Particle Filters

Dexter Barrows
February 27, 2016

1 Intro

Particle filters are usually presented in a frequentist framework ... the goal of a particle filter is typically to find *point estimates* of the top-level parameters by using the particle distribution to average over the lower-level (latent-state) parameters ...

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.

analogous to

2 Formulation

I think the bootstrap filter is the specific name for one particular simple particle filter (particle f.::bootstrap f. as MCMC::Metropolis-Hastings)

could use (many?) more citations - unless you're getting this all from a single source? 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 $f_2(\cdot, \theta)$.

1

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 will formalize the particle filter.

We will denote each particle $p^{(j)}$ as the jth 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   */
1 Initialize particle cohort by sampling $(p^{(1)}, p^{(2)}, ..., p^{(J)})$ from P_0

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

   /* Evolve   */
3   **for** $j = 1{:}J$ **do**
4          $X_t^{(j)} \leftarrow f_1(X_{t-1}^{(j)}, \theta^{(j)})$

   /* Weight   */
5   **for** $j = 1{:}J$ **do**
6          $w^{(j)} \leftarrow P(y_t|X_t^{(j)}, \theta^{(j)}) = f_2(X_t^{(j)}, \theta^{(j)})$

   /* Normalize   */
7   **for** $j = 1{:}J$ **do**
8          $w^{(j)} \leftarrow w^{(j)} / \sum_1^J w^{(j)}$

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

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

---

uses

see previous note about "th"

isn't this a bootstrap filter?

4   Particle Collapse

Not uncommonly, a situation may arise in which a single particle is assigned a normalized weight very close to 1 and all the other particles are assigned 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 together.

see previous note about subscripts like this

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.

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 posterior carried in the cohort of particles approaches maximum likelihood. However, this convergence may not be fast enough so 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

3

through the data will indeed force convergence of the process mean toward maximum likelihood, and the process variance toward 0.

6 IF2

probably spell out in the section header (maybe "Iterating Filtering 2 (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.

this seems a little loose ... "data cloning" (cite?) has a specific meaning. I think Ionides et al make an analogy to DC

~~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.

not sure i agree with this can you convince me?

but not sure they make any formal connection?

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 or geometric cooling, a decreasing step function, a combination of these, or though some other similar scheme.

multivariate Normal

The algorithm for IF2 can be seen in Algorithm 2.

4

Algorithm 2: IF2

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

/* Particle seeding distribution   */
2 $\Theta \leftarrow P_0$

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

   /* Pass perturbation   */
4 **for** $j = 1{:}J$ **do**
5 $\quad \lfloor \ p^{(j)} \sim h(\Theta^{(j)}, \sigma_m)$

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

7   **for** $j = 1{:}J$ **do**

      /* Iteration perturbation   */
8    $p^{(j)} \sim h(p^{(j)}, \sigma_m)$

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

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

     /* Normalize   */
11   **for** $j = 1{:}J$ **do**
12   $\quad \lfloor \ w^{(j)} \leftarrow w^{(j)} / \sum_1^J w^{(j)}$

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

   /* Collect particles for next pass   */
14 **for** $j = 1 : J$ **do**
15 $\quad \lfloor \ \Theta^{(j)} \leftarrow p^{(j)}$

/* 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

see previous comments about SIR

$$\frac{dS}{dt} = -\beta I S$$
$$\frac{dI}{dt} = \beta I S - rI \qquad (4)$$
$$\frac{dR}{dt} = rI$$

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.

see previous comments about level of technical detail; also, lots of material here repeats what's in the MCMC section

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

```
1    SIR ← function(Time, State, Pars) {
2
3        with(as.list(c(State, Pars)), {
4
5            B   ← R0*r/N        # calculate Beta
6            BSI ← B*S*I         # save product
7            rI  ← r*I           # save product
8
9            dS = -BSI           # change in Susceptible people
10           dI = BSI - rI       # change in Infected people
11           dR = rI             # change in Removed (recovered people)
12
13           return(list(c(dS, dI, dR)))
14
15       })
16
17   }
```

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

```
1    pars ← c(R0  = 3.0,   # new infected people per infected person
```

6

```
2              r   = 0.1,  # recovery rate
3              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

```
1    true_init_cond ← c(S = N - i_infec,
2                        I = i_infec,
3                        R = 0)
```

The ode() function is called as

```
1        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

```
1        set.seed(1001)  # set RNG seed for reproducibility
2        sigma ← 10      # observation error standard deviation
3        infec_counts_raw ← odeout[,3] + rnorm(101, 0, sigma)
4        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

```
1        plotdata ← data.frame(times=1:T,true=trueTraj,data=infec_counts)
2    curious why you used ggplot() here, qplot() in other chapter. colour mapping stuff is a little weird/wrong?
3        g ← ggplot(plotdata, aes(times)) +
4            geom_line(aes(y = true, colour = "True")) +
5            geom_point(aes(y = data, color = "Data")) +
6            labs(x = "Time", y = "Infection count", color = "") +
7            scale_color_brewer(palette="Paired") +
8            theme(panel.background = element_rect(fill = "#F0F0F0"))
```
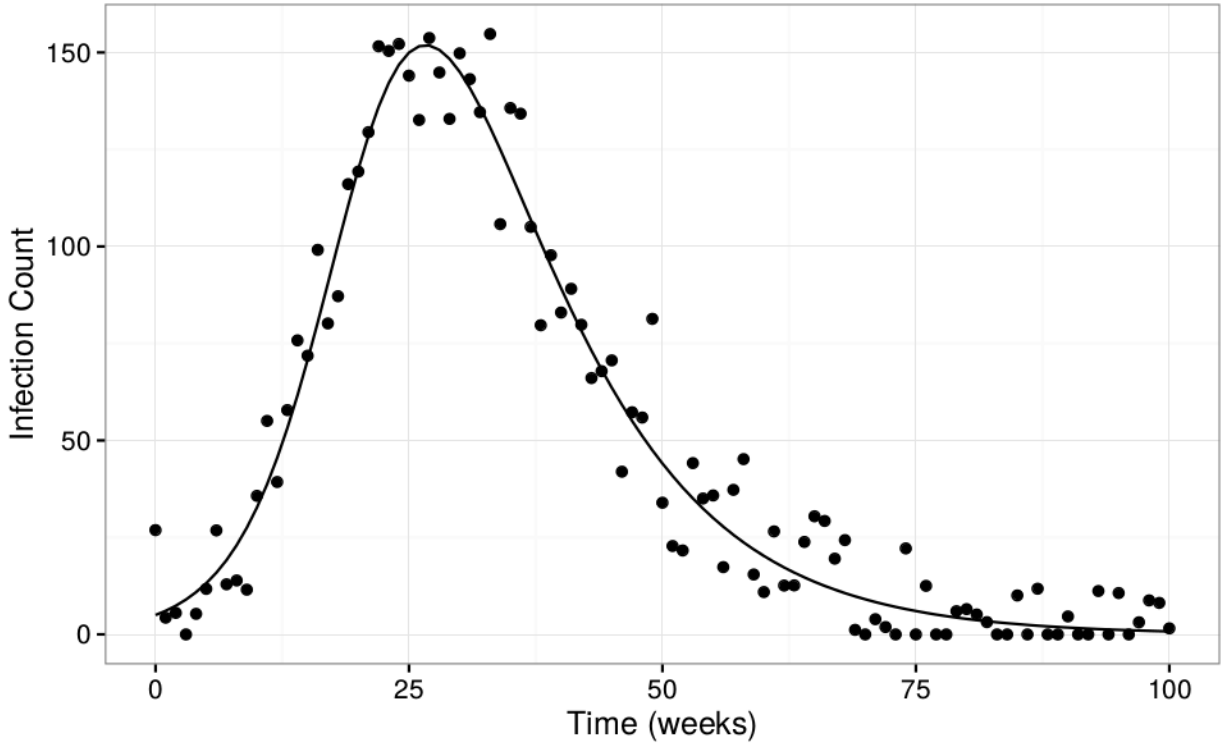
we obtain Figure 1.

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")
```

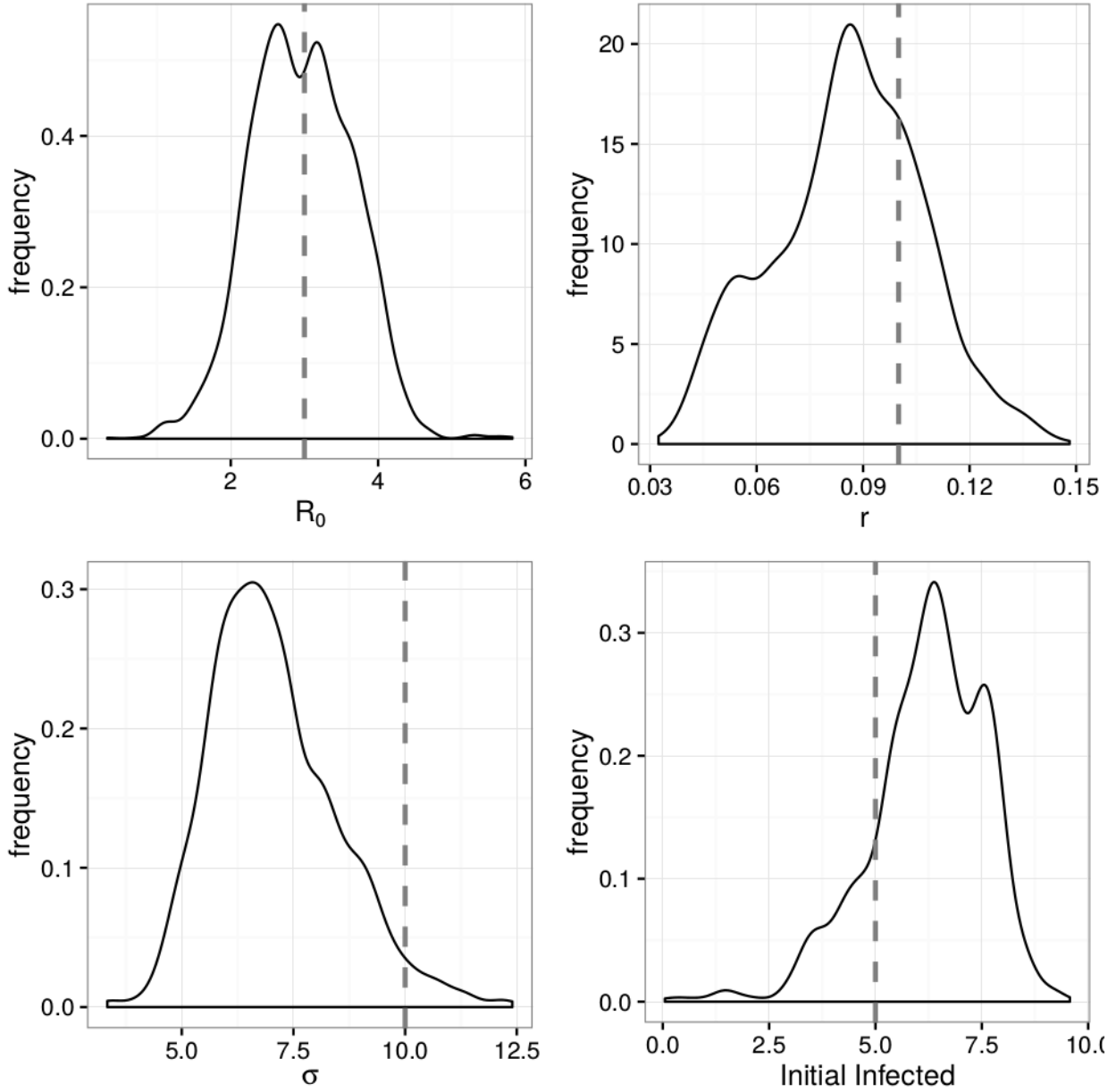The final kernel estimates for four of the key parameters are shown in Figure 2.

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

refer to our ongoing debate about whether the IF2 distributions have any inferential meaning; I think they're useful for *diagnostic* purposes but irrelevant for inferential purposes. You should either agree or disagree -- if you agree, clarify; if you disagree, make your argument clear.

Appendices

A Full R code

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

```
1  ##   Author: Dexter Barrows
2  ##   Github: dbarrows.github.io
3
4  library(deSolve)
5  library(ggplot2)
6  library(reshape2)
7  library(gridExtra)
8  library(Rcpp)
9
10 SIR ← function(Time, State, Pars) {
11
12     with(as.list(c(State, Pars)), {
13
14         B   ← R0*r/N
15         BSI ← B*S*I
16         rI  ← r*I
17
18         dS = -BSI
19         dI = BSI - rI
20         dR = rI
21
22         return(list(c(dS, dI, dR)))
23
24     })
25
26 }
27
28 T       ← 100
29 N       ← 500
30 sigma   ← 10
31 i_infec ← 5
32
33 ## Generate true trajecory and synthetic data
34 ##
35
36 true_init_cond ← c(S = N - i_infec,
37                    I = i_infec,
38                    R = 0)
39
40 true_pars ← c(R0 = 3.0,
41               r = 0.1,
42               N = 500.0)
43
44 odeout ← ode(true_init_cond, 0:T, SIR, true_pars)
```

10

```
45  trueTraj ← odeout[,3]
46
47  set.seed(1001)
48
49  infec_counts_raw ← odeout[,3] + rnorm(T+1, 0, sigma)
50  infec_counts     ← ifelse(infec_counts_raw < 0, 0, infec_counts_raw)
51
52  g ← qplot(0:T, odeout[,3], geom = "line", xlab = "Time (weeks)", ylab = "
        Infection Count") +
53      geom_point(aes(y = infec_counts)) +
54      theme_bw()
55
56  print(g)
57  ggsave(g, filename="dataplot.pdf", height=4, width=6.5)
58
59  ## Rcpp stuff
60  ##
61
62  sourceCpp(paste(getwd(),"d_if2.cpp",sep="/"))
63
64  paramdata ← data.frame(if2(infec_counts, T+1, N))
65  colnames(paramdata) ← c("R0", "r", "sigma", "Sinit", "Iinit", "Rinit")
66
67  ## Parameter density kernels
68  ##
69
70  R0points ← paramdata$R0
71  R0kernel ← qplot(R0points, geom = "density", xlab = expression(R[0]), ylab
        = "frequency") +
72          geom_vline(aes(xintercept=true_pars[["R0"]]), linetype="dashed",
                size=1, color="grey50") +
73          theme_bw()
74
75  print(R0kernel)
76  ggsave(R0kernel, filename="kernelR0.pdf", height=3, width=3.25)
77
78  rpoints ← paramdata$r
79  rkernel ← qplot(rpoints, geom = "density", xlab = "r", ylab = "frequency")
        +
80          geom_vline(aes(xintercept=true_pars[["r"]]), linetype="dashed",
                size=1, color="grey50") +
81          theme_bw()
82
83  print(rkernel)
84  ggsave(rkernel, filename="kernelr.pdf", height=3, width=3.25)
85
86  sigmapoints ← paramdata$sigma
87  sigmakernel ← qplot(sigmapoints, geom = "density", xlab = expression(sigma)
        , ylab = "frequency") +
88          geom_vline(aes(xintercept=sigma), linetype="dashed", size=1, color=
                "grey50") +
89          theme_bw()
90
91  print(sigmakernel)
```

```
92  ggsave(sigmakernel, filename="kernelsigma.pdf", height=3, width=3.25)
93
94  infecpoints <- paramdata$Iinit
95  infeckernel <- qplot(infecpoints, geom = "density", xlab = "Initial Infected
        ", ylab = "frequency") +
96          geom_vline(aes(xintercept=true_init_cond[['I']]), linetype="dashed"
                , size=1, color="grey50") +
97          theme_bw()
98
99  print(infeckernel)
100 ggsave(infeckernel, filename="kernelinfec.pdf", height=3, width=3.25)
101
102 # show grid
103 grid.arrange(R0kernel, rkernel, sigmakernel, infeckernel, ncol = 2, nrow =
        2)
104
105 pdf("if2kernels.pdf", height = 6.5, width = 6.5)
106 grid.arrange(R0kernel, rkernel, sigmakernel, infeckernel, ncol = 2, nrow =
        2)
107 dev.off()
108 #ggsave(filename="if2kernels.pdf", g2,  height=6.5, width=6.5)
```

B   Full C++ code

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

```
1  /*   Author: Dexter Barrows
2       Github: dbarrows.github.io
3
4       */
5
6  #include <stdio.h>
7  #include <math.h>
8  #include <sys/time.h>
9  #include <time.h>
10 #include <stdlib.h>
11 #include <string>
12 #include <cmath>
13 #include <cstdlib>
14 #include <fstream>
15
16 //#include "rand.h"
17 //#include "timer.h"
18
19 #define Treal    100        // time to simulate over
20 #define R0true   3.0        // infectiousness
21 #define rtrue    0.1        // recovery rate
22 #define Nreal    500.0      // population size
23 #define merr     10.0       // expected measurement error
24 #define I0       5.0        // Initial infected individuals
```

```
25
26  #include <Rcpp.h>
27  using namespace Rcpp;
28
29
30  struct Particle {
31      double R0;
32      double r;
33      double sigma;
34      double S;
35      double I;
36      double R;
37      double Sinit;
38      double Iinit;
39      double Rinit;
40  };
41
42  struct ParticleInfo {
43      double R0mean;       double R0sd;
44      double rmean;        double rsd;
45      double sigmamean;    double sigmasd;
46      double Sinitmean;    double Sinitsd;
47      double Iinitmean;    double Iinitsd;
48      double Rinitmean;    double Rinitsd;
49  };
50
51
52  int timeval_subtract (double *result, struct timeval *x, struct timeval *y)
       ;
53  int check_double(double x,double y);
54  void exp_euler_SIR(double h, double t0, double tn, int N, Particle *
       particle);
55  void copyParticle(Particle * dst, Particle * src);
56  void perturbParticles(Particle * particles, int N, int NP, int passnum,
       double coolrate);
57  bool isCollapsed(Particle * particles, int NP);
58  void particleDiagnostics(ParticleInfo * partInfo, Particle * particles, int
        NP);
59  NumericMatrix if2(NumericVector * data, int T, int N);
60  double randu();
61  double randn();
62
63  // [[Rcpp::export]]
64  NumericMatrix if2(NumericVector data, int T, int N) {
65
66      int    NP          = 2500;
67      int    nPasses     = 50;
68      double coolrate    = 0.975;
69
70      int    i_infec     = I0;
71
72      NumericMatrix paramdata(NP, 6);
73
74      srand(time(NULL));        // Seed PRNG with system time
```

in some future universe you might want
to make it easier to pass these parameters
in/control them from outside?

13

```c
    double w[NP];              // particle weights

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

    printf("Initializing particle states\n");

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

        double 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\n");


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

        printf("...%d / %d\n", pass, nPasses);

        perturbParticles(particles, N, NP, pass, coolrate);

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

```
128            particles[n].S = particles[n].Sinit;
129            particles[n].I = particles[n].Iinit;
130            particles[n].R = particles[n].Rinit;
131
132        }
133
134        // between-pass perturbations
135
136        for (int t = 1; t < T; t++) {
137
138            // between-iteration perturbations
139            perturbParticles(particles, N, NP, pass, coolrate);
140
141            // generate individual predictions and weight
142            for (int n = 0; n < NP; n++) {
143
144                exp_euler_SIR(1.0/10.0, 0.0, 1.0, N, &particles[n]);
145
146                double merr_par = particles[n].sigma;
147                double y_diff   = data[t] - particles[n].I;
148
149                w[n] = 1.0/(merr_par*sqrt(2.0*M_PI)) * exp( - y_diff*y_diff
                        / (2.0*merr_par*merr_par) );
150
151            }
152
153            // cumulative sum
154            for (int n = 1; n < NP; n++) {
155                w[n] += w[n-1];
156            }
157
158            // save particle states to resample from
159            for (int n = 0; n < NP; n++){
160                copyParticle(&particles_old[n], &particles[n]);
161            }
162
163            // resampling
164            for (int n = 0; n < NP; n++) {
165
166                double w_r = randu() * w[NP-1];
167                int i = 0;
168                while (w_r > w[i]) {
169                    i++;
170                }
171
172                // i is now the index to copy state from
173                copyParticle(&particles[n], &particles_old[i]);
174
175            }
176
177        }
178
179    }
180
```

```
181      ParticleInfo pInfo;
182      particleDiagnostics(&pInfo, particles, NP);
183
184      printf("Parameter results (mean | sd)\n");
185      printf("----------------------------\n");
186      printf("R0          %f %f\n", pInfo.R0mean, pInfo.R0sd);
187      printf("r           %f %f\n", pInfo.rmean, pInfo.rsd);
188      printf("sigma       %f %f\n", pInfo.sigmamean, pInfo.sigmasd);
189      printf("S_init   %f %f\n", pInfo.Sinitmean, pInfo.Sinitsd);
190      printf("I_init    %f %f\n", pInfo.Iinitmean, pInfo.Iinitsd);
191      printf("R_init    %f %f\n", pInfo.Rinitmean, pInfo.Rinitsd);
192
193      printf("\n");
194
195
196
197      // Get particle results to pass back to R
198
199      for (int n = 0; n < NP; n++) {
200
201          paramdata(n, 0) = particles[n].R0;
202          paramdata(n, 1) = particles[n].r;
203          paramdata(n, 2) = particles[n].sigma;
204          paramdata(n, 3) = particles[n].Sinit;
205          paramdata(n, 4) = particles[n].Iinit;
206          paramdata(n, 5) = particles[n].Rinit;
207
208      }
209
210      return paramdata;
211
212 }
213
214
215 /*  Use the Explicit Euler integration scheme to integrate SIR model
        forward in time
216      double h    - time step size
217      double t0   - start time
218      double tn   - stop time
219      double * y  - current system state; a three-component vector
            representing [S I R], susceptible-infected-recovered
220
221      */
222 void exp_euler_SIR(double h, double t0, double tn, int N, Particle *
        particle) {
223
224      int num_steps = floor( (tn-t0) / h );
225
226      double S = particle->S;
227      double I = particle->I;
228      double R = particle->R;
229
230      double R0   = particle->R0;
231      double r    = particle->r;
```

```
232    double B    = R0 * r / N;
233
234    for(int i = 0; i < num_steps; i++) {
235        // get derivatives
236        double dS = - B*S*I;
237        double dI = B*S*I - r*I;
238        double dR = r*I;
239        // step forward by h
240        S += h*dS;
241        I += h*dI;
242        R += h*dR;
243    }
244
245    particle->S = S;
246    particle->I = I;
247    particle->R = R;
248
249 }
250
251
252 /*  Particle pertubation function to be run between iterations and passes
253
254     */
255 void perturbParticles(Particle * particles, int N, int NP, int passnum,
         double coolrate) {
256
257    double coolcoef = pow(coolrate, passnum);
258
259    double spreadR0    = coolcoef * R0true / 10.0;
260    double spreadr     = coolcoef * rtrue  / 10.0;
261    double spreadsigma = coolcoef * merr   / 10.0;
262    double spreadIinit = coolcoef * I0     / 10.0;
263
264    double R0can, rcan, sigmacan, Iinitcan;
265
266    for (int n = 0; n < NP; n++) {
267
268        do {
269            R0can = particles[n].R0 + spreadR0*randn();
270        } while (R0can < 0);
271        particles[n].R0 = R0can;
272
273        do {
274            rcan = particles[n].r + spreadr*randn();
275        } while (rcan < 0);
276        particles[n].r = rcan;
277
278        do {
279            sigmacan = particles[n].sigma + spreadsigma*randn();
280        } while (sigmacan < 0);
281        particles[n].sigma = sigmacan;
282
283        do {
284            Iinitcan = particles[n].Iinit + spreadIinit*randn();
```

```
285        } while (Iinitcan < 0 || Iinitcan > 500);
286        particles[n].Iinit = Iinitcan;
287        particles[n].Sinit = N - Iinitcan;
288
289    }
290
291 }
292
293
294 /*  Convinience function for particle resampling process
295
296     */
297 void copyParticle(Particle * dst, Particle * src) {
298
299     dst->R0     = src->R0;
300     dst->r      = src->r;
301     dst->sigma  = src->sigma;
302     dst->S      = src->S;
303     dst->I      = src->I;
304     dst->R      = src->R;
305     dst->Sinit  = src->Sinit;
306     dst->Iinit  = src->Iinit;
307     dst->Rinit  = src->Rinit;
308
309 }
310
311
312 /*  Checks to see if particles are collapsed
313     This is done by checking if the standard deviations between the
            particles' parameter
314     values are significantly close to one another. Spread threshold may
            need to be tuned.
315
316     */
317 bool isCollapsed(Particle * particles, int NP) {
318
319     bool retVal;
320
321     double R0mean = 0, rmean = 0, sigmamean = 0, Sinitmean = 0, Iinitmean =
            0, Rinitmean = 0;
322
323     // means
324
325     for (int n = 0; n < NP; n++) {
326
327         R0mean      += particles[n].R0;
328         rmean       += particles[n].r;
329         sigmamean   += particles[n].sigma;
330         Sinitmean   += particles[n].Sinit;
331         Iinitmean   += particles[n].Iinit;
332         Rinitmean   += particles[n].Rinit;
333
334     }
335
```

```
336        R0mean        /= NP;
337        rmean         /= NP;
338        sigmamean     /= NP;
339        Sinitmean     /= NP;
340        Iinitmean     /= NP;
341        Rinitmean     /= NP;
342
343        double  R0sd = 0, rsd = 0, sigmasd = 0, Sinitsd = 0, Iinitsd = 0,
               Rinitsd = 0;
344
345        for (int n = 0; n < NP; n++) {
346
347            R0sd    += ( particles[n].R0 - R0mean ) * ( particles[n].R0 -
                   R0mean );
348            rsd     += ( particles[n].r - rmean ) * ( particles[n].r - rmean );
349            sigmasd += ( particles[n].sigma - sigmamean ) * ( particles[n].
                   sigma - sigmamean );
350            Sinitsd += ( particles[n].Sinit - Sinitmean ) * ( particles[n].
                   Sinit - Sinitmean );
351            Iinitsd += ( particles[n].Iinit - Iinitmean ) * ( particles[n].
                   Iinit - Iinitmean );
352            Rinitsd += ( particles[n].Rinit - Rinitmean ) * ( particles[n].
                   Rinit - Rinitmean );
353
354        }
355
356        R0sd         /= NP;
357        rsd          /= NP;
358        sigmasd      /= NP;
359        Sinitsd      /= NP;
360        Iinitsd      /= NP;
361        Rinitsd      /= NP;
362
363        if ( (R0sd + rsd + sigmasd) < 1e-5)
364            retVal = true;
365        else
366            retVal = false;
367
368        return retVal;
369
370 }
371
372 void particleDiagnostics(ParticleInfo * partInfo, Particle * particles, int
       NP) {
373
374        double  R0mean       = 0.0,
375                rmean        = 0.0,
376                sigmamean    = 0.0,
377                Sinitmean    = 0.0,
378                Iinitmean    = 0.0,
379                Rinitmean    = 0.0;
380
381        // means
382
```

```
383        for (int n = 0; n < NP; n++) {
384
385            R0mean       += particles[n].R0;
386            rmean        += particles[n].r;
387            sigmamean    += particles[n].sigma;
388            Sinitmean    += particles[n].Sinit;
389            Iinitmean    += particles[n].Iinit;
390            Rinitmean    += particles[n].Rinit;
391
392        }
393
394        R0mean       /= NP;
395        rmean        /= NP;
396        sigmamean    /= NP;
397        Sinitmean    /= NP;
398        Iinitmean    /= NP;
399        Rinitmean    /= NP;
400
401        // standard deviations
402
403        double  R0sd    = 0.0,
404                rsd     = 0.0,
405                sigmasd = 0.0,
406                Sinitsd = 0.0,
407                Iinitsd = 0.0,
408                Rinitsd = 0.0;
409
410        for (int n = 0; n < NP; n++) {
411
412            R0sd    += ( particles[n].R0 - R0mean ) * ( particles[n].R0 -
                    R0mean );
413            rsd     += ( particles[n].r - rmean ) * ( particles[n].r - rmean );
414            sigmasd += ( particles[n].sigma - sigmamean ) * ( particles[n].
                    sigma - sigmamean );
415            Sinitsd += ( particles[n].Sinit - Sinitmean ) * ( particles[n].
                    Sinit - Sinitmean );
416            Iinitsd += ( particles[n].Iinit - Iinitmean ) * ( particles[n].
                    Iinit - Iinitmean );
417            Rinitsd += ( particles[n].Rinit - Rinitmean ) * ( particles[n].
                    Rinit - Rinitmean );
418
419        }
420
421        R0sd         /= NP;
422        rsd          /= NP;
423        sigmasd      /= NP;
424        Sinitsd      /= NP;
425        Iinitsd      /= NP;
426        Rinitsd      /= NP;
427
428        partInfo->R0mean    = R0mean;
429        partInfo->R0sd      = R0sd;
430        partInfo->sigmamean = sigmamean;
431        partInfo->sigmasd   = sigmasd;
```

```
432      partInfo->rmean      = rmean;
433      partInfo->rsd        = rsd;
434      partInfo->Sinitmean  = Sinitmean;
435      partInfo->Sinitsd    = Sinitsd;
436      partInfo->Iinitmean  = Iinitmean;
437      partInfo->Iinitsd    = Iinitsd;
438      partInfo->Rinitmean  = Rinitmean;
439      partInfo->Rinitsd    = Rinitsd;
440
441  }
442
443  double randu() {
444
445      return (double) rand() / (double) RAND_MAX;
446   are you sure the standard system rand() is good enough? I'm not sure … http://bit.ly/1LcRV5W ?
447  }    (you should know enough to know how to find out whether it's OK …)
448
449
450  /*   Return a normally distributed random number with mean 0 and standard
         deviation 1
451       Uses the polar form of the Box-Muller transformation
452       From http://www.design.caltech.edu/erik/Misc/Gaussian.html
453       */
454  double randn() {
455
456      double x1, x2, w, y1;
457
458      do {
459          x1 = 2.0 * randu() - 1.0;
460          x2 = 2.0 * randu() - 1.0;
461          w = x1 * x1 + x2 * x2;
462      } while ( w >= 1.0 );
463
464      w = sqrt( (-2.0 * log( w ) ) / w );
465      y1 = x1 * w;
466
467      return y1;
468
469  }
```