a parallel random number generator provided by the `doRNG` package.
- Now, we can plot the likelihood slices:

```
foreach (v=c("Beta","mu_IR")) %do%
{
  x <- subset(p,slice==v)
  plot(x[[v]],x$loglik,xlab=v,ylab="loglik")
}
```



Slices offer a very limited perspective on the geometry of the likelihood surface. With two parameters, we can evaluate the likelihood at a grid of points and visualize the surface directly.

```
expand.grid(Beta=seq(from=1,to=4,length=50),
            mu_IR=seq(from=0.7,to=3,length=50),
            rho=0.8,
            N=2600) -> p

foreach (theta=iter(p,"row"),.combine=rbind,
         .inorder=FALSE) %dopar%
 {
   pfilter(sir,params=unlist(theta),Np=5000) -> pf
   theta$loglik <- logLik(pf)
   theta
 } -> p
```

```
pp <- mutate(p,loglik=ifelse(loglik>max(loglik)-100,loglik,NA))
ggplot(data=pp,mapping=aes(x=Beta,y=mu_IR,z=loglik,fill=loglik))+
  geom_tile(color=NA)+
  geom_contour(color='black',binwidth=3)+
  scale_fill_gradient()+
```

Likelihood maximization for the boarding school flu example

- We saw above that the default parameter set for the 'bsflu' pomp object is not particularly close to the MLE.
- One way to find the MLE is to try optimizing the estimated likelihood, computed by the particle filter, directly.
- There are many optimization algorithms to choose from, and many implemented in R.
- Three issues arise immediately (discussed more on following slides):

1. The particle filter gives us a stochastic estimate of the likelihood.
2. Lack of derivatives.
3. Constrained parameters.

1. The particle filter gives us a stochastic estimate of the likelihood

- We can reduce this variability by making the number of particles, `Np`, larger. However, we cannot make it go away.
- We can use deterministic optimization, by fixing the seed of the pseudo-random number generator, a side effect is that the objective function can become jagged, with many small local maxima and minima.
- If we use stochastic optimization, the underlying surface may be smoother but we see it only with Monte Carlo noise.
- This is the trade-off between a noisy and a rough objective function.

2. Lack of derivatives

- Because the particle filter gives us just an estimate of the likelihood and no information about the derivative, we must choose an algorithm that is "derivative-free".
- There are many such, but we can expect less efficiency than would be possible with derivative information.
- Note that finite differencing (i.e., a direct numerical estimate of the derivative) is not an especially promising way of constructing derivatives in the presence of Monte Carlo noise.

3. Constrained parameters

- For the boarding school flu example, the parameters are constrained to be positive, and $\rho < 1$.
- Such constraints are common, especially for rate parameters.
- We must select an optimizer that can solve this **constrained maximization problem**, or figure out some of way of turning it into an unconstrained maximization problem.
- For the latter, we transform the parameters onto a scale on which there are no constraints.

Cautions about parameter estimation for dynamic models

- When we write down a mechanistic model for a system, we have some idea of what we intend parameters to mean. In epidemiology, for example, we interpret parameters as a reporting rate, a contact rate between individuals, an immigration rate, a duration of immunity, etc.
- The data and the parameter estimation procedure do not know about our intended interpretation of the model. It can and does happen that some parameter estimates statistically consistent with the data may be scientifically absurd according to the scientific reasoning that went into building the model.
- This can arise as a consequence of weak identifiability.
- It can also be a warning that the data do not agree that our model represents reality in the way we had hoped. Perhaps more work is needed on model development.
- Scientifically unreasonable parameter estimates can sometimes be avoided by fixing some parameters at known, reasonable values. However, this risks suppressing the warning that the data were trying to give about weaknesses in the model, or in the biological interpretation of it.

An iterated filtering algorithm (IF2)

- We use the IF2 algorithm of (Ionides et al.; 2015).
- A particle filter is 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 a new particle filter with a smaller random walk variance.
- Theoretically, this procedure converges toward the region of parameter space maximizing the maximum likelihood.
- Empirically, we can test this claim on examples.

IF2 algorithm pseudocode

model 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: Number of iterations, M; number of particles, J; initial parameter swarm, $\{\Theta^0_j, j = 1,\dots,J\}$; perturbation density, $h_n(\theta|\varphi;\sigma)$; perturbation scale, $\sigma_{1:M}$

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

1. For m in 1:M 2. $\Theta^{F,m}_{0,j} \sim h_0(\theta|\Theta^{m-1}_j;\sigma_m)$ for j in 1:J 3. $X^{F,m}_{0,j} \sim f_{X_0}(x_0;\Theta^{F,m}_{0,j})$ for j in 1:J 4. For n in 1:N 5. $\Theta^{P,m}_{n,j} \sim h_n(\theta|\Theta^{F,m}_{n-1,j},\sigma_m)$ for j in 1:J 6. $X^{P,m}_{n,j} \sim f_{X_n|X_{n-1}}(x_n|X^{F,m}_{n-1,j};\Theta^{P,m}_j)$ for j in 1:J 7. $w^m_{n,j} = f_{Y_n|X_n}(y^*_n|X^{P,m}_{n,j};\Theta^{P,m}_{n,j})$ for j in 1:J 8. Draw $k_{1:J}$ with $P[k_j = i] = w^m_{n,i} \Big/ \sum_{u=1}^{J} w^m_{n,u}$ 9. $\Theta^{F,m}_{n,j} = \Theta^{P,m}_{n,k_j}$ and $X^{F,m}_{n,j} = X^{P,m}_{n,k_j}$ for j in 1:J 10. End For 11. Set $\Theta^m_j = \Theta^{F,m}_{N,j}$ for j in 1:J 12. End For

Comments on the IF2 algorithm

- 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, with the particles $j = 1, \ldots, J$ providing a Monte Carlo representation of the conditional distribution at time n given data $y_{1:n-1}^*$ for filtering iteration m.
- The weight $w_{n,j}^m$ gives the likelihood of the data at time n for particle j in filtering iteration m.

Choosing the algorithmic settings for IF2

- The initial parameter swarm, $\{\Theta_j^0, j = 1, \ldots, J\}$, usually consists of J identical replications of some starting parameter vector.
- J is set to be sufficient for particle filtering. By the time of the last iteration ($m = M$) one should not have effective sample size close to 1.
- Perturbations are usually chosen to be Gaussian, with σ_m being a scale factor for iteration m:

$$h_n(\theta|\varphi;\sigma) \sim N[\varphi, \sigma_m^2 V_n].$$

- V_n is usually taken to be diagonal,

$$V_n = \begin{pmatrix} v_{1,n}^2 & 0 & 0 & \to & 0 \\ 0 & v_{2,n}^2 & 0 & \to & 0 \\ 0 & 0 & v_{3,n}^2 & \to & 0 \\ \downarrow & & & \searrow & \downarrow \\ 0 & 0 & 0 & \to & v_{p,n}^2 \end{pmatrix}.$$

- If θ_i is a parameter that affects the dynamics or observations throughout the timeseries, it is called a **regular parameter**, and it is

Applying IF2 to a boarding school influenza outbreak

- We're going to reintroduce the boarding school flu example. This provides a reminder, but also lets us develop the model and the corresponding pomp object in a way that generalizes to other situations. It will be a template for the case studies that follow.
- For a relatively simple epidemiological example of IF2, we consider fitting a stochastic SIR model to an influenza outbreak in a British boarding school [@anonymous78]. Reports consist of the number of children confined to bed for each of the 14 days of the outbreak. The total number of children at the school was 763, and a total of 512 children spent time away from class. Only one adult developed influenza-like illness, so adults are omitted from the data and model. First, we read in the boarding school flu (bsflu) data:

```
bsflu_data <- read.table("bsflu_data.txt")
```

- Our model is a variation on a basic SIR Markov chain, with state $X(t) = (S(t), I(t), R_1(t), R_2(t), R_3(t))$ giving the number of individuals in the susceptible and infectious categories, and three stages of recovery.

The observation names (B, C) are the names of the data variables:

```
(bsflu_obsnames <- colnames(bsflu_data)[1:2])
```

```
## [1] "B" "C"
```

We do not need a representation of R_3 since the total population size is fixed at $P = 763$ and hence $R_3(t) = P - S(t) - I(t) - R_1(t) - R_2(t)$. Now, we write the model code:

```
source("bsflu2.R")
```

Controlling the run time

- To develop and debug code, it is nice to have a version that runs extra quickly, for which we set `run_level=1`. Here, `Np` is the number of particles (i.e., sequential Monte Carlo sample size), and `Nmif` is the number of iterations of the optimization procedure carried out below. Empirically, `Np=5000` and `Nmif=200` are around the minimum required to get stable results with an error in the likelihood of order 1 log unit for this example; this is implemented by setting `run_level=2`. One can then ramp up to larger values for more refined computations, implemented here by `run_level=3`.

```
run_level <- 1
switch(run_level, {
  bsflu_Np=100; bsflu_Nmif=10; bsflu_Neval=10; bsflu_Nglobal=10; bs
  },
       {bsflu_Np=20000; bsflu_Nmif=100; bsflu_Neval=10; bsflu_Nglob
       {bsflu_Np=60000; bsflu_Nmif=300; bsflu_Neval=10; bsflu_Nglob
)
```

Running a particle filter

Before engaging in iterated filtering, it is often a good idea to check that the basic particle filter is working since iterated filtering builds on this technique. Here, carrying out slightly circular reasoning, we are going to test `pfilter` on a previously computed point estimate read in from `bsflu_params.csv`

```
bsflu_params <- data.matrix(
  read.table("mif_bsflu_params.csv",
  row.names=NULL,header=TRUE))
bsflu_mle <- bsflu_params[which.max(bsflu_params[,"logLik"]),][bsflu
```

We are going to treat μ_{R_1} and μ_{R_2} as known, fixed at the empirical mean of the bed-confinement and convalescence times for symptomatic cases:

```
bsflu_fixed_params <- c(mu_R1=1/(sum(bsflu_data$B)/512),mu_R2=1/(sum
```

Parallelization

- It is convenient to do some parallelization to speed up the computations. Most machines are multi-core nowadays, and using this computational capacity involves only
  1. the following lines of code to let R know you plan to use multiple processors;
  2. using the parallel for loop provided by 'foreach'.

```
library(doParallel)
registerDoParallel()
library(doRNG)
```

- We proceed to carry out replicated particle filters at this tentative MLE:

```
stew(file=sprintf("pf-%d.rda",run_level),{
  t_pf <- system.time(
    pf <- foreach(i=1:20,.packages='pomp') %dopar% try(
                    pfilter(bsflu2,params=bsflu_mle,Np=bsflu_Np)
                  )
  )

},seed=1320290398,kind="L'Ecuyer")

(L_pf <- logmeanexp(sapply(pf,logLik),se=TRUE))

##                      se
## -77.3381503   0.5328977
```

Caching computations in Rmarkdown

- In `rround(t_pf["elapsed"],1)` seconds, we obtain an unbiased likelihood estimate of `rround(L_pf[1],2)` with a Monte standard error of `rround(L_pf[2],2)`.
- It is not unusual for computations in a POMP analysis to take hours to run on many cores.
- The computations for a final version of a manuscript may take days.
- Usually, we use some mechanism like the different values of `run_level` so that preliminary versions of the manuscript take less time to run.
- However, when editing the text or working on a different part of the manuscript, we don't want to re-run long pieces of code.
- Saving results so that the code is only re-run when necessary is called **caching**.

- You may already be familiar with Rmarkdown's own version of caching.
- In the notes, we tell Rmarkdown to cache. For example, in (notes13.Rmd) the first R chunk, called `knitr-opts`, contains the following:

```
opts_chunk$set(
  cache=TRUE,
  )
```

- Rmarkdown uses a library called `knitr` to process the Rmd file, so options for Rmarkdown are formally options for knitr.
- Having set the option `cache=TRUE`, Rmarkdown caches every chunk, meaning that a chunk will only be re-run if code in that chunk is edited.
- You can force Rmarkdown to recompute all the chunks by deleting the `cache` subdirectory.

Practical advice for caching

- What if changes elsewhere in the document affect the proper evaluation of your chunk, but you didn't edit any of the code in the chunk itself? Rmarkdown will get this wrong. **It will not recompute the chunk**.
- A perfect caching system doesn't exist. **Always delete the entire cache and rebuild a fresh cache before finishing a manuscript.**
- Rmarkdown caching is good for relatively small computations, such as producing figures or things that may take a minute or two and are annoying if you have to recompute them every time you make any edits to the text.
- For longer computations, it is good to have full manual control. In `pomp`, this is provided by two related functions, `stew` and `bake`.

stew and bake

- Notice the function `stew` in the replicated particle filter code above.
- Here, `stew` looks for a file called `pf-[run_level].rda`.
- If it finds this file, it simply loads the contents of this file.
- If the file doesn't exist, it carries out the specified computation and saves it in a file of this name.
- `bake` is similar to `stew`. The difference is that `bake` uses `readRDS` and `saveRDS`, whereas `stew` uses `load` and `save`.
- either way, the computation will not be re-run unless you manually delete `pf-[run_level].rda`.
- `stew` and `bake` reset the seed appropriately whether or not the computation is recomputed. Othewise, caching risks adverse consequences for reproducibility.

A local search of the likelihood surface

- Let's carry out a local search using `mif2` around this previously identified MLE. For that, we need to set the `rw.sd` and `cooling.fraction.50` algorithmic parameters:

```
bsflu_rw.sd <- 0.02
bsflu_cooling.fraction.50 <- 0.5

stew(file=sprintf("local_search-%d.rda",run_level),{

  t_local <- system.time({
    mifs_local <- foreach(i=1:bsflu_Nlocal,.packages='pomp', .combi
      mif2(
        bsflu2,
        params=bsflu_mle,
        Np=bsflu_Np,
        Nmif=bsflu_Nmif,
        cooling.type="geometric",
        cooling.fraction.50=bsflu_cooling.fraction.50,
        rw.sd=rw.sd(
          Beta=bsflu_rw.sd,
```

- Although the filtering carried out by `mif2` in the final filtering iteration generates an approximation to the likelihood at the resulting point estimate, this is not usually good enough for reliable inference. Partly, this is because some parameter perturbations remain in the last filtering iteration. Partly, this is because `mif2` is usually carried out with a smaller number of particles than is necessary for a good likelihood evaluation—the errors in `mif2` average out over many iterations of the filtering.
- Therefore, we evaluate the likelihood, together with a standard error, using replicated particle filters at each point estimate:

```
stew(file=sprintf("lik_local-%d.rda",run_level),{
    t_local_eval <- system.time({
    liks_local <- foreach(i=1:bsflu_Nlocal,.packages='pomp',.combine
      evals <- replicate(bsflu_Neval, logLik(pfilter(bsflu2,params=
      logmeanexp(evals, se=TRUE)
    }
  })
},seed=900242057,kind="L'Ecuyer")

results_local <- data.frame(logLik=liks_local[,1],logLik_se=liks_lo
```

A global search of the likelihood surface using randomized starting values

- 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). By convention, we use **initial values** for the initialization of the dynamic system and **starting values** for initialization of the parameter search.
- Practical parameter estimation involves trying many starting values for the parameters. One can specify a large box in parameter space that contains all parameter vectors which seem remotely sensible. 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 flu model, a box containing reasonable parameter values might be

```
bsflu_box <- rbind(
  Beta=c(0.001,0.01),
  mu_IR=c(0.5,2),
```

Diagnosing success or failure of the maximization procedure

- The `plot` method for an object of class `mif2d.pomp` presents some graphical convergence and filtering diagnostics for the maximization procedure.
- It is often useful to look at superimposed convergence diagnostic plots for multiple Monte Carlo replications of the maximization procedure, perhaps with different starting values.
- Concatenating objects of class `mif2d.pomp` gives a list of class `mif2List`. The `plot` method for a mif2List object gives us the superimposed convergence diagnostic plots.
- Above, we built a list of mif2d.pomp objects for the global maximum likelihood search, fitting a model to the boarding school flu data. Let's first check the classes of the resulting objects.

```
class(mifs_global)

## [1] "mif2List"
## attr(,"package")
## [1] "pomp"
```

Question 12.4. Interpreting the diagnostics.

1. Do these plots suggest we have successfully maximized the likelihood, or not? Why?
2. What would the convergence plots look like if we cooled too quickly? Or too slowly? Can you find evidence for either of these here? (The algorithmic parameter `cooling.fraction.50` is the fraction by which we decrease the random walk standard deviation in 50 filtering iterations.)
3. Here, we did 300 `mif` iterations. Should we have done more? Could we have saved ourselves computational effort by doing less, without compromising our analysis?
4. Some parameter estimates show strong agreement between the different mif runs from different starting values. Others less so. How do you interpret this? Diversity in parameter estimates could be a signal of poor numerical maximization. It could signal a multi-modal likelihood surface. Or, it could simply correspond to a flat likelihood surface where the maximum is not precisely identifiable. Can we tell from the diagnostic plots which of these is going on here?

Effective sample size

- Maximization via particle filtering requires that the particle filter is working effectively. One way to monitor this is to pay attention to the **effective sample size** on the last filtering iteration.
- The effective sample size (ESS) is computed here as

$$\mathrm{ESS}_n = \frac{\left(\sum_{j=1}^{J} w_{n,j}\right)^2}{\sum_{j=1}^{J} w_{n,j}^2},$$

  where $\{w_{n,j}\}$ are the weights defined in step 3 of the particle filter pseudo code.
- The ESS approximates the number of independent, equally weighted, samples from the filtering distribution that would be equally informative to the one weighted sample that we have obtained by the particle filter.
- For our example, do you have any concerns about the number of particles?

Question 12.5. **Constructing a profile likelihood**. How strong is the evidence about the contact rate, β, given this model and data? Use `mif2` to construct a profile likelihood. Due to time constraints, you may be able to compute only a preliminary version.

- How would you profile over the basic reproduction number, $R_0 = \beta P/\mu_{IR}$. Is this more or less well determined that β for this model and data?

Checking model source code

For various reasons, it can be surprisingly hard to make sure that the written equations and the code are perfectly matched. Here are some things to think about:

1. Papers should be written to be readable to as broad a community as possible. Code must be written to run successfully. People do not want to clutter papers with numerical details which they hope and belief are scientifically irrelevant. What problems can arise due to this, and what solutions are available?
2. Suppose that there is an error in the coding of `rprocess`. Suppose that plug-and-play statistical methodology is used to infer parameters. A conscientious researcher carries out a simulation study, using `simulate` to generate some realizations from the fitted model and checking that the inference methodology can successfully recover the known parameters for this model, up to some statistical error. Will this procedure help to identify the error in `rprocess`? If not, how does one debug `rprocess`? What research practices help minimize the risk of errors in simulation code?

Question 12.6. Check the source code for the `bsflu pomp` object. Does the code implement the model described?

Question 12.7. **Assessing and improving algorithmic parameters**. Develop your own heuristics to try to improve the performance of `mif2` in the previous example. Specifically, for a global optimization procedure carried out using random starting values in the specified box, let $\hat{\Theta}_{\max}$ be a random Monte Carlo estimate of the resulting MLE, and let $\hat{\theta}$ be the true (unknown) MLE. We can define the maximization error in the log likelihood to be

$$e = \ell(\hat{\theta}) - E[\ell(\hat{\Theta}_{\max})].$$

We cannot directly evaluate e, since there is also Monte Carlo error in our evaluation of $\ell(\theta)$, but we can compute it up to a known precision. Plan some code to estimates e for a search procedure using a computational effort of $JM = 2 \times 10^7$, comparable to that used for each mif computation in the global search. Discuss the strengths and weaknesses of this quantification of optimization success. See if you can choose J and M subject to this constraint, together with choices of `rw.sd` and the cooling rate, `cooling.fraction.50`, to arrive at a quantifiably better procedure. Computationally, you may not be readily able to run your full procedure, but you could run a quicker version of it.

Question 12.8. **Finding sharp peaks in the likelihood surface**. Even in this small, 3 parameter, example, it takes a considerable amount of computation to find the global maximum (with values of β around 0.004) starting from uniform draws in the specified box. The problem is that, on the scale on which "uniform" is defined, the peak around $\beta \approx 0.004$ is very narrow. Propose and test a more favorable way to draw starting parameters for the global search, with better scale invariance properties.

Question 12.9. **Adding a latent class**. Modify the model to include a latent period between becoming exposed and becoming infectious. See what effect this has on the maximized likelihood.

Acknowledgments and License

- Produced with R version 3.6.2 and `pomp` version 2.7.
- These notes build on previous versions at `ionides.github.io/531w16` and `ionides.github.io/531w18`.
- Those notes draw on material developed for a short course on Simulation-based Inference for Epidemiological Dynamics (`http://kingaa.github.io/sbied/`) by Aaron King and Edward Ionides, taught at the University of Washington Summer Institute in Statistics and Modeling in Infectious Diseases, from 2015 through 2019.

References I

Arulampalam, M. S., Maskell, S., Gordon, N. and Clapp, T. (2002). A tutorial on particle filters for online nonlinear, non-Gaussian Bayesian tracking, *IEEE Trans. Sig. Proc.* **50**: 174–188.

Doucet, A., de Freitas, N. and Gordon, N. (eds) (2001). *Sequential Monte Carlo Methods in Practice*, Springer-Verlag, New York.

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, *Proceedings of the National Academy of Sciences of USA* **112**(3): 719—-724.
URL:
http://www.pnas.org/content/early/2015/01/07/1410597112.abstract

Kitagawa, G. (1987). Non-Gaussian state-space modeling of nonstationary time series, *Journal of the American Statistical Association* **82**(400): 1032–1041.