Spatial Epidemics

Dexter Barrows
March 6, 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 $i = 1, ..., N$, where N is the number of locations. Further, let N_i be the number of neighbours location i has. The model is then

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

Neighbours for a particular location are numbered $j = 1, ..., N_i$. 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].

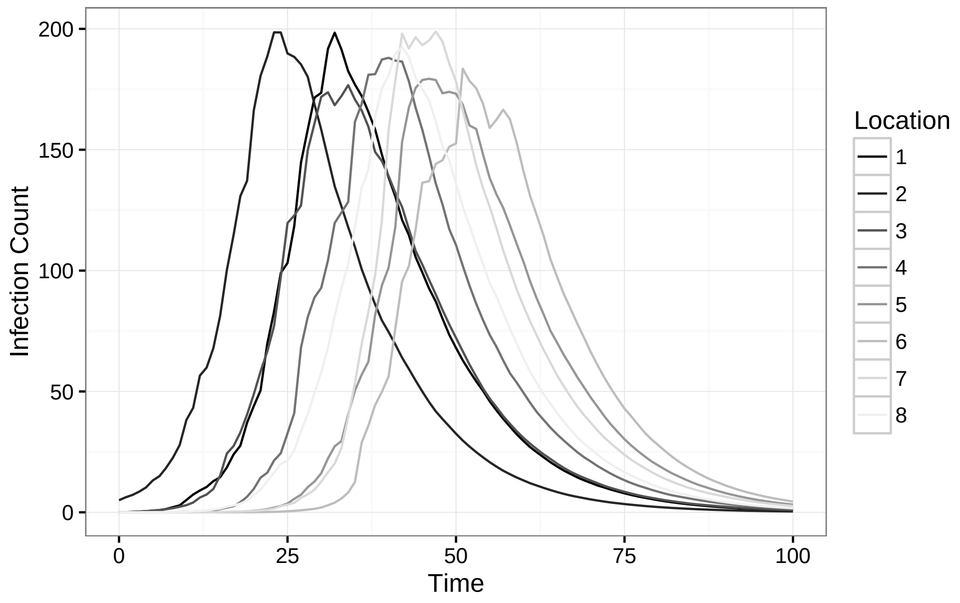

Figure 1: Evolution of a spatial epidemic in a ring topology. The outbreak was started with 5 cases in Location 2. Parameters were $R_0 = 3.0$, $\gamma = 0.1$, $\eta = 0.5$, $\sigma_{err} = 0.5$, and $\phi = 0.5$.

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

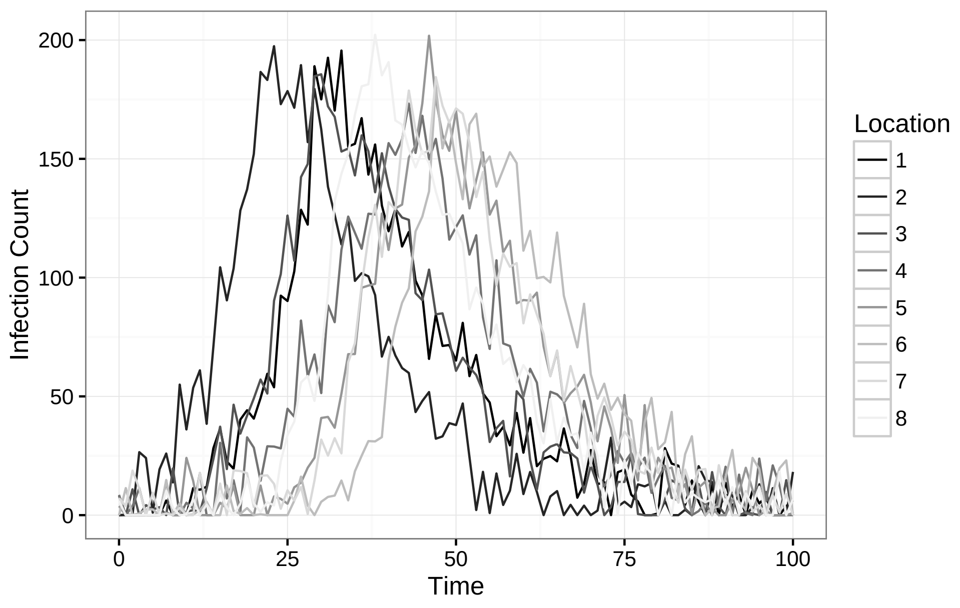

Figure 2: Evolution of a spatial epidemic as in Figure [1], with added observation noise drawn from $\mathcal{N}(0, 10)$.

2 Dewdrop Regression

Appendices

A Spatial SIRS R Function Code

R code to simulate the outlined Spatial SIR function.

```
## ymat:  Contains the initial conditions where:
#       - rows are locations
#       - columns are S, I, R
## pars:  Contains the parameters: global values for R0, r, N, eta, berr
## T:     The stop time. Since 0 in included, there should be T+1 time
   steps in the simulation
## neinum:  Number of neighbors for each location, in order
## neibmat: Contains lists of neighbors for each location
#     - rows are parent locations (nodes)
#       - columns are locations each parent is attached to (edges)
StocSSIR ← function(ymat, pars, T, steps, neinum, neibmat) {

  ## number of locations
    nloc ← dim(ymat)[1]

    ## storage
    ## dims are locations, (S,I,R,B), times
    # output array
    out ← array(NA, c(nloc, 4, T+1), dimnames = list(NULL, c("S","I","R","B
       "), NULL))
    # temp storage
    BSI ← numeric(nloc)
    rI ← numeric(nloc)

    ## extract parameters
    R0 ← pars[['R0']]
    r ← pars[['r']]
    N ← pars[['N']]
    eta ← pars[['eta']]
    berr ← pars[['berr']]
    phi ← pars[['phi']]

    B0 ← rep(R0*r/N, nloc)

    ## state vectors
    S ← ymat[,'S']
    I ← ymat[,'I']
    R ← ymat[,'R']
    B ← B0

    ## assign starting to output matrix
    out[,,1] ← cbind(ymat, B0)

    h ← 1 / steps

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

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

        for (loc in 1:nloc) {
          n ← neinum[loc]
          sphi ← 1 - phi*(n/(n+1))
          ophi ← phi/(n+1)
          nBIsum ← B[neibmat[loc,1:n]] %*% I[neibmat[loc,1:n]]
          BSI[loc] ← S[loc]*( sphi*B[loc]*I[loc] + ophi*nBIsum )
        }

        #if(i == 1)
        # print(BSI)

        rI ← r*I

        dS ← -BSI
        dI ← BSI - rI
        dR ← rI

        S ← S + h*dS
        I ← I + h*dI
        R ← R + h*dR

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

    }

    #out[,,2] ← cbind(S,I,R,B)

  return(out)

}

### Suggested parameters
#
# T       ← 60
# i_infec ← 5
# steps   ← 7
# N       ← 500
# sigma   ← 10
#
# 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
```

B RStan Spatial SIR Code

This code implements a Spatial SIR model in Rstan.

```
data {

    int     <lower=1>   T;      // total integration steps
    int     <lower=1>   nloc;   // number of locations
    real                y[nloc, T];   // observed number of cases
    int     <lower=1>   N;      // population size
    real                h;      // step size
    int     <lower=0>   neinum[nloc];        // number of neighbors each
       location has
    int                 neibmat[nloc, nloc]; // neighbor list for each
       location

}

parameters {

    real <lower=0, upper=10>         R0;     // R0
    real <lower=0, upper=10>         r;      // recovery rate
    real <lower=0, upper=20>         sigma;  // observation error
    real <lower=0, upper=30>         Iinit[nloc];    // initial infected for
        each location
    real <lower=0, upper=1>          eta;    // geometric walk attraction
       strength
    real <lower=0, upper=1>          berr;   // beta walk noise
    real <lower=-1.5, upper=1.5>     Bnoise[nloc,T];   // Beta vector
    real <lower=0, upper=1>          phi;    // interconnectivity strength

}

model {

    real S[nloc, T];
    real I[nloc, T];
    real R[nloc, T];
    real B[nloc, T];
    real B0;

    real BSI[nloc, T];
    real rI[nloc, T];
    int  n;
    real sphi;
    real ophi;
    real nBIsum;

    B0 ← R0 * r / N;

    for (loc in 1:nloc) {
        S[loc, 1] ← N - Iinit[loc];
        I[loc, 1] ← Iinit[loc];
        R[loc, 1] ← 0.0;
        B[loc, 1] ← B0;
    }

    for (t in 2:T) {
        for (loc in 1:nloc) {

            Bnoise[loc, t] ~ normal(0,berr);
            B[loc, t] ← exp( log(B[loc, t-1]) + eta * ( log(B0) - log(B[loc
                , t-1]) ) + Bnoise[loc, t] );

            n ← neinum[loc];
            sphi ← 1.0 - phi*( n/(n+1.0) );
            ophi ← phi/(n+1.0);

            nBIsum ← 0.0;
            for (j in 1:n)
                nBIsum ← nBIsum + B[neibmat[loc, j], t-1] * I[neibmat[loc,
                    j], t-1];

            BSI[loc, t] ← S[loc, t-1]*( sphi*B[loc, t-1]*I[loc, t-1] + ophi
                *nBIsum );
            rI[loc, t]  ← r*I[loc, t-1];

            S[loc, t] ← S[loc, t-1] + h*( - BSI[loc, t] );
            I[loc, t] ← I[loc, t-1] + h*( BSI[loc, t] - rI[loc, t] );
            R[loc, t] ← R[loc, t-1] + h*( rI[loc, t] );

            if (y[loc, t] > 0) {
                y[loc, t] ~ normal( I[loc, t], sigma );
            }

        }
    }

    R0      ~ lognormal(1,1);
    r       ~ lognormal(1,1);
    sigma   ~ lognormal(1,1);
    for (loc in 1:nloc) {
        Iinit[loc] ~ normal(y[loc, 1], sigma);
    }

}
```

C IF2 Spatial SIR Code

This code implements a Spatial SIR 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 Nreal        500.0        // population size
#define etatrue      0.5          // real drift attraction strength
#define berrtrue     0.5          // real beta drift noise
#define phitrue      0.5          // real connectivity strength
#define merr         10.0         // expected measurement error
#define I0           5.0          // Initial infected individuals

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

#include <Rcpp.h>
using namespace Rcpp;

struct Particle {
    double R0;
    double r;
    double sigma;
    double eta;
    double berr;
    double phi;
    double * S;
    double * I;
    double * R;
    double * B;
    double * Iinit;
};


int timeval_subtract (double *result, struct timeval *x, struct timeval *y)
    ;
int check_double(double x,double y);
void initializeParticles(Particle ** particles, int NP, int nloc, int N);
void exp_euler_SSIR(double h, double t0, double tn, int N, Particle *
    particle,
                    NumericVector neinum, NumericMatrix neibmat, int nloc)
```

```
                         ;
void copyParticle(Particle * dst, Particle * src, int nloc);
void perturbParticles(Particle * particles, int N, int NP, int nloc, int
    passnum, double coolrate);
double randu();
double randn();

// [[Rcpp::export]]
Rcpp::List if2_spa(NumericMatrix data, int T, int N, int NP, int nPasses,
    double coolrate, NumericVector neinum, NumericMatrix neibmat, int nloc)
     {

     NumericMatrix paramdata(NP, 6);     // for R0, r, sigma, eta, berr, phi
     NumericMatrix initInfec(nloc, NP);  // for Iinit
     NumericMatrix infecmeans(nloc, T);  // mean infection counts for each
         location
     NumericMatrix finalstate(nloc, 4);  // SIRB means for each location

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

     double w[NP];           // particle weights

     // initialize particles
     printf("Initializing particle states\n");
     Particle * particles = NULL;        // particle estimates for current
         step
     Particle * particles_old = NULL;    // intermediate particle states for
          resampling
     initializeParticles(&particles, NP, nloc, N);
     initializeParticles(&particles_old, NP, nloc, N);

     // copy particle test
     copyParticle(&particles[0], &particles_old[0], nloc);

     // perturb particle test
     perturbParticles(particles, N, NP, nloc, 1, coolrate);

     // evolution test
     // reset particle system evolution states
     for (int n = 0; n < NP; n++) {
         for (int loc = 0; loc < nloc; loc++) {
             particles[n].S[loc] = N - particles[n].Iinit[loc];
             particles[n].I[loc] = particles[n].Iinit[loc];
             particles[n].R[loc] = 0.0;
             particles[n].B[loc] = (double) particles[n].R0 * particles[n].r
                  / N;
         }
     }
     printf("Before S:%f | I:%f | R:%f\n", particles[0].S[0], particles[0].I
         [0], particles[0].R[0]);
     exp_euler_SSIR(1.0/7.0, 0.0, 1.0, N, &particles[0], neinum, neibmat,
         nloc);
     printf("After S:%f | I:%f | R:%f\n", particles[0].S[0], particles[0].I
         [0], particles[0].R[0]);

     // START PASSES THROUGH DATA

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


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

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

         // reset particle system evolution states
         for (int n = 0; n < NP; n++) {
             for (int loc = 0; loc < nloc; loc++) {
                  particles[n].S[loc] = N - particles[n].Iinit[loc];
                  particles[n].I[loc] = particles[n].Iinit[loc];
                  particles[n].R[loc] = 0.0;
                  particles[n].B[loc] = (double) particles[n].R0 * particles[
                      n].r / N;
             }
         }

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

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

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

                  exp_euler_SSIR(1.0/7.0, 0.0, 1.0, N, &particles[n], neinum,
                       neibmat, nloc);

                  double merr_par = particles[n].sigma;

                  w[n] = 1.0;
                  for (int loc = 0; loc < nloc; loc++) {
                      double y_diff   = data(loc, t) - particles[n].I[loc];
                      w[n] *= 1.0/(merr_par*sqrt(2.0*M_PI)) * exp( - y_diff*
                          y_diff / (2.0*merr_par*merr_par) );
                  }

             }

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

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

             // 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], nloc);

             }

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

         }

         // between-pass perturbations, not after last pass
         if (pass < (nPasses + 1))
             perturbParticles(particles, N, NP, nloc, pass, coolrate);

     }

     // pack parameter data (minus initial conditions)
     for (int n = 0; n < NP; n++) {
         paramdata(n, 0) = particles[n].R0;
         paramdata(n, 1) = particles[n].r;
         paramdata(n, 2) = particles[n].sigma;
         paramdata(n, 3) = particles[n].eta;
         paramdata(n, 4) = particles[n].berr;
         paramdata(n, 5) = particles[n].phi;
     }

     // Pack initial condition data
     for (int n = 0; n < NP; n++) {
         for (int loc = 0; loc < nloc; loc++) {
             initInfec(loc, n) = particles[n].Iinit[loc];
         }
     }

     return Rcpp::List::create(  Rcpp::Named("paramdata") = paramdata,
                                  Rcpp::Named("initInfec") = initInfec,
                                  Rcpp::Named("infecmeans") = infecmeans,
                                  Rcpp::Named("finalstate") = finalstate);



  }


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

     */
 void exp_euler_SSIR(double h, double t0, double tn, int N, Particle *
    particle,
                     NumericVector neinum, NumericMatrix neibmat, int nloc)
                         {

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

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

     double R0   = particle->R0;
     double r    = particle->r;
     double B0   = R0 * r / N;
     double eta  = particle->eta;
     double berr = particle->berr;
     double phi  = particle->phi;

     double * B = particle->B;

     //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");

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

         for (int loc = 0; loc < nloc; loc++) {

             B[loc] = exp( log(B[loc]) + eta*(log(B0) - log(B[loc])) + berr*
                 randn() );

             int n = neinum[loc];
             double sphi = 1.0 - phi*((double) n/(n+1.0) );
             double ophi = phi/(n+1.0);

             double nBIsum = 0.0;
             for (int j = 0; j < n; j++)
                 nBIsum += B[(int) neibmat(loc, j) - 1] * I[(int) neibmat(
                     loc, j) - 1];

             double BSI = S[loc]*( sphi*B[loc]*I[loc] + ophi*nBIsum );
             double rI  = r*I[loc];

             // get derivatives
             double dS = - BSI;
             double dI = BSI - rI;
             double dR = rI;

             // step forward by h
             S[loc] += h*dS;
             I[loc] += h*dI;
             R[loc] += h*dR;

             //if (loc == 1)
             //  printf("%f\t|%f\t|%f\t|%f\t|%f\t|%f\t|%f\t|%f\t|%f\t|%f\t|\
                 n", sphi, ophi, BSI, rI, dS, dI, dR, S[1], I[1], R[1]);

         }

     }

     /*particle->S = S;
     particle->I = I;
     particle->R = R;
     particle->B = B;*/

 }

 /*  Initializes particles
     */
 void initializeParticles(Particle ** particles, int NP, int nloc, int N) {

     // allocate space for doubles
     *particles = (Particle*) malloc (NP*sizeof(Particle));

     // allocate space for arays inside particles
     for (int n = 0; n < NP; n++) {
         (*particles)[n].S = (double*) malloc(nloc*sizeof(double));
         (*particles)[n].I = (double*) malloc(nloc*sizeof(double));
         (*particles)[n].R = (double*) malloc(nloc*sizeof(double));
         (*particles)[n].B = (double*) malloc(nloc*sizeof(double));
         (*particles)[n].Iinit = (double*) malloc(nloc*sizeof(double));
     }

     // initialize all all parameters
     for (int n = 0; n < NP; n++) {

         double R0can, rcan, sigmacan, Iinitcan, etacan, berrcan, phican;

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

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

         for (int loc = 0; loc < nloc; loc++)
             (*particles)[n].B[loc] = (double) R0can * rcan / N;

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

         do {
             etacan = etatrue + PSC*etatrue*randn();
         } while (etacan < 0 || etacan > 1);
         (*particles)[n].eta = etacan;

         do {
             berrcan = berrtrue + PSC*berrtrue*randn();
         } while (berrcan < 0);
         (*particles)[n].berr = berrcan;

         do {
             phican = phitrue + PSC*phitrue*randn();
         } while (phican < 0 || phican > 1);
         (*particles)[n].phi = phican;

         for (int loc = 0; loc < nloc; loc++) {
             do {
                 Iinitcan = I0 + I0*randn();
             } while (Iinitcan < 0 || N < Iinitcan);
             (*particles)[n].Iinit[loc] = Iinitcan;
         }

     }

 }

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

     */
 void perturbParticles(Particle * particles, int N, int NP, int nloc, int
    passnum, double coolrate) {

     //double coolcoef = exp( - (double) passnum / 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 spreadeta    = coolcoef * etatrue / 10.0;
     double spreadberr   = coolcoef * berrtrue / 10.0;
     double spreadphi    = coolcoef * phitrue / 10.0;

     double R0can, rcan, sigmacan, Iinitcan, etacan, berrcan, phican;

     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 {
             etacan = particles[n].eta + PSC*spreadeta*randn();
         } while (etacan < 0 || etacan > 1);
         particles[n].eta = etacan;

         do {
             berrcan = particles[n].berr + PSC*spreadberr*randn();
         } while (berrcan < 0);
         particles[n].berr = berrcan;

         do {
             phican = particles[n].phi + PSC*spreadphi*randn();
         } while (phican < 0 || phican > 1);
         particles[n].phi = phican;

         for (int loc = 0; loc < nloc; loc++) {
             do {
                 Iinitcan = particles[n].Iinit[loc] + spreadIinit*randn();
             } while (Iinitcan < 0 || Iinitcan > 500);
             particles[n].Iinit[loc] = Iinitcan;
         }
     }
```

```

}

/*  Convinience function for particle resampling process
    */
void copyParticle(Particle * dst, Particle * src, int nloc) {

    dst->R0     = src->R0;
    dst->r      = src->r;
    dst->sigma  = src->sigma;
    dst->eta    = src->eta;
    dst->berr   = src->berr;

    for (int n = 0; n < nloc; n++) {
        dst->S[n]       = src->S[n];
        dst->I[n]       = src->I[n];
        dst->R[n]       = src->R[n];
        dst->B[n]       = src->B[n];
        dst->Iinit[n]   = src->Iinit[n];
    }

}



double randu() {

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

}

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

    double Smean = 0, Imean = 0, Rmean = 0;

    for (int n = 0; n < NP; n++) {
        Smean += particles[n].S;
        Imean += particles[n].I;
        Rmean += particles[n].R;
    }

    state->S = (double) Smean / NP;
    state->I = (double) Imean / NP;
    state->R = (double) Rmean / NP;

}
*/

/*  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
    */
```

```c
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;

}
```