Particle Filters

Dexter Barrows
February 28, 2016

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 will 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 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.

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

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

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 */

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

  /* 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

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

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.

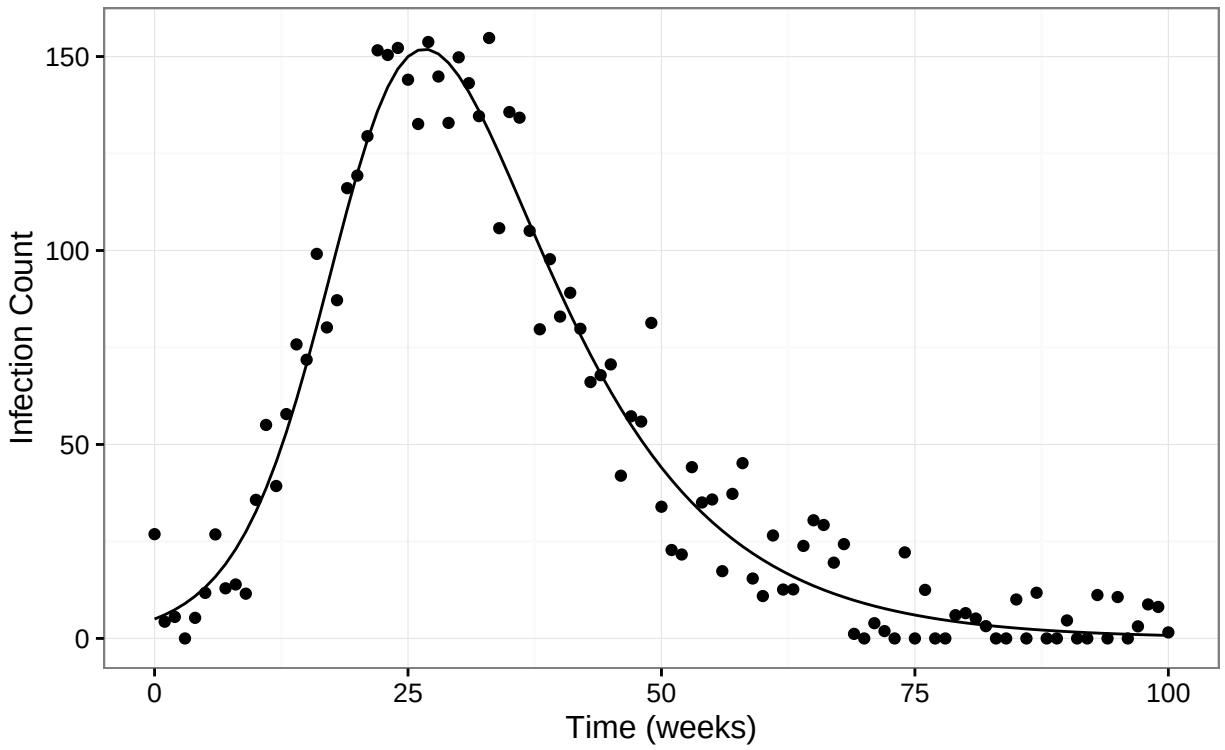

Figure 1: True SIR ODE solution infected counts, and with added observation noise

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.

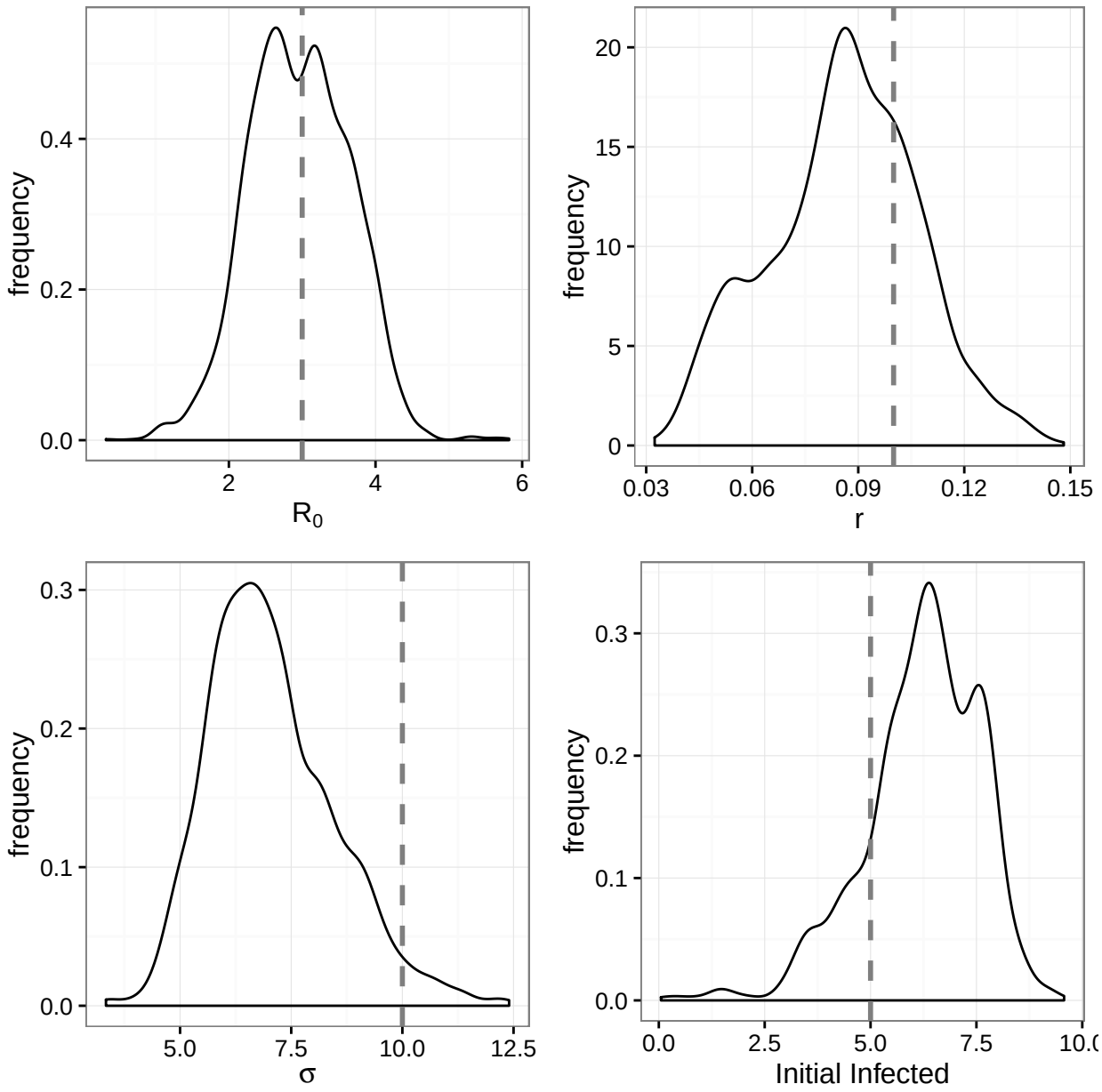

Figure 2: Kernel estimates for four essential system parameters. True values are indicated by solid vertical lines, sample means by dashed lines.

Appendices

A Full R code

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

```
##  Author: Dexter Barrows
##  Github: dbarrows.github.io

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

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

## 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, SIR, true_pars)
```

```
trueTraj ← odeout[,3]

set.seed(1001)

infec_counts_raw ← odeout[,3] + rnorm(T+1, 0, sigma)
infec_counts     ← ifelse(infec_counts_raw < 0, 0, infec_counts_raw)

g ← qplot(0:T, odeout[,3], geom = "line", xlab = "Time (weeks)", ylab = "
    Infection Count") +
    geom_point(aes(y = infec_counts)) +
    theme_bw()

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

## Rcpp stuff
##

sourceCpp(paste(getwd(),"d_if2.cpp",sep="/"))

paramdata ← data.frame(if2(infec_counts, T+1, N))
colnames(paramdata) ← c("R0", "r", "sigma", "Sinit", "Iinit", "Rinit")

## Parameter density kernels
##

R0points ← paramdata$R0
R0kernel ← qplot(R0points, geom = "density", xlab = expression(R[0]), ylab
    = "frequency") +
        geom_vline(aes(xintercept=true_pars[["R0"]]), linetype="dashed",
            size=1, color="grey50") +
        theme_bw()

print(R0kernel)
ggsave(R0kernel, filename="kernelR0.pdf", height=3, width=3.25)

rpoints ← paramdata$r
rkernel ← qplot(rpoints, geom = "density", xlab = "r", ylab = "frequency")
    +
        geom_vline(aes(xintercept=true_pars[["r"]]), linetype="dashed",
            size=1, color="grey50") +
        theme_bw()

print(rkernel)
ggsave(rkernel, filename="kernelr.pdf", height=3, width=3.25)

sigmapoints ← paramdata$sigma
sigmakernel ← qplot(sigmapoints, geom = "density", xlab = expression(sigma)
    , ylab = "frequency") +
        geom_vline(aes(xintercept=sigma), linetype="dashed", size=1, color=
            "grey50") +
        theme_bw()

print(sigmakernel)
```

```
ggsave(sigmakernel, filename="kernelsigma.pdf", height=3, width=3.25)

infecpoints ← paramdata$Iinit
infeckernel ← qplot(infecpoints, geom = "density", xlab = "Initial Infected
    ", ylab = "frequency") +
        geom_vline(aes(xintercept=true_init_cond[['I']]), linetype="dashed"
            , size=1, color="grey50") +
        theme_bw()

print(infeckernel)
ggsave(infeckernel, filename="kernelinfec.pdf", height=3, width=3.25)

# 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()
#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.

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

    */

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

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

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

```
75
76      double w[NP];           // particle weights
77
78      Particle particles[NP];     // particle estimates for current step
79      Particle particles_old[NP]; // intermediate particle states for
            resampling
80
81      printf("Initializing particle states\n");
82
83      // initialize particle parameter states (seeding)
84      for (int n = 0; n < NP; n++) {
85
86          double R0can, rcan, sigmacan, Iinitcan;
87
88          do {
89              R0can = R0true + R0true*randn();
90          } while (R0can < 0);
91          particles[n].R0 = R0can;
92
93          do {
94              rcan = rtrue + rtrue*randn();
95          } while (rcan < 0);
96          particles[n].r = rcan;
97
98          do {
99              sigmacan = merr + merr*randn();
100         } while (sigmacan < 0);
101         particles[n].sigma = sigmacan;
102
103         do {
104             Iinitcan = i_infec + i_infec*randn();
105         } while (Iinitcan < 0 || N < Iinitcan);
106         particles[n].Sinit = N - Iinitcan;
107         particles[n].Iinit = Iinitcan;
108         particles[n].Rinit = 0.0;
109
110     }
111
112     // START PASSES THROUGH DATA
113
114     printf("Starting filter\n");
115     printf("---------------\n");
116     printf("Pass\n");
117
118
119     for (int pass = 0; pass < nPasses; pass++) {
120
121         printf("...%d / %d\n", pass, nPasses);
122
123         perturbParticles(particles, N, NP, pass, coolrate);
124
125         // initialize particle system states
126         for (int n = 0; n < NP; n++) {
127
```

```
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;
```

```
    double B    = R0 * r / N;

    for(int i = 0; i < num_steps; i++) {
        // get derivatives
        double dS = - B*S*I;
        double dI = B*S*I - r*I;
        double 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 N, int NP, int passnum,
    double coolrate) {

    double coolcoef = pow(coolrate, passnum);

    double spreadR0     = coolcoef * R0true / 10.0;
    double spreadr      = coolcoef * rtrue  / 10.0;
    double spreadsigma  = coolcoef * merr   / 10.0;
    double spreadIinit  = coolcoef * I0     / 10.0;

    double 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();
```

```
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;
```

```
    partInfo->rmean     = rmean;
    partInfo->rsd       = rsd;
    partInfo->Sinitmean = Sinitmean;
    partInfo->Sinitsd   = Sinitsd;
    partInfo->Iinitmean = Iinitmean;
    partInfo->Iinitsd   = Iinitsd;
    partInfo->Rinitmean = Rinitmean;
    partInfo->Rinitsd   = Rinitsd;

}

double randu() {

    return (double) rand() / (double) RAND_MAX;

}


/*  Return a normally distributed random number with mean 0 and standard
    deviation 1
    Uses the polar form of the Box-Muller transformation
    From http://www.design.caltech.edu/erik/Misc/Gaussian.html
    */
double randn() {

    double x1, x2, w, y1;

    do {
        x1 = 2.0 * randu() - 1.0;
        x2 = 2.0 * randu() - 1.0;
        w = x1 * x1 + x2 * x2;
    } while ( w >= 1.0 );

    w = sqrt( (-2.0 * log( w ) ) / w );
    y1 = x1 * w;

    return y1;

}
```