Particle Filters

Dexter Barrows
November 11, 2015

1 Intro

Particle filters are similar to MCMC-based methods in that they attempt to draw samples from an approximation of the posterior distribution of model parameters θ given observed data D. Instead of constructing a Markov chain and approximating its stationary distribution, a cohort of "particles" are used to move through the data in an on-line (sequential) fashion with the cohort being culled of poorly-performing particles at each iteration via importance sampling. If the culled particles are not replenished, this will be a Sequential Importance Sampling (SIS) particle filter. If the culled particles are replenished from surviving particles, in a sense setting up a process not dissimilar from Darwinian selection, then this will be a Sequential Importance Resampling (SIR) particle filter.

2 Formulation

Particle filters, also called Sequential Monte-Carlo (SMC) or bootstrap filters, feature similar core functionality as the venerable Kalman Filter. As the algorithm moves through the data (sequence of observations), a prediction-update cycle is used to simulate the evolution of the model M with different particular parameter selections, track how closely these predictions approximate the new observed value, and update the current cohort appropriately.

Two separate functions are used to simulate the evolution and observation processes. The "true" state evolution is specified by

$$X_{t+1} \sim f_1(X_t, \theta), \tag{1}$$

And the observation process by

$$Y_t \sim f_2(X_t, \theta). \tag{2}$$

Note that components of θ can contribute to both functions, but a typical formulation is to have some components contribute to $f_1(\cdot, \theta)$ and others to $f_2(\cdot, \theta)$.

The prediction part of the cycle utilises $f_1(\cdot, \theta)$ to update each particle's current state estimate to the next time step, while $f_2(\cdot, \theta)$ is used to evaluate a weighting w for each particle which will be used to determine how closely that particle is estimating the true underlying state of the system. Note that $f_2(\cdot, \theta)$ could be thought of as a probability of observing a piece of data y_t given the particle's current state estimate and parameter set, $P(y_t|X_t, \theta)$. Then, the new cohort of particles is drawn from the old cohort proportional to the weights. This process is repeated until the set of observations D is exhausted.

3 Algorithm

Now we can formalize the particle filter.

We will denote each particle $p^{(j)}$ as the j^{th} particle consisting of a state estimate at time t, $X_t^{(j)}$, a parameter set $\theta^{(j)}$, and a weight $w^{(j)}$. Note that the state estimates will evolve with the system as the cohort traverses the data.

The algorithm for a Sequential Importance Resampling particle is shown in Algorithm 1.

Algorithm 1: SIR particle filter

/* Select a starting point */
Input : Observations $D = y_1, y_2, ..., y_T$, initial particle distribution P_0 of size J

/* Setup */
Initialize particle cohort by sampling $(p^{(1)}, p^{(2)}, ..., p^{(J)})$ from P_0
for $t = 1 : T$ **do**
 /* Evolve */
 for j = *1:J* **do**
  $X_t^{(j)} \leftarrow f_1(X_{t-1}^{(j)}, \theta^{(j)})$
 /* Weight */
 for j = *1:J* **do**
  $w^{(j)} \leftarrow P(y_t|X_t^{(j)}, \theta^{(j)}) = f_2(X_t^{(j)}, \theta^{(j)})$
 /* Normalize */
 for j = *1:J* **do**
  $w^{(j)} \leftarrow w^{(j)} / \sum_1^J w^{(j)}$
 /* Resample */
 $p^{(1:J)} \leftarrow \text{sample}(p^{(1:J)}, \text{prob} = w, \text{replace} = true)$

/* Samples from approximated posterior distribution */
Output: Cohort of posterior samples $(\theta^{(1)}, \theta^{(2)}, ..., \theta^{(J)})$

4 Particle Collapse

Not uncommonly, a situation may arise in which a single particle gets assigned a normalized weight very close to 1 and all the other particles have weights very close to 0. When this occurs, the next generation of the cohort will overwhelmingly consist of descendants of the heavily-weighted particle, termed particle collapse or degeneracy.

Since the basic SIR particle filter does not perturb either the particle system states or system parameter values, the cohort will quickly consist solely of identical particles, effectively halting further exploration of the parameter space as new data is introduced.

A similar situation occurs when a small number of particles (but not necessarily a single particle) split almost all of the normalized weight between them, then jointly dominate the resampling process for the remainder of the iterations. This again halts the exploration of the parameter space with new data.

In either case, the hallmark feature used to detect collapse is the same - at some point the cohort will consist of particles with very similar or identical parameter sets which will consequently result in their assigned weights being extremely close.

Mathematically, we are interested in the number of effective particles, N_{eff}, which represents the number of particles that are acceptably dissimilar. This is estimated by evaluating

$$N_{eff} = \frac{1}{\sum_1^J (w^{(j)})^2}. \tag{3}$$

This can be used to diagnose not only when collapse has occurred, but can also indicate when it is near.

Diagram!!!!!!

5 Iterated Filtering and Data Cloning

A particle filter hinges on the idea that as it progresses through the data set D, its estimate of the the posterior carried in the cohort of particles approaches maximum likelihood. However, this convergence may not converge fast enough that the estimate it produces is of quality before the data runs out. One way around this problem is to "clone" the data and make multiple passes through it as if it were a continuation of the original time series. Note that the system state contained in each particle will have to be reset with each pass.

Rigorous proofs have been developed (references to Ionides et. al. work) that show that by treating the parameters as stochastic processes instead of fixed values, the multiple passes through the data will indeed force convergence of the process mean toward maximum likelihood, and the process variance toward 0.

6 IF2

The successor to Iterated Filtering 1 (reference), Iterated Filtering 2 is simpler, faster, and demonstrated better convergence toward maximum likelihood (reference). The core concept involves a two-pronged approach. First, Data cloning is used to allow more time for the parameter stochastic process means to converge to maximum likelihood, and frequent cooled perturbation of the particle parameters allow better exploration of the parameter space while still allowing convergence to good point estimates.

It is worth noting that IF2 is not designed to estimate the full posterior distribution, but in practice can be used to do so within reason. Further, IF2 thwarts the problem of particle collapse by keeping at least some perturbation in the system at all times. It is important to note that while true particle collapse will not occur, there is still risk of a pseudo-collapse in which all particles will be extremely close to one another so as to be virtually indistinguishable. However this will only occur with the use of overly-aggressive cooling strategies or by specifying an excessive number of passes through the data.

An important new quantity is the particle perturbation density denoted $h(\theta|, \sigma)$. Typically this is multi-normal with σ being a vector of variances proportional to the expected values of θ. In practice the proportionality can be derived from current means or specified ahead of time. Further, these intensities must decrease over time. This can be done via exponential cooling, a decreasing step function, a combination of these, or some similar way.

The algorithm for IF2 can be seen in Algorithm 2.

Algorithm 2: IF2

/* Select a starting point */

Input : Observations $D = y_1, y_2, ..., y_T$, initial particle distribution P_0 of size J, decreasing sequence of perturbation intensity vectors $\sigma_1, \sigma_2, ..., \sigma_M$

/* Setup */

Initialize particle cohort by sampling $(p^{(1)}, p^{(2)}, ..., p^{(J)})$ from P_0

/* Particle seeding distribution */

$\Theta \leftarrow P_0$

for $m = 1 : M$ **do**

    /* Pass perturbation */

    for j = *1:J* **do**

        $p^{(j)} \sim h(\Theta^{(j)}, \sigma_m)$

    for $t = 1 : T$ **do**

        for j = *1:J* **do**

            /* Iteration perturbation */

            $p^{(j)} \sim h(p^{(j)}, \sigma_m)$

            /* Evolve */

            $X_t^{(j)} \leftarrow f_1(X_{t-1}^{(j)}, \theta^{(j)})$

            /* Weight */

            $w^{(j)} \leftarrow P(y_t | X_t^{(j)}, \theta^{(j)}) = f_2(X_t^{(j)}, \theta^{(j)})$

        /* Normalize */

        for j = *1:J* **do**

            $w^{(j)} \leftarrow w^{(j)} / \sum_1^J w^{(j)}$

        /* Resample */

        $p^{(1:J)} \leftarrow \text{sample}(p^{(1:J)}, \text{prob} = w, \text{replace} = \textit{true})$

    /* Collect particles for next pass */

    for $j = 1 : J$ **do**

        $\Theta^{(j)} \leftarrow p^{(j)}$

/* Samples from approximated posterior distribution */

Output: Cohort of posterior samples $(\theta^{(1)}, \theta^{(2)}, ..., \theta^{(J)})$

7 Fitting an SIR Model to Synthetic Epidemic Data with IF2

Here we will examine a test case in which IF2 will be used to fit a Susceptible-Infected-Removed (SIR) epidemic model to mock infectious count data.

The synthetic data was produced by taking the solution to a basic SIR ODE model, sampling it at regular intervals, and perturbing those values by adding in observation noise. The SIR model used was

$$\begin{aligned} \frac{dS}{dt} &= -\beta I S \\ \frac{dI}{dt} &= \beta I S - rI \\ \frac{dR}{dt} &= rI \end{aligned} \tag{4}$$

where S is the number of individuals susceptible to infection, I is the number of infectious individuals, R is the number of recovered individuals, $\beta = R_0 r/N$ is the force of infection, R_0 is the number of secondary cases per infected individual, r is the recovery rate, and N is the population size.

The solution to this system was obtained using the `ode()` function from the `deSolve` package. The required derivative array function in the format required by `ode()` was specified as

```
SIR <- function(Time, State, Pars) {

    with(as.list(c(State, Pars)), {

        B   <- R0*r/N     # calculate Beta
        BSI <- B*S*I      # save product
        rI  <- r*I        # save product

        dS = -BSI         # change in Susceptible people
        dI = BSI - rI     # change in Infected people
        dR = rI           # change in Removed (recovered people)

        return(list(c(dS, dI, dR)))

    })

}
```

The true parameter values were set to $R_0 = 3.0, r = 0.1, N = 500$ by

```
pars  <- c(R0  = 3.0,  # new infected people per infected person
           r   = 0.1,  # recovery rate
           N   = 500)  # population size
```

The initial conditions were set to 5 infectious individuals, 495 people susceptible to infection, and no one had yet recovered from infection and been removed. These were set using

```
true_init_cond <- c(S = N - i_infec,
                    I = i_infec,
                    R = 0)
```

The `ode()` function is called as

```
odeout <- ode(y = true_init_cond, times = 0:(T-1), func = SIR, parms =
    true_pars)
```

where `odeout` is a $T \times 4$ matrix where the rows correspond to solutions at the given times (the first row is the initial condition), and the columns correspond to the solution times and S-I-R counts at those times.

The observation error was taken to be $\varepsilon_{obs} \sim \mathcal{N}(0, \sigma)$, where individual values were drawn for each synthetic data point.

These "true" values were perturbed to mimic observation error by

```
set.seed(1001)  # set RNG seed for reproducibility
sigma <- 10     # observation error standard deviation
infec_counts_raw <- odeout[,3] + rnorm(101, 0, sigma)
infec_counts     <- ifelse(infec_counts_raw < 0, 0, infec_counts)
```

where the last two lines simply set negative observations (impossible) to 0.

Plotting the data using the `ggplot2` package by

```
plotdata <- data.frame(times=1:T,true=trueTraj,data=infec_counts)

g <- ggplot(plotdata, aes(times)) +
        geom_line(aes(y = true, colour = "True")) +
        geom_point(aes(y = data, color = "Data")) +
        labs(x = "Time", y = "Infection count", color = "") +
        scale_color_brewer(palette="Paired") +
        theme(panel.background = element_rect(fill = "#F0F0F0"))
```

we obtain Figure 1.

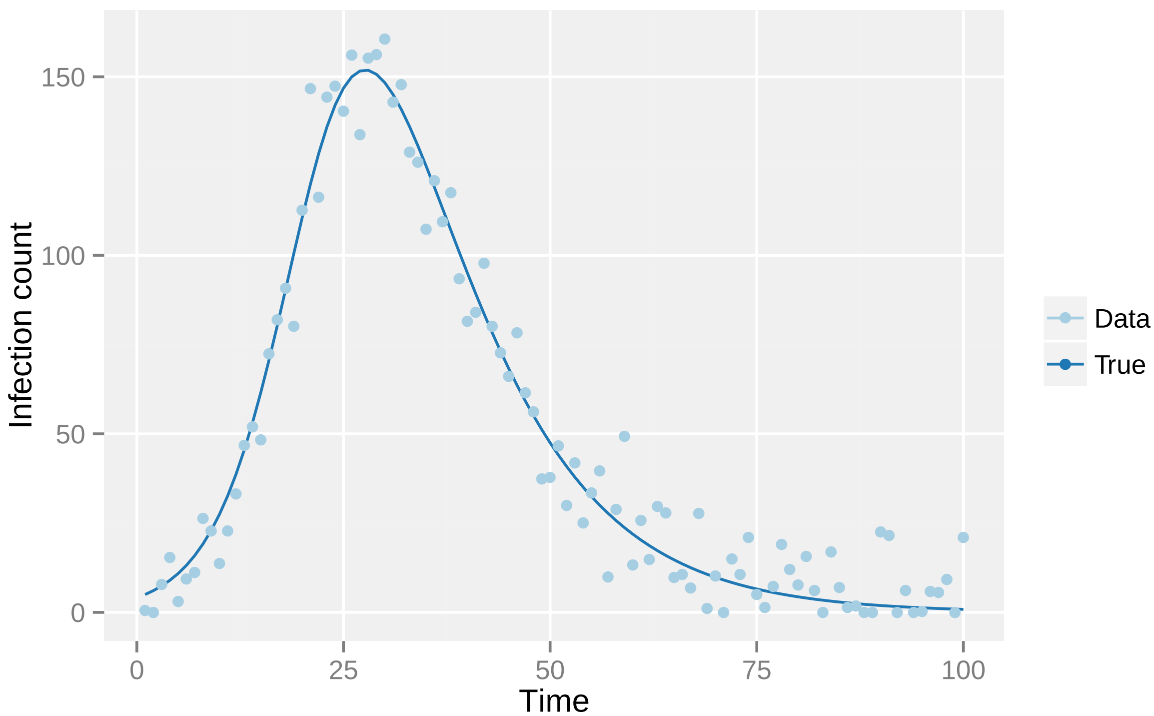

Figure 1: True SIR ODE solution infected counts, and with added observation noise

The IF2 algorithm was implemented in C++ for speed, and integrated into the R workflow using the `Rcpp` package. The C++ code is compiled using

```
sourceCpp(paste(getwd(),"if2.cpp",sep="/"))
```

Then run and packed into a data frame using

```
paramdata <- data.frame(if2(infec_counts[1:Tlim], Tlim, N))
colnames(paramdata) <- c("R0", "r", "sigma", "Sinit", "Iinit", "Rinit")
```

The final kernel estimates for four of the key parameters are shown in Figure 2.

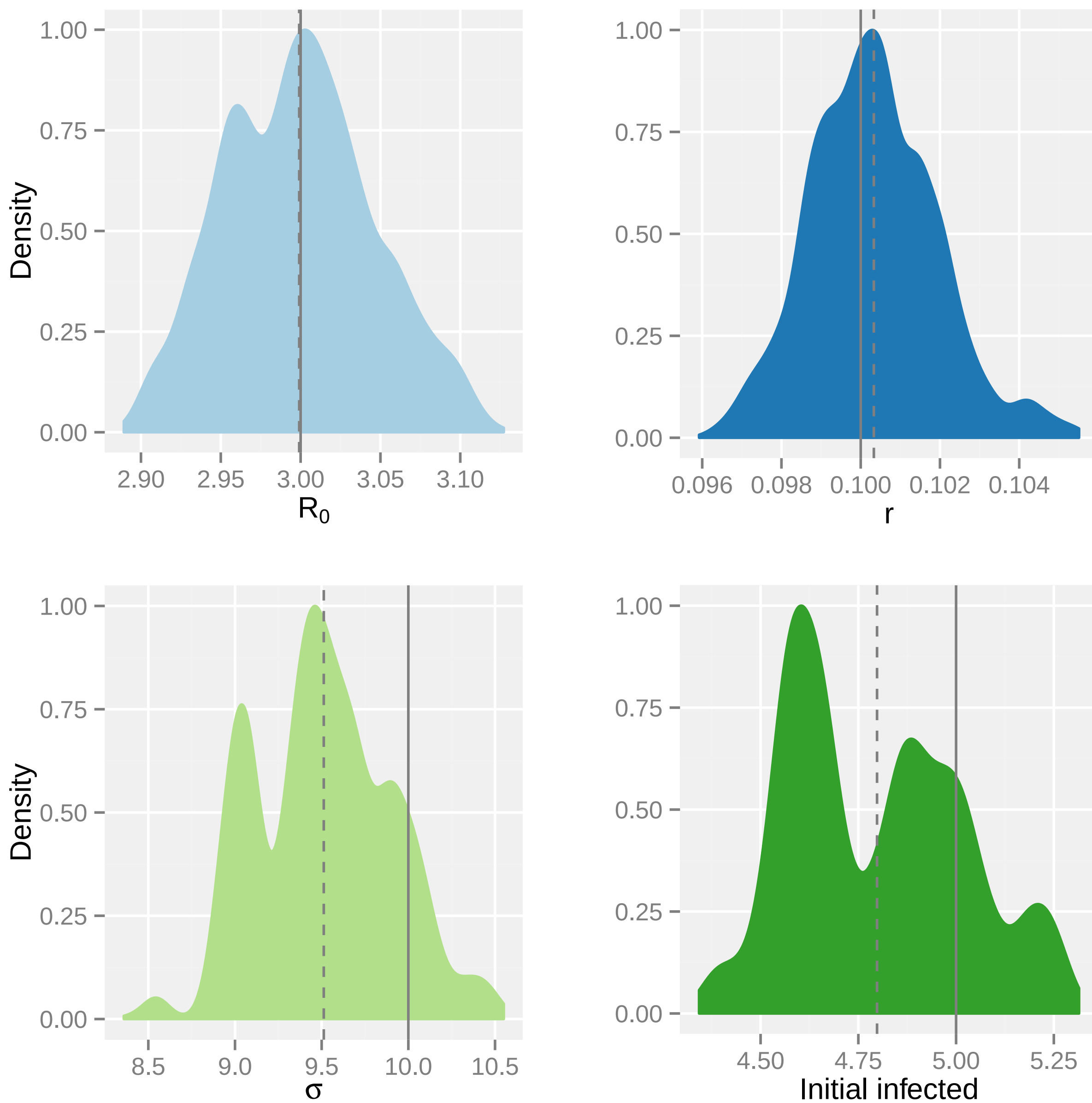

Figure 2: Kernel estimates for four essential system parameters. True values are indicated by solid vertical lines, sample means by dashed lines.

Bootstrapping from the final particle sample gives us awesome stuff too as we can see in Figure 3.

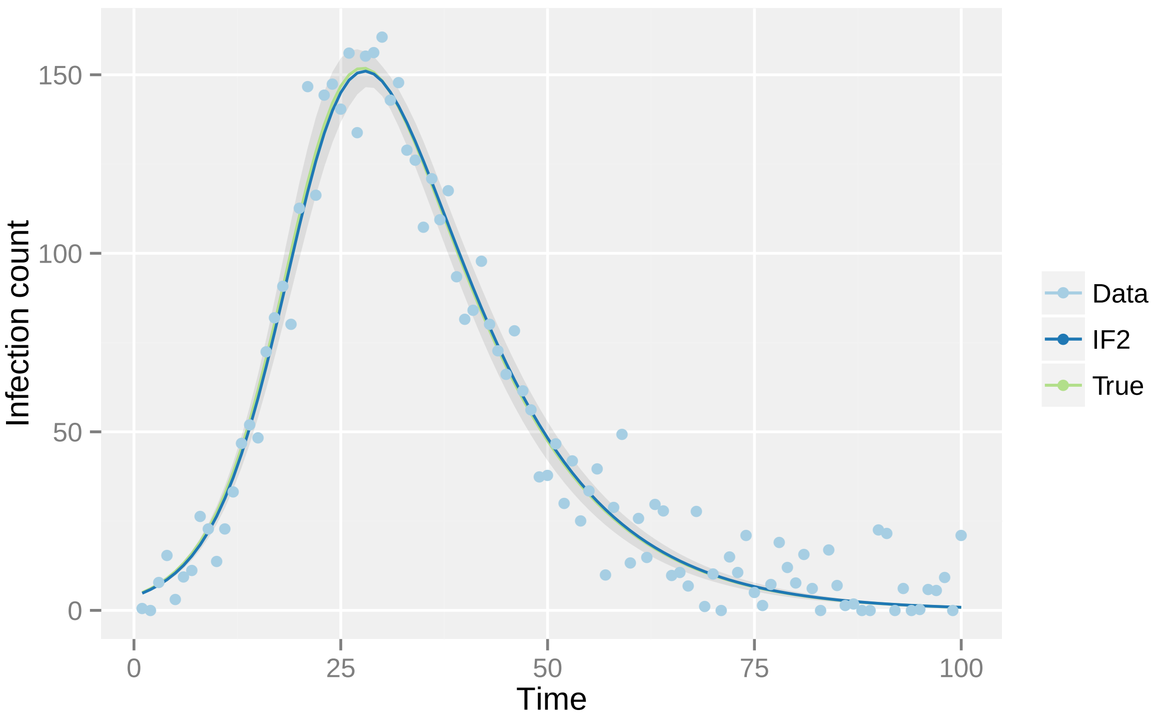

Figure 3: IF2 estimate of the system state at each time point using bootstrapping. Ribbon shows 2.5% and 97.5% quantiles.

Appendices

A Full R code

This code will run all the indicated analysis and produce all plots.

```
library(deSolve)
library(ggplot2)
library(reshape2)
library(gridExtra)
library(Rcpp)
library(RColorBrewer)

SIR ← function(Time, State, Pars) {

    with(as.list(c(State, Pars)), {

        B   ← R0*r/N
        BSI ← B*S*I
        rI  ← r*I

        dS = -BSI
        dI = BSI - rI
        dR = rI

        return(list(c(dS, dI, dR)))

    })

}

T       ← 100
N       ← 500
sigma   ← 10
i_infec ← 5
Tlim    ← T

## Generate true trajecory and synthetic data
##

true_init_cond ← c(S = N - i_infec,
                   I = i_infec,
                   R = 0)

true_pars ← c(R0 = 3.0,
              r = 0.1,
              N = 500.0)

odeout ← ode(true_init_cond, 0:(T-1), SIR, true_pars)
trueTraj ← odeout[,3]
```

```r

set.seed(1000)

infec_counts_raw ← odeout[,3] + rnorm(T, 0, sigma)
infec_counts     ← ifelse(infec_counts_raw < 0, 0, infec_counts_raw)

plotdata ← data.frame(times=1:T,true=trueTraj,data=infec_counts)

g ← ggplot(plotdata, aes(times)) +
        geom_line(aes(y = true, colour = "True")) +
        geom_point(aes(y = data, color = "Data")) +
        labs(x = "Time", y = "Infection count", color = "") +
        scale_color_brewer(palette="Paired") +
        theme(panel.background = element_rect(fill = "#F0F0F0"))

print(g)
ggsave(g, filename="dataplot.pdf", height=4, width=6.5)

## Rcpp stuff
##

sourceCpp(paste(getwd(),"if2.cpp",sep="/"))

paramdata ← data.frame(if2(infec_counts[1:Tlim], Tlim, N))
colnames(paramdata) ← c("R0", "r", "sigma", "Sinit", "Iinit", "Rinit")

## sample from parameter distributions
##

nTraj   ← 100
datlen  ← dim(paramdata)[1]
inds    ← sample.int(datlen,nTraj,replace = TRUE)
params  ← paramdata[inds,]

bootstrapdata ← matrix(NA, nrow = nTraj, ncol = T)

for (i in 1:nTraj) {

    init_cond ← c(S = params$Sinit[i],
                  I = params$Iinit[i],
                  R = params$Rinit[i])
    pars ← c(R0 = params$R0[i],
             r = params$r[i],
             N = 500.0)

    odeout ← ode(init_cond, 0:(T-1), SIR, pars)

    bootstrapdata[i,] ← odeout[,3]

}


meanTraj    ← colMeans(bootstrapdata)
quantTraj   ← apply(bootstrapdata, 2, quantile, probs = c(0.025,0.975))
```

```

datapart ← c(infec_counts[1:Tlim],rep(NA,T-Tlim))

plotdata ← data.frame(times=1:T,true=trueTraj,est=meanTraj,quants=t(quantTraj),datapart=datapart)

g ← ggplot(plotdata, aes(times)) +
        geom_ribbon(aes(ymin = quants.2.5., ymax=quants.97.5.), alpha=0.1) +
        geom_line(aes(y = true, colour = "True")) +
        geom_line(aes(y = est, colour = "IF2")) +
        geom_point(aes(y = datapart, color = "Data")) +
        labs(x = "Time", y = "Infection count", color = "") +
        scale_color_brewer(palette="Paired") +
        theme(panel.background = element_rect(fill = "#F0F0F0"))

print(g)
ggsave(g, filename="if2plot.pdf", height=4, width=6.5)

f ← function(pal) brewer.pal(brewer.pal.info[pal, "maxcolors"], pal)
kcolours ← f("Paired")


trueval.R0      ← 3.0
trueval.r       ← 0.1
trueval.sigma   ← 10.0
trueval.Iinit   ← 5

meanval.R0      ← mean(paramdata$R0)
meanval.r       ← mean(paramdata$r)
meanval.sigma   ← mean(paramdata$sigma)
meanval.Iinit   ← mean(paramdata$Iinit)

linecolour ← "grey50"
lineweight ← 0.5

kerdataR0 ← data.frame(R0points = paramdata$R0)
R0kernel ← ggplot(kerdataR0, aes(x = R0points, y = ..scaled..)) +
                geom_density(color = kcolours[1], fill = kcolours[1]) +
                theme(panel.background = element_rect(fill = "#F0F0F0")) +
                scale_color_brewer(palette="Paired") +
                labs(x = expression(R[0]), y = "Density", color = "") +
                geom_vline(aes(xintercept=trueval.R0), linetype="solid", size=lineweight, color=linecolour) +
                geom_vline(aes(xintercept=meanval.R0), linetype="dashed", size=lineweight, color=linecolour)

kerdatar ← data.frame(rpoints = paramdata$r)
rkernel ← ggplot(kerdatar, aes(x = rpoints, y = ..scaled..)) +
                geom_density(color = kcolours[2], fill = kcolours[2]) +
                theme(panel.background = element_rect(fill = "#F0F0F0")) +
                scale_color_brewer(palette="Paired") +
                labs(x = "r", y = "", color = "") +
```

```
                    geom_vline(aes(xintercept=trueval.r), linetype="solid", size=lineweight, color=linecolour) +
                    geom_vline(aes(xintercept=meanval.r), linetype="dashed", size=lineweight, color=linecolour)

kerdatasigma <- data.frame(sigmapoints = paramdata$sigma)
sigmakernel <- ggplot(kerdatasigma, aes(x = sigmapoints, y = ..scaled..)) +
                    geom_density(color = kcolours[3], fill = kcolours[3]) +
                    theme(panel.background = element_rect(fill = "#F0F0F0")) +
                    scale_color_brewer(palette="Paired") +
                    labs(x = expression(sigma), y = "Density", color = "") +
                    geom_vline(aes(xintercept=trueval.sigma), linetype="solid", size=lineweight, color=linecolour) +
                    geom_vline(aes(xintercept=meanval.sigma), linetype="dashed", size=lineweight, color=linecolour)

kerdatainfec <- data.frame(infecpoints = paramdata$Iinit)
infeckernel <- ggplot(kerdatainfec, aes(x = infecpoints, y = ..scaled..)) +
                    geom_density(color = kcolours[4], fill = kcolours[4]) +
                    theme(panel.background = element_rect(fill = "#F0F0F0")) +
                    scale_color_brewer(palette="Paired") +
                    labs(x = "Initial infected", y = "", color = "") +
                    geom_vline(aes(xintercept=trueval.Iinit), linetype="solid", size=lineweight, color=linecolour) +
                    geom_vline(aes(xintercept=meanval.Iinit), linetype="dashed", size=lineweight, color=linecolour)

# show grid
grid.arrange(R0kernel, rkernel, sigmakernel, infeckernel, ncol = 2, nrow = 2)

pdf("if2kernels.pdf", height = 6.5, width = 6.5)
grid.arrange(R0kernel, rkernel, sigmakernel, infeckernel, ncol = 2, nrow = 2)
dev.off()
```

B Full C++ code

Stan model code to be used with the preceding R code.

```
/*  Author: Dexter Barrows
    Github: dbarrows.github.io

    */

/*  Runs a particle filter on synthetic noisy data and attempts to
    reconstruct underlying true state at each time step. Note that
    this program uses gnuplot to plot the data, so an x11
    environment must be present.

```

```
    Also, the accompanying "pf.plg" file contains the instructions
    gnuplot will use. It must be present in the same directory as
    the executable generated by compiling this file.

    */

#include <stdio.h>
#include <math.h>
#include <sys/time.h>
#include <time.h>
#include <stdlib.h>
#include <string>

#include "rand.h"
#include "timer.h"

#define T       100         // time to simulate over
#define R0true  3.0         // infectiousness
#define rtrue   0.1         // recovery rate
#define N       500.0       // population size
#define merr    10.0        // expected measurement error


struct Particle {
    float R0;
    float r;
    float sigma;
    float S;
    float I;
    float R;
    float Sinit;
    float Iinit;
    float Rinit;
};

struct ParticleInfo {
    float R0mean;       float R0sd;
    float rmean;        float rsd;
    float sigmamean;    float sigmasd;
    float Sinitmean;    float Sinitsd;
    float Iinitmean;    float Iinitsd;
    float Rinitmean;    float Rinitsd;
};


int timeval_subtract (double *result, struct timeval *x, struct timeval *y);
int check_float(float x,float y);
void exp_euler_SIR(float h, float t0, float tn, Particle * particle);
void copyParticle(Particle * dst, Particle * src);
void perturbParticles(Particle * particles, int NP, int passnum, float coolrate);
bool isCollapsed(Particle * particles, int NP);
```

```
void particleDiagnostics(ParticleInfo * partInfo, Particle * particles, int
    NP);


int main(int argc, char *argv[]) {

    float   i_infec     = 5;
    int     Tlim        = T;
    int     NP          = 3000;
    int     nPasses     = 40;
    float   coolrate    = 7;

    srand(time(NULL));                              // Seed PRNG with
        system time

    Particle particle_true;
    particle_true.R0    = R0true;
    particle_true.r     = rtrue;
    particle_true.sigma = merr;
    particle_true.S     = N - i_infec;
    particle_true.I     = i_infec;
    particle_true.R     = 0;


    printf("System parameters\n");
    printf("-----------------\n");
    printf("R0:    %f\n", R0true);
    printf("r:     %f\n", rtrue);
    printf("merr:  %f\n", merr);

    float y_true[T];    // true number of infected peeps
    float y_noise[T];   // true number of infected peeps with observation
        noise
    float y_est[T];     // particle mean state estimation

    float y_par_noise;  // particle estimates with noise
    float w[NP];        // particle weights

    Particle particles[NP];     // particle estimates for current step
    Particle particles_old[NP]; // intermediate particle states for
        resampling

    // generate our true trajectory and noisy observation data
    y_true[0] = particle_true.I;
    y_noise[0] = y_true[0] +  merr*randn();
    if (y_noise[0] < 0)
        y_noise[0] = 0;
    for (int i = 1; i < T; i++) {
        exp_euler_SIR( 1.0/100, 0.0, 1.0, &particle_true);
        y_true[i] = particle_true.I;
        y_noise[i] = y_true[i] + merr*randn();
        if (y_noise[i] < 0)
            y_noise[i] = 0;
    }
```

```
    double restime;
    struct timeval  tdr0, tdr1;

    gettimeofday (&tdr0, NULL);



    printf("Initializing particle states\n");

    // initialize particle parameter states (seeding)
    for (int n = 0; n < NP; n++) {

        float R0can, rcan, sigmacan, Iinitcan;

        do {
            R0can = R0true + R0true*randn();
        } while (R0can < 0);
        particles[n].R0 = R0can;

        do {
            rcan = rtrue + rtrue*randn();
        } while (rcan < 0);
        particles[n].r = rcan;

        do {
            sigmacan = merr + merr*randn();
        } while (sigmacan < 0);
        particles[n].sigma = sigmacan;

        do {
            Iinitcan = i_infec + i_infec*randn();
        } while (Iinitcan < 0 || N < Iinitcan);
        particles[n].Sinit = N - Iinitcan;
        particles[n].Iinit = Iinitcan;
        particles[n].Rinit = 0.0;

    }

    // START PASSES THROUGH DATA



    printf("Starting filter\n");
    printf("---------------\n");
    printf("Pass");


    for (int pass = 0; pass < nPasses; pass++) {

        printf("...%d", pass);

        perturbParticles(particles, NP, pass, coolrate);

```

```
        // initialize particle system states
        for (int n = 0; n < NP; n++) {

            particles[n].S = particles[n].Sinit;
            particles[n].I = particles[n].Iinit;
            particles[n].R = particles[n].Rinit;

        }

        // between-pass perturbations

        for (int t = 1; t < Tlim; t++) {

            // between-iteration perturbations
            perturbParticles(particles, NP, pass, coolrate);

            // generate individual predictions and weight
            for (int n = 0; n < NP; n++) {

                exp_euler_SIR(1.0/10.0, 0.0, 1.0, &particles[n]);

                float merr_par  = particles[n].sigma;
                float y_diff    = y_noise[t] - particles[n].I;

                w[n] = 1.0/(merr_par*sqrt(2.0*M_PI)) * exp( - y_diff*y_diff
                    / (2.0*merr_par*merr_par) );

            }

            // cumulative sum
            for (int n = 1; n < NP; n++) {
                w[n] += w[n-1];
            }

            // save particle states to resample from
            for (int n = 0; n < NP; n++){
                copyParticle(&particles_old[n], &particles[n]);
            }

            // resampling
            for (int n = 0; n < NP; n++) {

                float w_r = randu() * w[NP-1];
                int i = 0;
                while (w_r > w[i]) {
                    i++;
                }

                // i is now the index to copy state from
                copyParticle(&particles[n], &particles_old[i]);

            }

        }
```

```
    }

    ParticleInfo pInfo;
    particleDiagnostics(&pInfo, particles, NP);

    printf("\n");

    gettimeofday (&tdr1, NULL);
    timeval_subtract (&restime, &tdr1, &tdr0);
    printf ("Single threaded runtime %e\n", restime);


    // Save paramenter distribution results for post-processing

    std::string paramfile("pfdata.dat");
    FILE * pfout = fopen(paramfile.c_str(), "w");

    //printf("Writing parameter results to file '%s'...\n", paramfile.c_str
        ());

    for (int n = 0; n < NP; n++) {

        fprintf(pfout, "%f ", particles[n].R0);
        fprintf(pfout, "%f ", particles[n].r);
        fprintf(pfout, "%f ", particles[n].sigma);
        fprintf(pfout, "%f ", particles[n].Sinit);
        fprintf(pfout, "%f ", particles[n].Iinit);
        fprintf(pfout, "%f\n", particles[n].Rinit);

    }

    fclose(pfout);

    // Save results for plotting

    std::string datafile("plotdata.dat");

    printf("Writing plotting results to file '%s'...\n", datafile.c_str());

    FILE * paramout = fopen(datafile.c_str(), "w");

    for (int t = 0; t < T; t++) {

        fprintf(paramout, "%d ", t);
        fprintf(paramout, "%f ", y_true[t]);

        if (t < Tlim)
            fprintf(paramout, "%f ", y_noise[t]);
        else
            fprintf(paramout, "%d ", -1);

        fprintf(paramout, "%f\n", y_est[t]);

```

```
    }

    fclose(paramout);

    printf("Plotting using gnuplot...\n");
    printf("Press ENTER close plot and continue\n");

    std::string syscall("gnuplot -e \"filename='");
    syscall += datafile;
    syscall += "'\" pf.plg";

    //system( syscall.c_str() );

}


/*  Use the Explicit Euler integration scheme to integrate SIR model forward in time
    float h     - time step size
    float t0    - start time
    float tn    - stop time
    float * y   - current system state; a three-component vector representing [S I R], susceptible-infected-recovered

    */
void exp_euler_SIR(float h, float t0, float tn, Particle * particle) {

    float t = t0;

    int num_steps = floor( (tn-t0) / h );

    float S = particle->S;
    float I = particle->I;
    float R = particle->R;

    float R0    = particle->R0;
    float r     = particle->r;
    float B     = R0 * r / N;

    for(int i = 0; i < num_steps; i++) {
        // get derivatives
        float dS = - B*S*I;
        float dI = B*S*I - r*I;
        float dR = r*I;
        // step forward by h
        S += h*dS;
        I += h*dI;
        R += h*dR;
    }

    particle->S = S;
    particle->I = I;
    particle->R = R;

```

```
}


/*  Particle pertubation function to be run between iterations and passes

    */
void perturbParticles(Particle * particles, int NP, int passnum, float coolrate) {

    float coolcoef = exp( - (float) passnum / coolrate );

    float spreadR0      = coolcoef * R0true / 3.0;
    float spreadr       = coolcoef * rtrue  / 3.0;
    float spreadsigma   = coolcoef * merr   / 3.0;
    float spreadIinit   = coolcoef * 5.0    / 3.0;

    float R0can, rcan, sigmacan, Iinitcan;

    for (int n = 0; n < NP; n++) {

        do {
            R0can = particles[n].R0 + spreadR0*randn();
        } while (R0can < 0);
        particles[n].R0 = R0can;

        do {
            rcan = particles[n].r + spreadr*randn();
        } while (rcan < 0);
        particles[n].r = rcan;

        do {
            sigmacan = particles[n].sigma + spreadsigma*randn();
        } while (sigmacan < 0);
        particles[n].sigma = sigmacan;

        do {
            Iinitcan = particles[n].Iinit + spreadIinit*randn();
        } while (Iinitcan < 0 || Iinitcan > 500);
        particles[n].Iinit = Iinitcan;
        particles[n].Sinit = N - Iinitcan;

    }

}


/*  Convinience function for particle resampling process

    */
void copyParticle(Particle * dst, Particle * src) {

    dst->R0     = src->R0;
    dst->r      = src->r;
    dst->sigma  = src->sigma;
```

```
    dst->S      = src->S;
    dst->I      = src->I;
    dst->R      = src->R;
    dst->Sinit  = src->Sinit;
    dst->Iinit  = src->Iinit;
    dst->Rinit  = src->Rinit;

}


/*  Checks to see if particles are collapsed
    This is done by checking if the standard deviations between the particles' parameter
    values are significantly close to one another. Spread threshold may need to be tuned.

    */
bool isCollapsed(Particle * particles, int NP) {

    bool retVal;

    float R0mean = 0, rmean = 0, sigmamean = 0, Sinitmean = 0, Iinitmean = 0, Rinitmean = 0;

    // means

    for (int n = 0; n < NP; n++) {

        R0mean      += particles[n].R0;
        rmean       += particles[n].r;
        sigmamean   += particles[n].sigma;
        Sinitmean   += particles[n].Sinit;
        Iinitmean   += particles[n].Iinit;
        Rinitmean   += particles[n].Rinit;

    }

    R0mean      /= NP;
    rmean       /= NP;
    sigmamean   /= NP;
    Sinitmean   /= NP;
    Iinitmean   /= NP;
    Rinitmean   /= NP;

    float   R0sd = 0, rsd = 0, sigmasd = 0, Sinitsd = 0, Iinitsd = 0, Rinitsd = 0;

    for (int n = 0; n < NP; n++) {

        R0sd    += ( particles[n].R0 - R0mean ) * ( particles[n].R0 - R0mean );
        rsd     += ( particles[n].r - rmean ) * ( particles[n].r - rmean );
        sigmasd += ( particles[n].sigma - sigmamean ) * ( particles[n].sigma - sigmamean );
```

```
        Sinitsd += ( particles[n].Sinit - Sinitmean ) * ( particles[n].Sinit - Sinitmean );
        Iinitsd += ( particles[n].Iinit - Iinitmean ) * ( particles[n].Iinit - Iinitmean );
        Rinitsd += ( particles[n].Rinit - Rinitmean ) * ( particles[n].Rinit - Rinitmean );

    }

    R0sd        /= NP;
    rsd         /= NP;
    sigmasd     /= NP;
    Sinitsd     /= NP;
    Iinitsd     /= NP;
    Rinitsd     /= NP;

    if ( (R0sd + rsd + sigmasd) < 1e-5)
        retVal = true;
    else
        retVal = false;

    return retVal;

}

void particleDiagnostics(ParticleInfo * partInfo, Particle * particles, int NP) {

    float   R0mean      = 0.0,
            rmean       = 0.0,
            sigmamean   = 0.0,
            Sinitmean   = 0.0,
            Iinitmean   = 0.0,
            Rinitmean   = 0.0;

    // means

    for (int n = 0; n < NP; n++) {

        R0mean      += particles[n].R0;
        rmean       += particles[n].r;
        sigmamean   += particles[n].sigma;
        Sinitmean   += particles[n].Sinit;
        Iinitmean   += particles[n].Iinit;
        Rinitmean   += particles[n].Rinit;

    }

    R0mean      /= NP;
    rmean       /= NP;
    sigmamean   /= NP;
    Sinitmean   /= NP;
    Iinitmean   /= NP;
    Rinitmean   /= NP;
```

```

    // standard deviations

    float   R0sd    = 0.0,
            rsd     = 0.0,
            sigmasd = 0.0,
            Sinitsd = 0.0,
            Iinitsd = 0.0,
            Rinitsd = 0.0;

    for (int n = 0; n < NP; n++) {

        R0sd    += ( particles[n].R0 - R0mean ) * ( particles[n].R0 - R0mean );
        rsd     += ( particles[n].r - rmean ) * ( particles[n].r - rmean );
        sigmasd += ( particles[n].sigma - sigmamean ) * ( particles[n].sigma - sigmamean );
        Sinitsd += ( particles[n].Sinit - Sinitmean ) * ( particles[n].Sinit - Sinitmean );
        Iinitsd += ( particles[n].Iinit - Iinitmean ) * ( particles[n].Iinit - Iinitmean );
        Rinitsd += ( particles[n].Rinit - Rinitmean ) * ( particles[n].Rinit - Rinitmean );

    }

    R0sd        /= NP;
    rsd         /= NP;
    sigmasd     /= NP;
    Sinitsd     /= NP;
    Iinitsd     /= NP;
    Rinitsd     /= NP;

    partInfo->R0mean    = R0mean;
    partInfo->R0sd      = R0sd;
    partInfo->sigmamean = sigmamean;
    partInfo->sigmasd   = sigmasd;
    partInfo->rmean     = rmean;
    partInfo->rsd       = rsd;
    partInfo->Sinitmean = Sinitmean;
    partInfo->Sinitsd   = Sinitsd;
    partInfo->Iinitmean = Iinitmean;
    partInfo->Iinitsd   = Iinitsd;
    partInfo->Rinitmean = Rinitmean;
    partInfo->Rinitsd   = Rinitsd;

}
```