S-map and SIRS

Dexter Barrows
March 18, 2016

1 S-maps

A family of forecasting methods that shy away from the mechanistic model-based approaches outlined in the previous sections have been developed by Sugihara (references) over the last several decades. As these methods do not include a mechanistic model in their forecasting process, they also do not attempt to perform parameter inference. Instead they attempt to reconstruct the underlying dynamical process as a weighted linear model from a time series.

One such method, the sequential locally weighted global linear maps (S-map), builds a global linear map model and uses it to produce forecasts directly. Despite relying on a linear mapping, the S-map does not assume the time series on which it is operating is the product of linear system dynamics, and in fact was developed to accommodate non-linear dynamics.

The S-map works by first constructing a time series embedding of length E, known as the library and denoted $\{\mathbf{x_i}\}$. Consider a time series of length T denoted $x_1, x_2, ..., x_T$. Each element in the time series with indices in the range $E, E+1, ..., T$ will have a corresponding entry in the library such that a given element x_t will correspond to a library vector of the form $\mathbf{x_i} = (x_t, x_{t-1}, ..., x_{t-E+1})$. Next, given a forecast length L (representing L time steps into the future), each library vector $\mathbf{x_i}$ is assigned a prediction from the time series $y_i = x_{t+L}$, where x_t is the first entry in $\mathbf{x_i}$. Finally, a forecast $\hat{y}_t$ for specified predictor vector $\mathbf{x_t}$ (usually from the library itself), is generated using an exponentially weighted function of the library $\{\mathbf{x_i}\}$, predictions $\{y_i\}$, and predictor vector $\mathbf{x_t}$.

This function is defined as follows:

First construct a matrix A and vector b defined as

$$\begin{aligned} A(i,j) &= w(||\mathbf{x_i} - \mathbf{x_t}||)\mathbf{x_i}(j) \\ b(i) &= w(||\mathbf{x_i} - \mathbf{x_t}||)y_i \end{aligned} \tag{1}$$

where i ranges over 1 to the length of the library, and j ranges over $[0, E]$. It should be noted that in the above equations and the ones that follow, $x_t(0) = 1$ to account for the

linear term in the map.

The weighting function w is defined as

$$w(d) = \exp\left(\frac{-\theta d}{\bar{d}}\right), \tag{2}$$

where d is the euclidean distance between the predictor vector and library vectors in Equation (1) and $\bar{d}$ is the average of these distances. We can then see that θ serves as a way to specify the appropriate level of penalization applied to poorly-matching library vectors – if θ is 0 all weights are the same (no penalization), and increasing θ increases the level of penalization.

Now we solve the system $Ac = b$ to obtain the linear weightings used in to generate the forecast according to

$$\hat{y}_t = \sum_{j=0}^{E} c_t(j)\mathbf{x_t}(j). \tag{3}$$

In this way we have produced a forecast value for a single time. This process can be repeated for a sequence of times $T+1, T+2, ...$ to project a time series into the future.

2 S-map Algorithm

The above description can be summarized in Algorithm [1].

Algorithm 1: S-map

/* Select a starting point */

Input : Time series $x_1, x_2, ..., x_T$, embedding dimension E, distance penalization θ, forecast length L, predictor vector $\mathbf{x_t}$

/* Construct library $\{\mathbf{x_i}\}$ */

for $i = E : T$ **do**
  $\mathbf{x_i} = (x_i, x_{i-1}, ..., x_{i-E-1})$

/* Construct mapping from library vectors to predictions */

for $i = 1 : (T_E + 1)$ **do**
  for $j = 1 : E$ **do**
    $A(i, j) = w(||\mathbf{x_i} - \mathbf{x_t}||)\mathbf{x_i}(j)$

for $i = 1 : (T_E + 1)$ **do**
  $b(i) = w(||\mathbf{x_i} - \mathbf{x_t}||)y_i$

/* Use SVD to solve the mapping system, Ac = b */

$SVD(Ac = b)$

/* Compute forecast */

$\hat{y}_t = \sum_{j=0}^{E} c_t(j)\mathbf{x_t}(j)$

/* Forecasted value in time series */

Output: Forecast $\hat{y}_t$

3 SIRS Model

In an epidemic or infectious disease context, the S-map algorithm will only really work on time series that appear cyclic. While there is nothing mechanically that prevents it from operating on a time series that do not appear cyclic, S-mapping requires a long time series in order to build a quality library. Without one the forecasting process would produce unreliable data.

With that in mind, the only fair way to compare the efficacy of s-mapping to IF2 or Hamiltonian MCMC is to generate data from a SIRS model with a seasonal component, and have all methods operate on the resulting time series.

The basic skeleton of the SIRS model is similar to the stochastic SIR model described

previously. The deterministic ODE component of the model is as follows.

$$\begin{aligned}\frac{dS}{dt} &= -\Gamma(t)\beta SI + \eta R \\ \frac{dI}{dt} &= \Gamma(t)\beta SI - \gamma I \\ \frac{dR}{dt} &= \gamma I - \eta R,\end{aligned} \tag{4}$$

There are two new features here. We have a re-susceptibility rate η through which people become able to be reinfected, and a seasonality factor Γ defined as

$$\Gamma(t) = \exp\left(2cos\left(\frac{2\pi}{365}t\right) - 2\right). \tag{5}$$

This function oscillates between 1 and e^{-4} (close to 0) and is meant to represent transmission damping during the off-season, for example summer for influenza. Further, it displays flatter troughs and sharper peaks to exaggerate its effect in peak season.

As before, β is allowed to walk restricted by a geometric mean, described by

$$\beta_{t+1} = \exp\left(\log(\beta_t) + \eta(\log(\bar{\beta}) - \log(\beta_t)) + \epsilon_t\right). \tag{6}$$

When simulated for the equivalent of 5 years (260 weeks), and adding noise drawn from $\mathcal{N}(0, \sigma)$we obtain Figure [1].

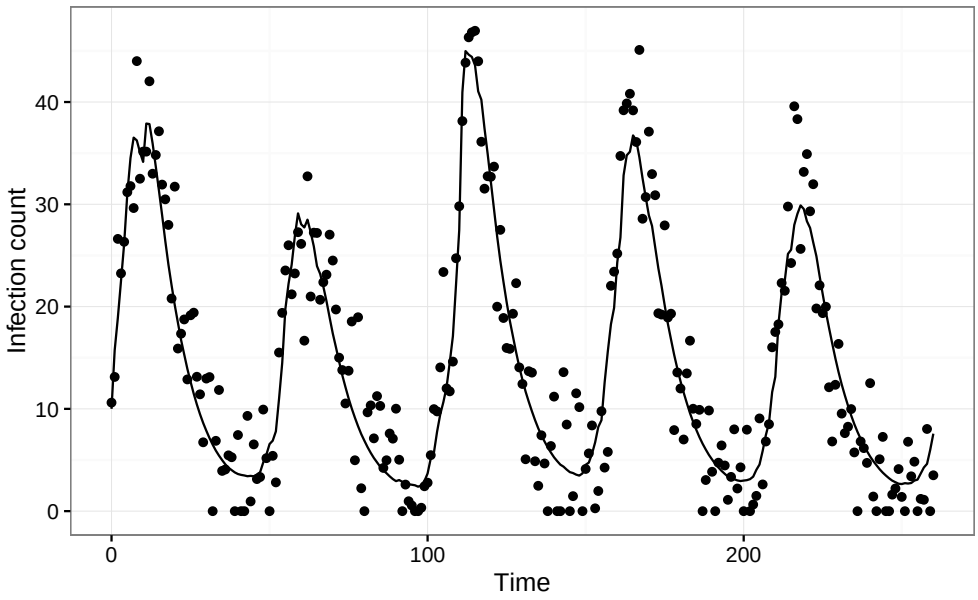

Figure 1: Five cycles generated by the SIRS function. The solid line the the true number of cases, dots show case counts with added observation noise. The Parameter values were $R0 = 3.0$, $\gamma = 0.1$, $\eta = 1$, $\sigma = 5$, and 10 initial cases.

We can see how the S-map can reconstruct the next cycle in the time series in Figure [2].

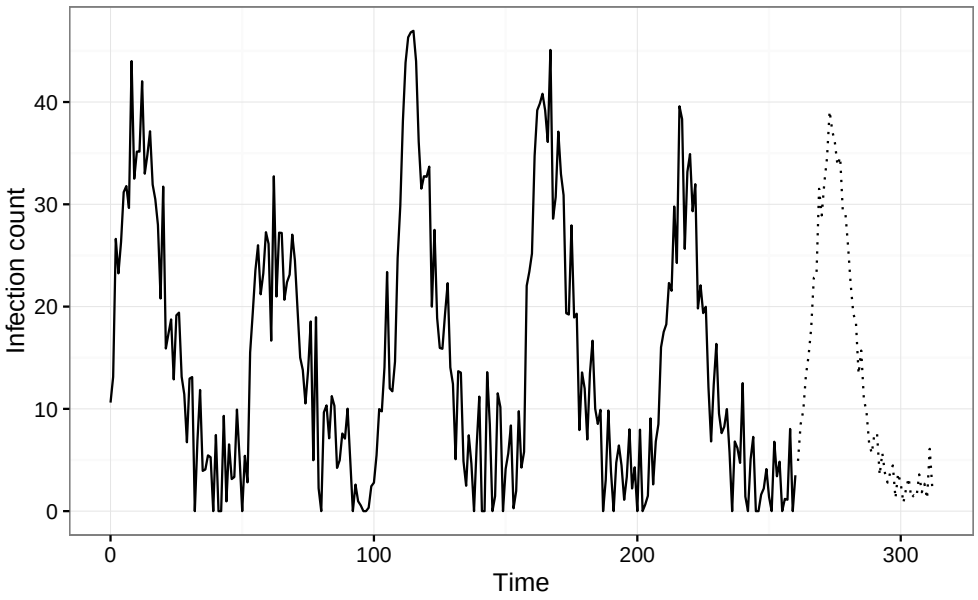

Figure 2: S-map applied to the data from the previous figure. The solid line shows the infection counts with observation noise form the previous plot, and the dotted line is the S-map forecast. Parameters chosen were $E = 14$ and $\theta = 3$.

The parameters used in the S-map algorithm to obtain the forecast used in Figure [2] were obtained using a grid search of potential parameters outlined in (Sugihara ref). The script is included in the appendices.

4 SIRS Model Forecasting

Naturally we wish to compare the efficacy of this comparatively simple technique against the more complex and more computationally taxing frameworks we have established to perform forecasting using IF2 and HMCMC.

To do this we generated a series of artificial time series of length 260 meant to represent 5 years of weekly incidence counts and used each method to forecast up to 2 years into the future. Our goal here was to determine how forecast error changed with forecast length.

The results of the simulation are shown in Figure [3].

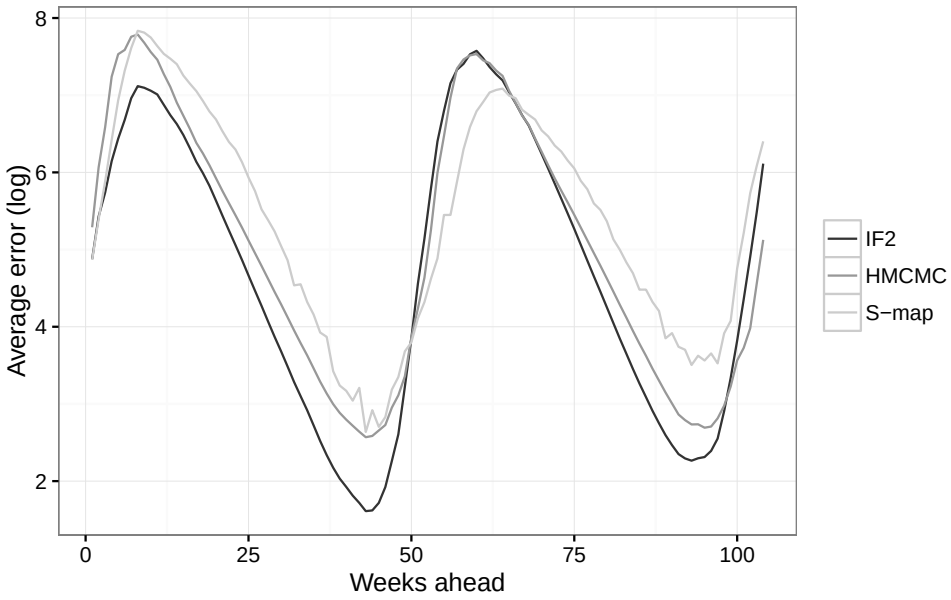

Figure 3: Error as a function of forecast length.

Interestingly, all methods produce roughly the same result, which is to say the spike in each outbreak cycle are difficult to accurately predict. IF2 produces better results than either HMCMC and the S-map for the majority of forecast lengths, with the S-map producing the poorest results with the exception of the second rise in infection rates in which it outperforms the other methods.

While the S-map may not provide the same fidelity or forecast as IF2 or HMCMC, it shines when it comes to running time. Figure [4] shows the running times over 20 simulations.

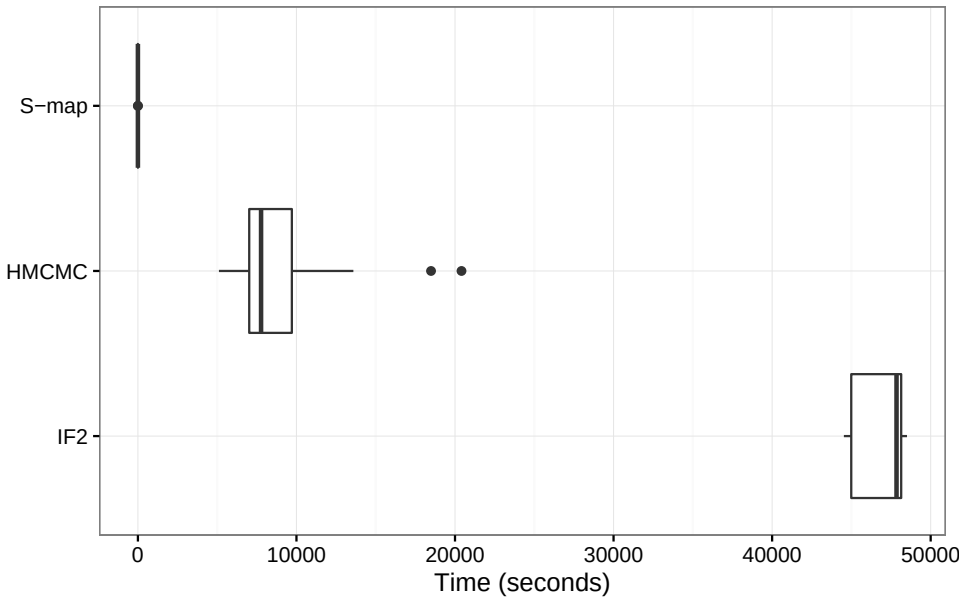

Figure 4: Runtimes for producing SIRS forecasts. The box shows the middle 50th quantile, the bold line is the median, and the dots are outliers.

It is clear from the above figure that the S-map running times are minute compared to the other methods, but to emphasize the degree: The average running time for the S-map is about 1.49×10^{-1} seconds, for IF2 it is about 4.70×10^{4}, and for HMCMC it is about 9.20×10^{3}. This is a speed-up of over 316,000x compared to IF2 and over 61,800x compared to HMCMC.

Appendices

A SIRS R Function Code

R code to simulate the outlines SIRS function.

```
StocSIRS ← function(y, pars, T, steps) {

  out ← matrix(NA, nrow = (T+1), ncol = 4)

  R0 ← pars[['R0']]
  r ← pars[['r']]
  N ← pars[['N']]
  eta ← pars[['eta']]
  berr ← pars[['berr']]
    re ← pars[['re']]

  S ← y[['S']]
  I ← y[['I']]
  R ← y[['R']]

  B0 ← R0 * r / N
  B ← B0

  out[1,] ← c(S,I,R,B)

  h ← 1 / steps

  for ( i in 1:(T*steps) ) {

        #Bfac ← 1/2 - cos((2*pi/365)*i)/2
        Bfac ← exp(2*cos((2*pi/365)*i) - 2)

    B ← exp( log(B) + eta*(log(B0) - log(B)) + rnorm(1, 0, berr) )

    BSI ← Bfac*B*S*I
    rI ← r*I
        reR ← re*R

    dS ← -BSI + reR
    dI ← BSI - rI
    dR ← rI - reR

    S ← S + h*dS  #newInf
    I ← I + h*dI  #newInf - h*dR
    R ← R + h*dR  #h*dR

    if (i %% steps == 0)
      out[i/steps+1,] ← c(S,I,R,B)

```

```
  }

  colnames(out) ← c("S","I","R","B")
  return(out)

}

### suggested parameters
#
# T     ← 200
# i_infec   ← 10
# steps   ← 7
# N     ← 500
# sigma   ← 5
#
# pars ← c(R0 = 3.0,  # new infected people per infected person
#            r = 0.1,  # recovery rate
#        N = 500,    # population size
#        eta = 0.5,  # geometric random walk
#        berr = 0.5, # Beta geometric walk noise
#            re = 1)   # resuceptibility rate
```

B SMAP Code

This code implements an SMAP function on a user-provided time series.

```
library(pracma)

smap ← function(data, E, theta, stepsAhead) {

    # construct library
    tseries ← as.vector(data)
    liblen  ← length(tseries) - E + 1 - stepsAhead
    lib     ← matrix(NA, liblen, E)

    for (i in 1:E) {
        lib[,i] ← tseries[(E-i+1):(liblen+E-i)]
    }

    # predict from the last index
    tslen ← length(tseries)
    predictee ← rev(t(as.matrix(tseries[(tslen-E+1):tslen])))
    predictions ← numeric(stepsAhead)

    #allPredictees ← matrix(NA, stepsAhead, E)

    # for each prediction index (number of steps ahead)
    for(i in 1:stepsAhead) {

        # set up weight calculation
```

```
        predmat <- repmat(predictee, liblen, 1)
        distances <- sqrt( rowSums( abs(lib - predmat)^2 ) )
        meanDist <- mean(distances)

        # calculate weights
        weights <- exp( - (theta * distances) / meanDist )

        # construct A, B

        preds <- tseries[(E+i):(liblen+E+i-1)]

        A <- cbind( rep(1.0, liblen), lib ) * repmat(as.matrix(weights), 1,
            E+1)
        B <- as.matrix(preds * weights)

        # solve system for C

        Asvd <- svd(A)
        C <- Asvd$v %*% diag(1/Asvd$d) %*% t(Asvd$u) %*% B

        # get prediction

        predsum <- sum(C * c(1,predictee))

        # save

        predictions[i] <- predsum

        # next predictee

        #predictee <- c( predsum, predictee[-E] )
        #allPredictees[i,] <- predictee

    }

    return(predictions)

}
```

C SMAP Parameter Optimization Code

This code determines the optimal parameter values to be used by the S-map algorithm.

```
library(deSolve)
library(ggplot2)
library(RColorBrewer)
library(pracma)

set.seed(1010)

```

```
## external files
##
stoc_sirs_file  ← paste(getwd(), "../sir-functions", "StocSIRS.r", sep = "/
    ")
smap_file     ← paste(getwd(), "smap.r", sep = "/")
source(stoc_sirs_file)
source(smap_file)



## parameters
##
T     ← 6*52
Tlim  ← T - 52
i_infec ← 10
steps   ← 7
N     ← 500
sigma   ← 5

true_pars ← c(  R0 = 3.0,   # new infected people per infected person
                r = 0.1,  # recovery rate
           N = 500,    # population size
           eta = 0.5,  # geometric random walk
           berr = 0.5, # Beta geometric walk noise
                re = 1)   # resuceptibility rate

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

## trial parameter values to check.options
##
Elist ← 1:20
thetalist ← 10*exp(-(seq(0,9.5,0.5)))
nTrials ← 100

ssemat ← matrix(NA, 20, 20)

for (i in 1:length(Elist)) {
  for (j in 1:length(thetalist)) {

    ssemean ← 0

    for (k in 1:nTrials) {

      E ← Elist[i]
      theta ← thetalist[j]

      ## get true trajectory
      ##
      sdeout ← StocSIRS(true_init_cond, true_pars, T, steps)

      ## perturb to get data
      ##
```

```
      infec_counts_raw ← sdeout[1:(Tlim+1),'I'] + rnorm(Tlim+1,0,sigma)
      infec_counts   ← ifelse(infec_counts_raw < 0, 0, infec_counts_raw)

      predictions ← smap(infec_counts, E, theta, 52)

      err ← sdeout[(Tlim+2):dim(sdeout)[1],'I'] - predictions
      sse ← sum(err^2)

      ssemean ← ssemean + (sse / nTrials)

    }

    ssemat[i,j] ← ssemean


  }
}

quartz()
image(-ssemat)
quartz()
filled.contour(-ssemat)

#print(ssemat)
#cms ← colMeans(ssemat)
#rms ← rowMeans(ssemat)

#Emin ← Elist[which.min(rms)]
#thetamin ← thetalist[which.min(cms)]
#print(Emin)
#print(thetamin)

mininds ← which(ssemat==min(ssemat),arr.ind=TRUE)

Emin ← Elist[mininds['row']]
thetamin ← thetalist[mininds['col']]

print(Emin)
print(thetamin)
```

D RStan SIRS Code

This code implements a periodic SIRS model in Rstan.

```
data {

    int     <lower=1>   T;      // total integration steps
    real                y[T];   // observed number of cases
    int     <lower=1>   N;      // population size
    real                h;      // step size
```

```
7
8 }
9
10 parameters {
11
12     real <lower=0, upper=10>         R0;      // R0
13     real <lower=0, upper=10>         r;       // recovery rate
14     real <lower=0, upper=10>         re;      // resusceptibility rate
15     real <lower=0, upper=20>         sigma;   // observation error
16     real <lower=0, upper=30>         Iinit;     // initial infected
17     real <lower=0, upper=1>          eta;     // geometric walk attraction
           strength
18     real <lower=0, upper=1>          berr;    // beta walk noise
19     real <lower=-1.5, upper=1.5>     Bnoise[T];   // Beta vector
20
21 }
22
23 //transformed parameters {
24 //    real B0 ← R0 * r / N;
25 //}
26
27 model {
28
29     real S[T];
30     real I[T];
31     real R[T];
32     real B[T];
33     real B0;
34
35     real pi;
36     real Bfac;
37
38     pi ← 3.1415926535;
39
40     B0 ← R0 * r / N;
41
42     B[1] ← B0;
43
44     S[1] ← N - Iinit;
45     I[1] ← Iinit;
46     R[1] ← 0.0;
47
48     for (t in 2:T) {
49
50         Bnoise[t] ~ normal(0,berr);
51         Bfac ← exp(2*cos((2*pi/365)*t) - 2);
52         B[t] ← exp( log(B0) + eta * ( log(B[t-1]) - log(B0) ) + Bnoise[t] )
               ;
53
54         S[t] ← S[t-1] + h*( - Bfac*B[t]*S[t-1]*I[t-1] + re*R[t-1] );
55         I[t] ← I[t-1] + h*( Bfac*B[t]*S[t-1]*I[t-1] - I[t-1]*r );
56         R[t] ← R[t-1] + h*( I[t-1]*r - re*R[t-1] );
57
58         if (y[t] > 0) {
```

```
            y[t] ~ normal( I[t], sigma );
        }

    }

    R0      ~ lognormal(1,1);
    r       ~ lognormal(1,1);
    sigma   ~ lognormal(1,1);
    re      ~ lognormal(1,1);
    Iinit   ~ normal(y[1], sigma);

}
```

E IF2 SIRS Code

This code implements a periodic SIRS model using IF2 in C++.

```
/*  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 retrue      0.05        // resusceptibility rate
#define Nreal       500.0       // population size
#define etatrue     0.5         // real drift attraction strength
#define berrtrue    0.5         // real beta drift noise
#define merr        5.0         // expected measurement error
#define I0          5.0         // Initial infected individuals

#define PSC         0.5         // scale factor for more sensitive
    parameters

#include <Rcpp.h>
using namespace Rcpp;
```

```
33
34 struct State {
35     double S;
36     double I;
37     double R;
38 };
39
40 struct Particle {
41     double R0;
42     double r;
43     double re;
44     double sigma;
45     double eta;
46     double berr;
47     double B;
48     double S;
49     double I;
50     double R;
51     double Sinit;
52     double Iinit;
53     double Rinit;
54 };
55
56 struct ParticleInfo {
57     double R0mean;      double R0sd;
58     double rmean;       double rsd;
59     double remean;      double resd;
60     double sigmamean;   double sigmasd;
61     double etamean;     double etasd;
62     double berrmean;    double berrsd;
63     double Sinitmean;   double Sinitsd;
64     double Iinitmean;   double Iinitsd;
65     double Rinitmean;   double Rinitsd;
66 };
67
68
69 int timeval_subtract (double *result, struct timeval *x, struct timeval *y)
       ;
70 int check_double(double x,double y);
71 void exp_euler_SIRS(double h, double t0, double tn, int N, Particle *
       particle);
72 void copyParticle(Particle * dst, Particle * src);
73 void perturbParticles(Particle * particles, int N, int NP, int passnum,
       double coolrate);
74 void particleDiagnostics(ParticleInfo * partInfo, Particle * particles, int
        NP);
75 void getStateMeans(State * state, Particle* particles, int NP);
76 NumericMatrix if2(NumericVector * data, int T, int N);
77 double randu();
78 double randn();
79
80 // [[Rcpp::export]]
81 Rcpp::List if2_sirs(NumericVector data, int T, int N, int NP, int nPasses,
       double coolrate) {
```

```
82
83      int npar = 9;
84
85      NumericMatrix paramdata(NP, npar);
86      NumericMatrix means(nPasses, npar);
87      NumericMatrix sds(nPasses, npar);
88      NumericMatrix statemeans(T, 3);
89      NumericMatrix statedata(NP, 4);
90
91      srand(time(NULL));      // Seed PRNG with system time
92
93      double w[NP];           // particle weights
94
95      Particle particles[NP];     // particle estimates for current step
96      Particle particles_old[NP]; // intermediate particle states for
            resampling
97
98      printf("Initializing particle states\n");
99
100     // initialize particle parameter states (seeding)
101     for (int n = 0; n < NP; n++) {
102
103         double R0can, rcan, recan, sigmacan, Iinitcan, etacan, berrcan;
104
105         do {
106             R0can = R0true + R0true*randn();
107         } while (R0can < 0);
108         particles[n].R0 = R0can;
109
110         do {
111             rcan = rtrue + rtrue*randn();
112         } while (rcan < 0);
113         particles[n].r = rcan;
114
115         do {
116             recan = retrue + retrue*randn();
117         } while (recan < 0);
118         particles[n].re = recan;
119
120         particles[n].B = (double) R0can * rcan / N;
121
122         do {
123             sigmacan = merr + merr*randn();
124         } while (sigmacan < 0);
125         particles[n].sigma = sigmacan;
126
127         do {
128             etacan = etatrue + PSC*etatrue*randn();
129         } while (etacan < 0 || etacan > 1);
130         particles[n].eta = etacan;
131
132         do {
133             berrcan = berrtrue + PSC*berrtrue*randn();
134         } while (berrcan < 0);
```

```
135         particles[n].berr = berrcan;
136
137         do {
138             Iinitcan = I0 + I0*randn();
139         } while (Iinitcan < 0 || N < Iinitcan);
140         particles[n].Sinit = N - Iinitcan;
141         particles[n].Iinit = Iinitcan;
142         particles[n].Rinit = 0.0;
143
144     }
145
146     // START PASSES THROUGH DATA
147
148     printf("Starting filter\n");
149     printf("---------------\n");
150     printf("Pass\n");
151
152
153     for (int pass = 0; pass < nPasses; pass++) {
154
155         printf("...%d / %d\n", pass, nPasses);
156
157         // reset particle system evolution states
158         for (int n = 0; n < NP; n++) {
159
160             particles[n].S = particles[n].Sinit;
161             particles[n].I = particles[n].Iinit;
162             particles[n].R = particles[n].Rinit;
163             particles[n].B = (double) particles[n].R0 * particles[n].r / N;
164
165         }
166
167         if (pass == (nPasses-1)) {
168             State sMeans;
169             getStateMeans(&sMeans, particles, NP);
170             statemeans(0,0) = sMeans.S;
171             statemeans(0,1) = sMeans.I;
172             statemeans(0,2) = sMeans.R;
173         }
174
175         for (int t = 1; t < T; t++) {
176
177             // generate individual predictions and weight
178             for (int n = 0; n < NP; n++) {
179
180                 exp_euler_SIRS(1.0/7.0, (double) t-1, (double) t, N, &
                        particles[n]);
181
182                 double merr_par = particles[n].sigma;
183                 double y_diff   = data[t] - particles[n].I;
184
185                 w[n] = 1.0/(merr_par*sqrt(2.0*M_PI)) * exp( - y_diff*y_diff
                        / (2.0*merr_par*merr_par) );
186
```

```
            }

            // 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++) {

                double 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]);

            }

            // between-iteration perturbations, not after last time step
            if (t < (T-1))
                perturbParticles(particles, N, NP, pass, coolrate);

            if (pass == (nPasses-1)) {
                State sMeans;
                getStateMeans(&sMeans, particles, NP);
                statemeans(t,0) = sMeans.S;
                statemeans(t,1) = sMeans.I;
                statemeans(t,2) = sMeans.R;
            }

        }

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

        means(pass, 0) = pInfo.R0mean;
        means(pass, 1) = pInfo.rmean;
        means(pass, 2) = pInfo.remean;
        means(pass, 3) = pInfo.sigmamean;
        means(pass, 4) = pInfo.etamean;
        means(pass, 5) = pInfo.berrmean;
        means(pass, 6) = pInfo.Sinitmean;
        means(pass, 7) = pInfo.Iinitmean;
        means(pass, 8) = pInfo.Rinitmean;

        sds(pass, 0) = pInfo.R0sd;
```

```
241         sds(pass, 1) = pInfo.rsd;
242         sds(pass, 2) = pInfo.resd;
243         sds(pass, 3) = pInfo.sigmasd;
244         sds(pass, 4) = pInfo.etasd;
245         sds(pass, 5) = pInfo.berrsd;
246         sds(pass, 6) = pInfo.Sinitsd;
247         sds(pass, 7) = pInfo.Iinitsd;
248         sds(pass, 8) = pInfo.Rinitsd;
249 
250         // between-pass perturbations, not after last pass
251         if (pass < (nPasses + 1))
252             perturbParticles(particles, N, NP, pass, coolrate);
253 
254     }
255 
256     ParticleInfo pInfo;
257     particleDiagnostics(&pInfo, particles, NP);
258 
259     printf("Parameter results (mean | sd)\n");
260     printf("-----------------------------\n");
261     printf("R0        %f %f\n", pInfo.R0mean, pInfo.R0sd);
262     printf("r         %f %f\n", pInfo.rmean, pInfo.rsd);
263     printf("re        %f %f\n", pInfo.remean, pInfo.resd);
264     printf("sigma     %f %f\n", pInfo.sigmamean, pInfo.sigmasd);
265     printf("eta       %f %f\n", pInfo.etamean, pInfo.etasd);
266     printf("berr    %f %f\n", pInfo.berrmean, pInfo.berrsd);
267     printf("S_init  %f %f\n", pInfo.Sinitmean, pInfo.Sinitsd);
268     printf("I_init    %f %f\n", pInfo.Iinitmean, pInfo.Iinitsd);
269     printf("R_init    %f %f\n", pInfo.Rinitmean, pInfo.Rinitsd);
270 
271     printf("\n");
272 
273 
274 
275     // Get particle results to pass back to R
276 
277     for (int n = 0; n < NP; n++) {
278 
279         paramdata(n, 0) = particles[n].R0;
280         paramdata(n, 1) = particles[n].r;
281         paramdata(n, 2) = particles[n].re;
282         paramdata(n, 3) = particles[n].sigma;
283         paramdata(n, 4) = particles[n].eta;
284         paramdata(n, 5) = particles[n].berr;
285         paramdata(n, 6) = particles[n].Sinit;
286         paramdata(n, 7) = particles[n].Iinit;
287         paramdata(n, 8) = particles[n].Rinit;
288 
289     }
290 
291     for (int n = 0; n < NP; n++) {
292 
293         statedata(n, 0) = particles[n].S;
294         statedata(n, 1) = particles[n].I;
```

```
295         statedata(n, 2) = particles[n].R;
296         statedata(n, 3) = particles[n].B;
297 
298     }
299 
300 
301 
302     return Rcpp::List::create(  Rcpp::Named("paramdata") = paramdata,
303                                 Rcpp::Named("means") = means,
304                                 Rcpp::Named("statemeans") = statemeans,
305                                 Rcpp::Named("statedata") = statedata,
306                                 Rcpp::Named("sds") = sds);
307 
308 }
309 
310 
311 /*  Use the Explicit Euler integration scheme to integrate SIR model
        forward in time
312     double h    - time step size
313     double t0   - start time
314     double tn   - stop time
315     double * y  - current system state; a three-component vector
           representing [S I R], susceptible-infected-recovered
316 
317     */
318 void exp_euler_SIRS(double h, double t0, double tn, int N, Particle *
        particle) {
319 
320     int num_steps = floor( (tn-t0) / h );
321 
322     double S = particle->S;
323     double I = particle->I;
324     double R = particle->R;
325 
326     double R0   = particle->R0;
327     double r    = particle->r;
328     double re   = particle->re;
329     double B0   = R0 * r / N;
330     double eta  = particle->eta;
331     double berr  = particle->berr;
332 
333     double B = particle->B;
334 
335     for(int i = 0; i < num_steps; i++) {
336 
337         //double Bfac = 0.5 - 0.95*cos( (2.0*M_PI/365)*(t0*num_steps+i) )
               /2.0;
338         double Bfac = exp(2*cos((2*M_PI/365)*(t0*num_steps+i)) - 2);
339         B = exp( log(B) + eta*(log(B0) - log(B)) + berr*randn() );
340 
341         double BSI = Bfac*B*S*I;
342         double rI  = r*I;
343         double reR = re*R;
344 
```

```
345         // get derivatives
346         double dS = - BSI + reR;
347         double dI = BSI - rI;
348         double dR = rI - reR;
349 
350         // step forward by h
351         S += h*dS;
352         I += h*dI;
353         R += h*dR;
354 
355     }
356 
357     particle->S = S;
358     particle->I = I;
359     particle->R = R;
360     particle->B = B;
361 
362 }
363 
364 
365 /*  Particle pertubation function to be run between iterations and passes
366 
367     */
368 void perturbParticles(Particle * particles, int N, int NP, int passnum,
        double coolrate) {
369 
370     //double coolcoef = exp( - (double) passnum / coolrate );
371     double coolcoef = pow(coolrate, passnum);
372 
373 
374     double spreadR0     = coolcoef * R0true / 10.0;
375     double spreadr      = coolcoef * rtrue / 10.0;
376     double spreadre     = coolcoef * retrue / 10.0;
377     double spreadsigma  = coolcoef * merr / 10.0;
378     double spreadIinit  = coolcoef * I0 / 10.0;
379     double spreadeta    = coolcoef * etatrue / 10.0;
380     double spreadberr   = coolcoef * berrtrue / 10.0;
381 
382 
383     double R0can, rcan, recan, sigmacan, Iinitcan, etacan, berrcan;
384 
385     for (int n = 0; n < NP; n++) {
386 
387         do {
388             R0can = particles[n].R0 + spreadR0*randn();
389         } while (R0can < 0);
390         particles[n].R0 = R0can;
391 
392         do {
393             rcan = particles[n].r + spreadr*randn();
394         } while (rcan < 0);
395         particles[n].r = rcan;
396 
397         do {
```

```
398             recan = particles[n].re + spreadre*randn();
399         } while (recan < 0);
400         particles[n].re = recan;
401 
402         do {
403             sigmacan = particles[n].sigma + spreadsigma*randn();
404         } while (sigmacan < 0);
405         particles[n].sigma = sigmacan;
406 
407         do {
408             etacan = particles[n].eta + PSC*spreadeta*randn();
409         } while (etacan < 0 || etacan > 1);
410         particles[n].eta = etacan;
411 
412         do {
413             berrcan = particles[n].berr + PSC*spreadberr*randn();
414         } while (berrcan < 0);
415         particles[n].berr = berrcan;
416 
417         do {
418             Iinitcan = particles[n].Iinit + spreadIinit*randn();
419         } while (Iinitcan < 0 || Iinitcan > 500);
420         particles[n].Iinit = Iinitcan;
421         particles[n].Sinit = N - Iinitcan;
422 
423     }
424 
425 }
426 
427 
428 /*  Convinience function for particle resampling process
429 
430     */
431 void copyParticle(Particle * dst, Particle * src) {
432 
433     dst->R0     = src->R0;
434     dst->r      = src->r;
435     dst->re     = src->re;
436     dst->sigma  = src->sigma;
437     dst->eta    = src->eta;
438     dst->berr   = src->berr;
439     dst->B      = src->B;
440     dst->S      = src->S;
441     dst->I      = src->I;
442     dst->R      = src->R;
443     dst->Sinit  = src->Sinit;
444     dst->Iinit  = src->Iinit;
445     dst->Rinit  = src->Rinit;
446 
447 }
448 
449 void particleDiagnostics(ParticleInfo * partInfo, Particle * particles, int
        NP) {
450 
```

```
451     double  R0mean      = 0.0,
452             rmean       = 0.0,
453             remean      = 0.0,
454             sigmamean   = 0.0,
455             etamean     = 0.0,
456             berrmean    = 0.0,
457             Sinitmean   = 0.0,
458             Iinitmean   = 0.0,
459             Rinitmean   = 0.0;
460
461     // means
462
463     for (int n = 0; n < NP; n++) {
464
465         R0mean      += particles[n].R0;
466         rmean       += particles[n].r;
467         remean      += particles[n].re;
468         etamean     += particles[n].eta,
469         berrmean    += particles[n].berr,
470         sigmamean   += particles[n].sigma;
471         Sinitmean   += particles[n].Sinit;
472         Iinitmean   += particles[n].Iinit;
473         Rinitmean   += particles[n].Rinit;
474
475     }
476
477     R0mean      /= NP;
478     rmean       /= NP;
479     remean      /= NP;
480     sigmamean   /= NP;
481     etamean     /= NP;
482     berrmean    /= NP;
483     Sinitmean   /= NP;
484     Iinitmean   /= NP;
485     Rinitmean   /= NP;
486
487     // standard deviations
488
489     double  R0sd    = 0.0,
490             rsd     = 0.0,
491             resd    = 0.0,
492             sigmasd = 0.0,
493             etasd   = 0.0,
494             berrsd  = 0.0,
495             Sinitsd = 0.0,
496             Iinitsd = 0.0,
497             Rinitsd = 0.0;
498
499     for (int n = 0; n < NP; n++) {
500
501         R0sd    += ( particles[n].R0 - R0mean ) * ( particles[n].R0 -
                R0mean );
502         rsd     += ( particles[n].r - rmean ) * ( particles[n].r - rmean );
503         resd    += ( particles[n].re - rmean ) * ( particles[n].re - rmean
```

```
            );
        sigmasd += ( particles[n].sigma - sigmamean ) * ( particles[n].
            sigma - sigmamean );
        etasd   += ( particles[n].eta - etamean ) * ( particles[n].eta -
            etamean );
        berrsd  += ( particles[n].berr - berrmean ) * ( particles[n].berr -
             berrmean );
        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;
    resd        /= NP;
    sigmasd     /= NP;
    etasd       /= NP;
    berrsd      /= NP;
    Sinitsd     /= NP;
    Iinitsd     /= NP;
    Rinitsd     /= NP;

    partInfo->R0mean    = R0mean;
    partInfo->R0sd      = R0sd;
    partInfo->rmean     = rmean;
    partInfo->rsd       = rsd;
    partInfo->remean    = remean;
    partInfo->resd      = resd;
    partInfo->sigmamean = sigmamean;
    partInfo->sigmasd   = sigmasd;
    partInfo->etamean   = etamean;
    partInfo->etasd     = etasd;
    partInfo->berrmean  = berrmean;
    partInfo->berrsd    = berrsd;
    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;

}

void getStateMeans(State * state, Particle* particles, int NP) {
```

```
551
552     double Smean = 0, Imean = 0, Rmean = 0;
553
554     for (int n = 0; n < NP; n++) {
555         Smean += particles[n].S;
556         Imean += particles[n].I;
557         Rmean += particles[n].R;
558     }
559
560     state->S = (double) Smean / NP;
561     state->I = (double) Imean / NP;
562     state->R = (double) Rmean / NP;
563
564 }
565
566
567 /*  Return a normally distributed random number with mean 0 and standard
        deviation 1
568     Uses the polar form of the Box-Muller transformation
569     From http://www.design.caltech.edu/erik/Misc/Gaussian.html
570     */
571 double randn() {
572
573     double x1, x2, w, y1;
574
575     do {
576         x1 = 2.0 * randu() - 1.0;
577         x2 = 2.0 * randu() - 1.0;
578         w = x1 * x1 + x2 * x2;
579     } while ( w >= 1.0 );
580
581     w = sqrt( (-2.0 * log( w ) ) / w );
582     y1 = x1 * w;
583
584     return y1;
585
586 }
```