

Spatial Epidemics

Dexter Barrows

March 18, 2016

1 Spatial SIR

Spatial epidemic models provide a way to capture not just the temporal trend in an epidemic, but to also integrate spatial data and infer how the infection is spreading in both space and time. One such model we can use is a dynamic spatiotemporal SIR model.

We wish to construct a model build upon the stochastic SIR compartment model described previously but one that consists of several connected spatial locations, each with its own set of compartments. Consider a set of locations numbered $i = 1, \dots, N$, where N is the number of locations. Further, let N_i be the number of neighbours location i has. The model is then

$$\begin{aligned}\frac{dS_i}{dt} &= - \left(1 - \phi \frac{N_i}{N_i + 1}\right) \beta_i S_i I_i - \left(\frac{\phi}{N_i + 1}\right) S_i \sum_{j=0}^{N_i} \beta_j I_j \\ \frac{dI_i}{dt} &= \left(1 - \phi \frac{N_i}{N_i + 1}\right) \beta_i S_i I_i + \left(\frac{\phi}{N_i + 1}\right) S_i \sum_{j=0}^{N_i} \beta_j I_j - \gamma I_i \\ \frac{dR_i}{dt} &= \gamma I_i,\end{aligned}\tag{1}$$

Neighbours for a particular location are numbered $j = 1, \dots, N_i$. We have a new parameter, $\phi \in [0, 1]$, which is the degree of connectivity. If we let $\phi = 0$ we have total spatial isolation, and the dynamics reduce to the basic SIR model. If we let $\phi = 1$ then each of the neighbouring locations will have weight equivalent to the parent location.

As before we let β embark on a geometric random walk defined as

$$\beta_{i,t+1} = \exp \left(\log(\beta_{i,t}) + \eta(\log(\bar{\beta}) - \log(\beta_{i,t})) + \epsilon_t \right).\tag{2}$$

Note that as β is a state variable, each location has its own stochastic process driving the evolution of its β state.

If we imagine a circular topology in which each of 10 locations is connected to exactly two neighbours (i.e. location 1 is connected to locations N and 2, location 2 is connected to

locations 1 and 3, etc.), and we start each location with completely susceptible populations except for a handful of infected individuals in one of the locations, we obtain a plot of the outbreak progression in Figure [1].

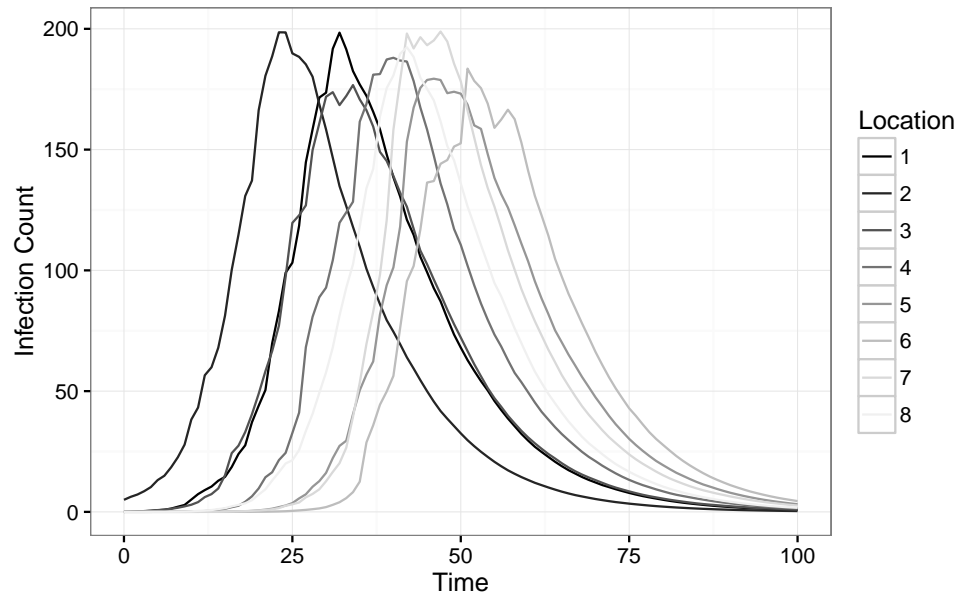


Figure 1: Evolution of a spatial epidemic in a ring topology. The outbreak was started with 5 cases in Location 2. Parameters were $R_0 = 3.0$, $\gamma = 0.1$, $\eta = 0.5$, $\sigma_{err} = 0.5$, and $\phi = 0.5$.

If we add noise to the data from Figure [1], we obtain Figure [2], below.

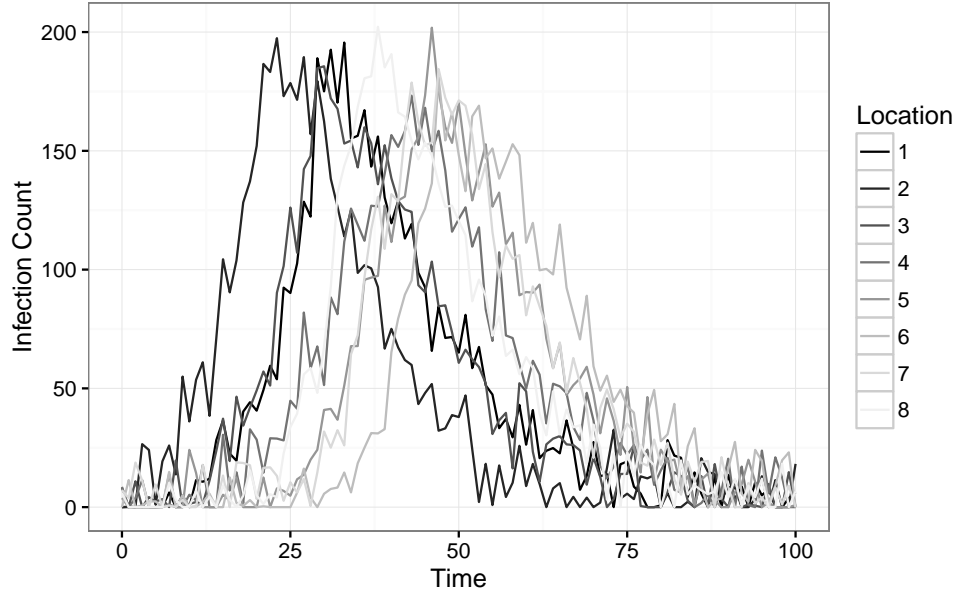


Figure 2: Evolution of a spatial epidemic as in Figure [1], with added observation noise drawn from $\mathcal{N}(0, 10)$.

2 Dewdrop Regression

Dewdrop regression (references) aims to overcome the primary disadvantage suffered by methods such as the S-map or its cousin Simplex Projection: the requirement of long time series from which to build a library. Suggested by Sugihara’s group in 2008, Dewdrop Regression works by stitching together shorter, related, time series, in order to give the S-map or similar methods enough data to operate on. The underlying idea is that as long as the underlying dynamics of the time series display similar behaviour (such as potentially collapsing to the same attractor), they can be treated as part of the same overarching system.

It is not enough to simply concatenate the shorter time series together – several procedures must be carried out and a few caveats observed. First, as the individual time series can be or drastically differing scales and breadths, they all must be rescaled to unit mean and variance. Then the library is constructed as before with an embedding dimension E , but any library vectors that span any of the seams joining the time series are discarded. Further, and predictions stemming from a library vector must stay within the time series from which they originated. In this way we are allowing the “shadow” of of the underlying dynamics of the separate time series to infer the forecasts for segments of other time series. Once the library has been constructed, S-mapping can be carried out as previously specified.

This procedure is especially well-suited to a the spatial model we are using. While the

dynamics are stochastic, they still display very similar means and variances. This means the rescaling process in Dewdrop Regression is not necessary and can be skipped. Further, the overall variation between the epidemic curves in each location is on the smaller side, meaning the S-map will have a high-quality library from which to build forecasts.

3 Spatial Model Forecasting

In order to compare the forecasting efficacy of Dewdrop Regression with S-mapping against IF2 and HMCMC, we generated 20 independent spatial data sets up to time $T = 50$ weeks in each of $L = 10$ locations and forecasted 10 weeks into the future. Forecasts were compared to that of the true model evolution, and the average SSE for each week ahead in the forecast were computed. The number of bootstrapping trajectories used by IF2 and HMCMC was reduced from 200 to 50 to curtail running times.

The results are shown in Figure [3].

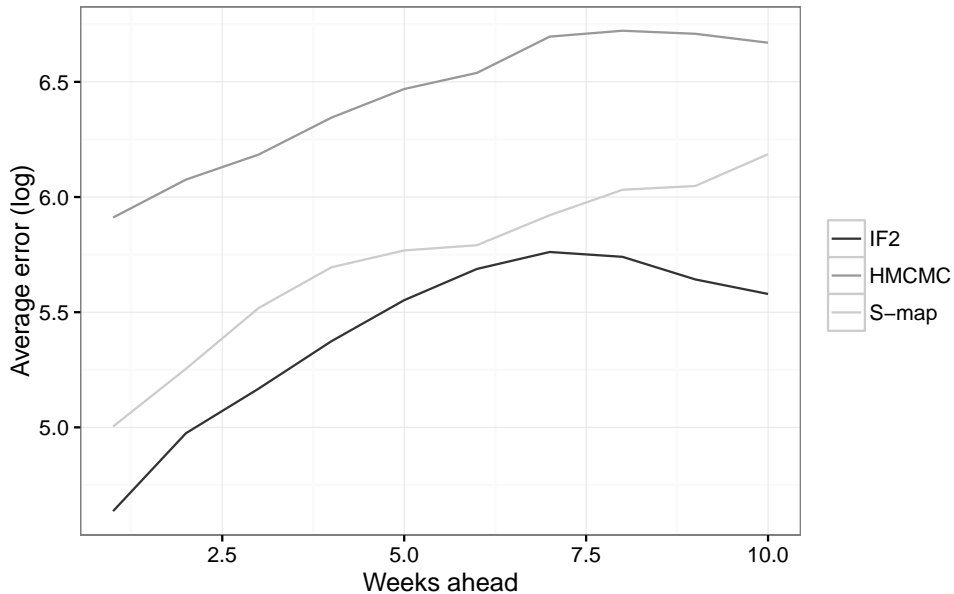


Figure 3: Average SSE (log scale) across each location and all trials as a function of the number of weeks ahead in the forecast.

The results show a clear delineation in forecast fidelity between methods. IF2 maintains an advantage regardless of how long the forecast produced. Interestingly, Dewdrop Regression with S-mapping performs almost as well as IF2, and outperforms HMCMC. HMCMC lags behind both methods by a healthy margin.

If we examine the runtimes for each forecast framework, we obtain the data in Figure

[4].

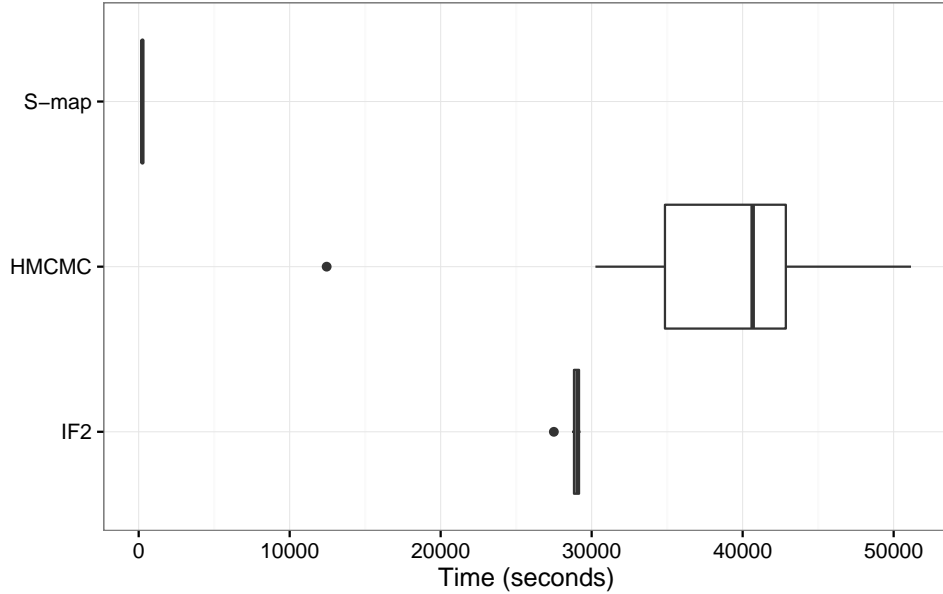


Figure 4: Runtimes for producing spatial SIR forecasts. The box shows the middle 50th quantile, the bold line is the median, and the dots are outliers.

As before, the S-map with Dewdrop Regression runs faster than the other two methods with a huge margin. It is again hard to see exactly how large the margin is from the figure due to the scale, but we can examine the average values: the average running time for S-mapping with Dewdrop Regression was about 249 seconds, whereas the average times for IF2 and HMC MC were about 2.90×10^4 and 3.88×10^4 , respectively. This is a speed-up of just over 116x over IF2 and 156x over HMC MC.

Considering how well S-mapping performed with regards to forecast error, it shows a significant advantage over HMC MC in particular – it outperforms it in both forecast error and running times.

Appendices

A Spatial SIR R Function Code

R code to simulate the outlined Spatial SIR function.

```
1 ## ymat:  Contains the initial conditions where:
2 #        - rows are locations
3 #        - columns are S, I, R
4 ## pars:  Contains the parameters: global values for R0, r, N, eta, berr
5 ## T:     The stop time. Since 0 is included, there should be T+1 time
6           steps in the simulation
7 ## neinum: Number of neighbors for each location, in order
8 ## neibmat: Contains lists of neighbors for each location
9 #        - rows are parent locations (nodes)
10 #        - columns are locations each parent is attached to (edges)
11 StocSSIR ← function(ymat, pars, T, steps, neinum, neibmat) {
12   ## number of locations
13   nloc ← dim(ymat)[1]
14
15   ## storage
16   ## dims are locations, (S,I,R,B), times
17   # output array
18   out ← array(NA, c(nloc, 4, T+1), dimnames = list(NULL, c("S","I","R","B"),
19   " "), NULL))
20   # temp storage
21   BSI ← numeric(nloc)
22   rI ← numeric(nloc)
23
24   ## extract parameters
25   R0 ← pars[['R0']]
26   r ← pars[['r']]
27   N ← pars[['N']]
28   eta ← pars[['eta']]
29   berr ← pars[['berr']]
30   phi ← pars[['phi']]
31
32   B0 ← rep(R0*r/N, nloc)
33
34   ## state vectors
35   S ← ymat[, 'S']
36   I ← ymat[, 'I']
37   R ← ymat[, 'R']
38   B ← B0
39
40   ## assign starting to output matrix
41   out[, , 1] ← cbind(ymat, B0)
42
43   h ← 1 / steps
```

```

43
44   for ( i in 1:(T*steps) ) {
45
46       B ← exp( log(B) + eta*(log(B0) - log(B)) + rnorm(nloc, 0, berr) )
47
48       for (loc in 1:nloc) {
49           n ← neinum[loc]
50           sphi ← 1 - phi*(n/(n+1))
51           ophi ← phi/(n+1)
52           nBIsu ← B[neibmat[loc,1:n]] %*% I[neibmat[loc,1:n]]
53           BSI[loc] ← S[loc]*( sphi*B[loc]*I[loc] + ophi*nBIsu )
54       }
55
56       #if(i == 1)
57       # print(BSI)
58
59       rI ← r*I
60
61       dS ← -BSI
62       dI ← BSI - rI
63       dR ← rI
64
65       S ← S + h*dS
66       I ← I + h*dI
67       R ← R + h*dR
68
69       if (i %% steps == 0) {
70           out[, , i/steps+1] ← cbind(S,I,R,B)
71       }
72
73   }
74
75   #out[, , 2] ← cbind(S,I,R,B)
76
77   return(out)
78
79 }
80
81 ### Suggested parameters
82 #
83 # T          ← 60
84 # i_infec ← 5
85 # steps     ← 7
86 # N         ← 500
87 # sigma     ← 10
88 #
89 # pars ← c(R0 = 3.0,      # new infected people per infected person
90 #          r = 0.1,      # recovery rate
91 #          N = 500,      # population size
92 #          eta = 0.5,    # geometric random walk
93 #          berr = 0.5)   # Beta geometric walk noise

```

B RStan Spatial SIR Code

This code implements a Spatial SIR model in Rstan.

```
1 data {
2
3   int      <lower=1>    T;      // total integration steps
4   int      <lower=1>    nloc;   // number of locations
5   real     y[nloc, T];  // observed number of cases
6   int      <lower=1>    N;      // population size
7   real     h;           // step size
8   int      <lower=0>    neinum[nloc]; // number of neighbors each
    location has
9   int      neibmat[nloc, nloc]; // neighbor list for each
    location
10
11 }
12
13 parameters {
14
15   real <lower=0, upper=10>    R0;      // R0
16   real <lower=0, upper=10>    r;       // recovery rate
17   real <lower=0, upper=20>    sigma;   // observation error
18   real <lower=0, upper=30>    Iinit[nloc]; // initial infected for
    each location
19   real <lower=0, upper=1>      eta;     // geometric walk attraction
    strength
20   real <lower=0, upper=1>      berr;    // beta walk noise
21   real <lower=-1.5, upper=1.5> Bnoise[nloc, T]; // Beta vector
22   real <lower=0, upper=1>      phi;    // interconnectivity strength
23
24 }
25
26 model {
27
28   real S[nloc, T];
29   real I[nloc, T];
30   real R[nloc, T];
31   real B[nloc, T];
32   real B0;
33
34   real BSI[nloc, T];
35   real rI[nloc, T];
36   int n;
37   real sphi;
38   real ophi;
39   real nBIsum;
40
41   B0 ← R0 * r / N;
42
43   for (loc in 1:nloc) {
44     S[loc, 1] ← N - Iinit[loc];
45     I[loc, 1] ← Iinit[loc];
```



```

46     R[loc, 1] ← 0.0;
47     B[loc, 1] ← B0;
48 }
49
50 for (t in 2:T) {
51     for (loc in 1:nloc) {
52
53         Bnoise[loc, t] ~ normal(0,berr);
54         B[loc, t] ← exp( log(B[loc, t-1]) + eta * ( log(B0) - log(B[loc
55             , t-1]) ) + Bnoise[loc, t] );
56
57         n ← neinum[loc];
58         sphl ← 1.0 - phi*( n/(n+1.0) );
59         ophi ← phi/(n+1.0);
60
61         nBIsun ← 0.0;
62         for (j in 1:n)
63             nBIsun ← nBIsun + B[neibmat[loc, j], t-1] * I[neibmat[loc,
64                 j], t-1];
65
66         BSI[loc, t] ← S[loc, t-1]*( sphl*B[loc, t-1]*I[loc, t-1] + ophi
67             *nBIsun );
68         rI[loc, t] ← r*I[loc, t-1];
69
70         S[loc, t] ← S[loc, t-1] + h*( - BSI[loc, t] );
71         I[loc, t] ← I[loc, t-1] + h*( BSI[loc, t] - rI[loc, t] );
72         R[loc, t] ← R[loc, t-1] + h*( rI[loc, t] );
73
74         if (y[loc, t] > 0) {
75             y[loc, t] ~ normal( I[loc, t], sigma );
76         }
77     }
78 }
79
80 R0 ~ lognormal(1,1);
81 r ~ lognormal(1,1);
82 sigma ~ lognormal(1,1);
83 for (loc in 1:nloc) {
84     Iinit[loc] ~ normal(y[loc, 1], sigma);
85 }

```

C IF2 Spatial SIR Code

This code implements a Spatial SIR model using IF2 in C++.

```

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 etatrue    0.5          // real drift attraction strength
24 #define berrtrue   0.5          // real beta drift noise
25 #define phitrue    0.5          // real connectivity strength
26 #define merr       10.0         // expected measurement error
27 #define I0         5.0          // Initial infected individuals
28
29 #define PSC        0.5          // perturbation scale factor for more
    sensitive parameters
30
31 #include <Rcpp.h>
32 using namespace Rcpp;
33
34 struct Particle {
35     double R0;
36     double r;
37     double sigma;
38     double eta;
39     double berr;
40     double phi;
41     double * S;
42     double * I;
43     double * R;
44     double * B;
45     double * Iinit;
46 };
47
48
49 int timeval_subtract (double *result, struct timeval *x, struct timeval *y)
    ;
50 int check_double(double x, double y);
51 void initializeParticles(Particle ** particles, int NP, int nloc, int N);
52 void exp_euler_SSIR(double h, double t0, double tn, int N, Particle *
    particle,
53                     NumericVector neinum, NumericMatrix neibmat, int nloc)

```

```

54 void copyParticle(Particle * dst, Particle * src, int nloc);
55 void perturbParticles(Particle * particles, int N, int NP, int nloc, int
    passnum, double coolrate);
56 double randu();
57 double randn();
58
59 // [[Rcpp::export]]
60 Rcpp::List if2_spa(NumericMatrix data, int T, int N, int NP, int nPasses,
    double coolrate, NumericVector neinum, NumericMatrix neibmat, int nloc)
    {
61
62     NumericMatrix paramdata(NP, 6);      // for R0, r, sigma, eta, berr, phi
63     NumericMatrix initInfec(nloc, NP);   // for Iinit
64     NumericMatrix infecmeans(nloc, T);   // mean infection counts for each
        location
65     NumericMatrix finalstate(nloc, 4);   // SIRB means for each location
66
67     srand(time(NULL));                  // Seed PRNG with system time
68
69     double w[NP];                      // particle weights
70
71     // initialize particles
72     printf("Initializing particle states\n");
73     Particle * particles = NULL;        // particle estimates for current
        step
74     Particle * particles_old = NULL;    // intermediate particle states for
        resampling
75     initializeParticles(&particles, NP, nloc, N);
76     initializeParticles(&particles_old, NP, nloc, N);
77
78     /*
79     // copy particle test
80     copyParticle(&particles[0], &particles_old[0], nloc);
81
82     // perturb particle test
83     perturbParticles(particles, N, NP, nloc, 1, coolrate);
84
85     // evolution test
86     // reset particle system evolution states
87     for (int n = 0; n < NP; n++) {
88         for (int loc = 0; loc < nloc; loc++) {
89             particles[n].S[loc] = N - particles[n].Iinit[loc];
90             particles[n].I[loc] = particles[n].Iinit[loc];
91             particles[n].R[loc] = 0.0;
92             particles[n].B[loc] = (double) particles[n].R0 * particles[n].r
                / N;
93         }
94     }
95     printf("Before S:%f | I:%f | R:%f\n", particles[0].S[0], particles[0].I
        [0], particles[0].R[0]);
96     exp_euler_SSIR(1.0/7.0, 0.0, 1.0, N, &particles[0], neinum, neibmat,
        nloc);
97     printf("After S:%f | I:%f | R:%f\n", particles[0].S[0], particles[0].I

```

```

98     [0], particles[0].R[0]);
99     */
100     // START PASSES THROUGH DATA
101
102     printf("Starting filter\n");
103     printf("-----\n");
104     printf("Pass\n");
105
106
107     for (int pass = 0; pass < nPasses; pass++) {
108
109         printf("...%d / %d\n", pass, nPasses);
110
111         // reset particle system evolution states
112         for (int n = 0; n < NP; n++) {
113             for (int loc = 0; loc < nloc; loc++) {
114                 particles[n].S[loc] = N - particles[n].Iinit[loc];
115                 particles[n].I[loc] = particles[n].Iinit[loc];
116                 particles[n].R[loc] = 0.0;
117                 particles[n].B[loc] = (double) particles[n].R0 * particles[
n].r / N;
118             }
119         }
120
121         if (pass == (nPasses-1)) {
122             double means[nloc];
123             for (int loc = 0; loc < nloc; loc++) {
124                 means[loc] = 0.0;
125                 for (int n = 0; n < NP; n++) {
126                     means[loc] += particles[n].I[loc] / NP;
127                 }
128                 infecmeans(loc, 0) = means[loc];
129             }
130         }
131
132         for (int t = 1; t < T; t++) {
133
134             // generate individual predictions and weight
135             for (int n = 0; n < NP; n++) {
136
137                 exp_euler_SSIR(1.0/7.0, 0.0, 1.0, N, &particles[n], neinum,
neibmat, nloc);
138
139                 double merr_par = particles[n].sigma;
140
141                 w[n] = 1.0;
142                 for (int loc = 0; loc < nloc; loc++) {
143                     double y_diff = data(loc, t) - particles[n].I[loc];
144                     w[n] *= 1.0/(merr_par*sqrt(2.0*M_PI)) * exp( - y_diff*
y_diff / (2.0*merr_par*merr_par) );
145                 }
146
147             }

```

```

148
149 // cumulative sum
150 for (int n = 1; n < NP; n++) {
151     w[n] += w[n-1];
152 }
153
154 // save particle states to resample from
155 for (int n = 0; n < NP; n++){
156     copyParticle(&particles_old[n], &particles[n], nloc);
157 }
158
159 // resampling
160 for (int n = 0; n < NP; n++) {
161
162     double w_r = randu() * w[NP-1];
163     int i = 0;
164     while (w_r > w[i]) {
165         i++;
166     }
167
168     // i is now the index to copy state from
169     copyParticle(&particles[n], &particles_old[i], nloc);
170
171 }
172
173 // between-iteration perturbations, not after last time step
174 if (t < (T-1))
175     perturbParticles(particles, N, NP, nloc, pass, coolrate);
176
177 if (pass == (nPasses-1)) {
178     double means[nloc];
179     for (int loc = 0; loc < nloc; loc++) {
180         means[loc] = 0.0;
181         for (int n = 0; n < NP; n++) {
182             means[loc] += particles[n].I[loc] / NP;
183         }
184         infecmeans(loc, t) = means[loc];
185     }
186 }
187
188 }
189
190 // between-pass perturbations, not after last pass
191 if (pass < (nPasses + 1))
192     perturbParticles(particles, N, NP, nloc, pass, coolrate);
193
194 }
195
196 // pack parameter data (minus initial conditions)
197 for (int n = 0; n < NP; n++) {
198     paramdata(n, 0) = particles[n].R0;
199     paramdata(n, 1) = particles[n].r;
200     paramdata(n, 2) = particles[n].sigma;
201     paramdata(n, 3) = particles[n].eta;

```

```

202     paramdata(n, 4) = particles[n].berr;
203     paramdata(n, 5) = particles[n].phi;
204 }
205
206 // Pack initial condition data
207 for (int n = 0; n < NP; n++) {
208     for (int loc = 0; loc < nloc; loc++) {
209         initInfec(loc, n) = particles[n].Iinit[loc];
210     }
211 }
212
213 // Pack final state means data
214 double Smeans[nloc], Imeans[nloc], Rmeans[nloc], Bmeans[nloc];
215 for (int loc = 0; loc < nloc; loc++) {
216     Smeans[loc] = 0.0;
217     Imeans[loc] = 0.0;
218     Rmeans[loc] = 0.0;
219     Bmeans[loc] = 0.0;
220     for (int n = 0; n < NP; n++) {
221         Smeans[loc] += particles[n].S[loc] / NP;
222         Imeans[loc] += particles[n].I[loc] / NP;
223         Rmeans[loc] += particles[n].R[loc] / NP;
224         Bmeans[loc] += particles[n].B[loc] / NP;
225     }
226     finalstate(loc, 0) = Smeans[loc];
227     finalstate(loc, 1) = Imeans[loc];
228     finalstate(loc, 2) = Rmeans[loc];
229     finalstate(loc, 3) = Bmeans[loc];
230 }
231
232
233 return Rcpp::List::create( Rcpp::Named("paramdata") = paramdata,
234                           Rcpp::Named("initInfec") = initInfec,
235                           Rcpp::Named("infecmeans") = infecmeans,
236                           Rcpp::Named("finalstate") = finalstate);
237
238
239
240 }
241
242
243 /* Use the Explicit Euler integration scheme to integrate SIR model
244    forward in time
245    double h      - time step size
246    double t0     - start time
247    double tn     - stop time
248    double * y    - current system state; a three-component vector
249                   representing [S I R], susceptible-infected-recovered
250
251    */
252 void exp_euler_SSIR(double h, double t0, double tn, int N, Particle *
253     particle,
254     NumericVector neinum, NumericMatrix neibmat, int nloc)
255 {

```

```

252
253 int num_steps = floor( (tn-t0) / h );
254
255 double * S = particle->S;
256 double * I = particle->I;
257 double * R = particle->R;
258 double * B = particle->B;
259
260 // create last state vectors
261 double S_last[nloc];
262 double I_last[nloc];
263 double R_last[nloc];
264 double B_last[nloc];
265
266 double R0 = particle->R0;
267 double r = particle->r;
268 double B0 = R0 * r / N;
269 double eta = particle->eta;
270 double berr = particle->berr;
271 double phi = particle->phi;
272
273 //printf("sphi \t\t| ophi \t\t| BSI \t\t| rI \t\t| dS \t\t| dI \t\t| dR
\t\t| S \t\t| I \t\t| R |\n");
274
275 for(int t = 0; t < num_steps; t++) {
276
277     for (int loc = 0; loc < nloc; loc++) {
278         S_last[loc] = S[loc];
279         I_last[loc] = I[loc];
280         R_last[loc] = R[loc];
281         B_last[loc] = B[loc];
282     }
283
284     for (int loc = 0; loc < nloc; loc++) {
285
286         B[loc] = exp( log(B_last[loc]) + eta*(log(B0) - log(B_last[loc]
))) + berr*randn() );
287
288         int n = neinum[loc];
289         double sphi = 1.0 - phi*( (double) n/(n+1.0) );
290         double ophi = phi/(n+1.0);
291
292         double nBIsun = 0.0;
293         for (int j = 0; j < n; j++)
294             nBIsun += B_last[(int) neibmat(loc, j) - 1] * I_last[(int)
neibmat(loc, j) - 1];
295
296         double BSI = S_last[loc]*( sphi*B_last[loc]*I_last[loc] + ophi*
nBIsun );
297         double rI = r*I_last[loc];
298
299         // get derivatives
300         double dS = - BSI;
301         double dI = BSI - rI;

```

```

302         double dR = rI;
303
304         // step forward by h
305         S[loc] += h*dS;
306         I[loc] += h*dI;
307         R[loc] += h*dR;
308
309         //if (loc == 1)
310         // printf("%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t\\
311             n", sphi, ophi, BSI, rI, dS, dI, dR, S[1], I[1], R[1]);
312     }
313
314 }
315
316 /*particle->S = S;
317 particle->I = I;
318 particle->R = R;
319 particle->B = B;*/
320
321 }
322
323 /*  Initializes particles
324 */
325 void initializeParticles(Particle ** particles, int NP, int nloc, int N) {
326
327     // allocate space for doubles
328     *particles = (Particle*) malloc (NP*sizeof(Particle));
329
330     // allocate space for arrays inside particles
331     for (int n = 0; n < NP; n++) {
332         (*particles)[n].S = (double*) malloc(nloc*sizeof(double));
333         (*particles)[n].I = (double*) malloc(nloc*sizeof(double));
334         (*particles)[n].R = (double*) malloc(nloc*sizeof(double));
335         (*particles)[n].B = (double*) malloc(nloc*sizeof(double));
336         (*particles)[n].Iinit = (double*) malloc(nloc*sizeof(double));
337     }
338
339     // initialize all all parameters
340     for (int n = 0; n < NP; n++) {
341
342         double R0can, rcan, sigmacan, Iinitcan, etacan, berrcan, phican;
343
344         do {
345             R0can = R0true + R0true*randn();
346         } while (R0can < 0);
347         (*particles)[n].R0 = R0can;
348
349         do {
350             rcan = rtrue + rtrue*randn();
351         } while (rcan < 0);
352         (*particles)[n].r = rcan;
353
354         for (int loc = 0; loc < nloc; loc++)

```



```

355         (*particles)[n].B[loc] = (double) R0can * rcan / N;
356
357     do {
358         sigmacan = merr + merr*randn();
359     } while (sigmacan < 0);
360     (*particles)[n].sigma = sigmacan;
361
362     do {
363         etacan = etatrue + PSC*etatrue*randn();
364     } while (etacan < 0 || etacan > 1);
365     (*particles)[n].eta = etacan;
366
367     do {
368         berrcan = berrtrue + PSC*berrtrue*randn();
369     } while (berrcan < 0);
370     (*particles)[n].berr = berrcan;
371
372     do {
373         phican = phitrue + PSC*phitrue*randn();
374     } while (phican <= 0 || phican >= 1);
375     (*particles)[n].phi = phican;
376
377     for (int loc = 0; loc < nloc; loc++) {
378         do {
379             Iinitcan = I0 + I0*randn();
380         } while (Iinitcan < 0 || N < Iinitcan);
381         (*particles)[n].Iinit[loc] = Iinitcan;
382     }
383
384 }
385
386 }
387
388 /* Particle pertubation function to be run between iterations and passes
389
390 */
391 void perturbParticles(Particle * particles, int N, int NP, int nloc, int
    passnum, double coolrate) {
392
393     //double coolcoef = exp( - (double) passnum / coolrate );
394     double coolcoef = pow(coolrate, passnum);
395
396     double spreadR0      = coolcoef * R0true / 10.0;
397     double spreadr       = coolcoef * rtrue / 10.0;
398     double spreadsigma   = coolcoef * merr / 10.0;
399     double spreadIinit   = coolcoef * I0 / 10.0;
400     double spreadeta     = coolcoef * etatrue / 10.0;
401     double spreadberr    = coolcoef * berrtrue / 10.0;
402     double spreadphi     = coolcoef * phitrue / 10.0;
403
404     double R0can, rcan, sigmacan, Iinitcan, etacan, berrcan, phican;
405
406     for (int n = 0; n < NP; n++) {
407

```

```

408     do {
409         R0can = particles[n].R0 + spreadR0*randn();
410     } while (R0can < 0);
411     particles[n].R0 = R0can;
412
413     do {
414         rcan = particles[n].r + spreadr*randn();
415     } while (rcan < 0);
416     particles[n].r = rcan;
417
418     do {
419         sigmacan = particles[n].sigma + spreadsigma*randn();
420     } while (sigmacan < 0);
421     particles[n].sigma = sigmacan;
422
423     do {
424         etacan = particles[n].eta + PSC*spreadeta*randn();
425     } while (etacan < 0 || etacan > 1);
426     particles[n].eta = etacan;
427
428     do {
429         berrcan = particles[n].berr + PSC*spreadberr*randn();
430     } while (berrcan < 0);
431     particles[n].berr = berrcan;
432
433     do {
434         phican = particles[n].phi + PSC*spreadphi*randn();
435     } while (phican <= 0 || phican >= 1);
436     particles[n].phi = phican;
437
438     for (int loc = 0; loc < nloc; loc++) {
439         do {
440             Iinitcan = particles[n].Iinit[loc] + spreadIinit*randn();
441         } while (Iinitcan < 0 || Iinitcan > 500);
442         particles[n].Iinit[loc] = Iinitcan;
443     }
444 }
445
446 }
447
448 /* Convenience function for particle resampling process
449 */
450 void copyParticle(Particle * dst, Particle * src, int nloc) {
451
452     dst->R0      = src->R0;
453     dst->r        = src->r;
454     dst->sigma    = src->sigma;
455     dst->eta      = src->eta;
456     dst->berr     = src->berr;
457     dst->phi      = src->phi;
458
459     for (int n = 0; n < nloc; n++) {
460         dst->S[n]      = src->S[n];
461         dst->I[n]      = src->I[n];

```

```

462         dst->R[n]      = src->R[n];
463         dst->B[n]      = src->B[n];
464         dst->Iinit[n]   = src->Iinit[n];
465     }
466
467 }
468
469
470
471 double randu() {
472
473     return (double) rand() / (double) RAND_MAX;
474
475 }
476
477 /*
478 void getStateMeans(State * state, Particle* particles, int NP) {
479
480     double Smean = 0, Imean = 0, Rmean = 0;
481
482     for (int n = 0; n < NP; n++) {
483         Smean += particles[n].S;
484         Imean += particles[n].I;
485         Rmean += particles[n].R;
486     }
487
488     state->S = (double) Smean / NP;
489     state->I = (double) Imean / NP;
490     state->R = (double) Rmean / NP;
491
492 }
493 */
494
495 /* Return a normally distributed random number with mean 0 and standard
496    deviation 1
497    Uses the polar form of the Box-Muller transformation
498    From http://www.design.caltech.edu/erik/Misc/Gaussian.html
499    */
500 double randn() {
501     double x1, x2, w, y1;
502
503     do {
504         x1 = 2.0 * randu() - 1.0;
505         x2 = 2.0 * randu() - 1.0;
506         w = x1 * x1 + x2 * x2;
507     } while ( w >= 1.0 );
508
509     w = sqrt( (-2.0 * log( w ) ) / w );
510     y1 = x1 * w;
511
512     return y1;
513
514 }

```
