SMAP and SIRS

Dexter Barrows
February 29, 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

Algorithm 1: S-map

```
   /* Select a starting point                                          */
   Input  : Time series $x_1, x_2, ..., x_T$, embedding dimension $E$, distance penalization $\theta$,
            forecast length $L$, predictor vector $\mathbf{x_t}$
   /* Construct library                                                */
1  for $i = E : T$ do
2      $\{\mathbf{x_i}\}$, $\mathbf{x_i} = (x_i, x_{i-1}, ..., x_{i-E-1})$
3  for $i = 1 : (T_E + 1)$ do
4      $b(i) = w(||\mathbf{x_i} - \mathbf{x_t}||)y_i$ for $j = 1 : E$ do
5          $A(i, j) = w(||\mathbf{x_i} - \mathbf{x_t}||)\mathbf{x_i}(j)$
   /* Use SVD to solve Ac = b                                          */
6  $SVD(Ac = b)$
   /* Compute forecast                                                 */
7  $\hat{y}_t = \sum_{j=0}^{E} c_t(j)\mathbf{x_t}(j)$
   /* Forecasted value in time series                                  */
   Output: Forecast $\hat{y}_t$
```

Appendices

A 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)

}
```

B 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

}

parameters {

    real <lower=0, upper=10>        R0;     // R0
    real <lower=0, upper=10>        r;      // recovery rate
    real <lower=0, upper=10>        re;     // resusceptibility rate
    real <lower=0, upper=20>        sigma;  // observation error
    real <lower=0, upper=30>        Iinit;    // initial infected
    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[T];   // Beta vector

}

//transformed parameters {
//    real B0 ← R0 * r / N;
//}

```

```
model {

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

    real pi;
    real Bfac;

    pi ← 3.1415926535;

    B0 ← R0 * r / N;

    B[1] ← B0;

    S[1] ← N - Iinit;
    I[1] ← Iinit;
    R[1] ← 0.0;

    for (t in 2:T) {

        Bnoise[t] ~ normal(0,berr);
        Bfac ← exp(2*cos((2*pi/365)*t) - 2);
        B[t] ← exp( log(B0) + eta * ( log(B[t-1]) - log(B0) ) + Bnoise[t] )
            ;

        S[t] ← S[t-1] + h*( - Bfac*B[t]*S[t-1]*I[t-1] + re*R[t-1] );
        I[t] ← I[t-1] + h*( Bfac*B[t]*S[t-1]*I[t-1] - I[t-1]*r );
        R[t] ← R[t-1] + h*( I[t-1]*r - re*R[t-1] );

        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);

}
```

C 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;

struct State {
    double S;
    double I;
    double R;
};

struct Particle {
    double R0;
    double r;
    double re;
    double sigma;
    double eta;
    double berr;
    double B;
    double S;
    double I;
    double R;
    double Sinit;
    double Iinit;
    double Rinit;
};

struct ParticleInfo {
    double R0mean;      double R0sd;
    double rmean;       double rsd;
    double remean;      double resd;
    double sigmamean;   double sigmasd;
    double etamean;     double etasd;
    double berrmean;    double berrsd;
    double Sinitmean;   double Sinitsd;
    double Iinitmean;   double Iinitsd;
    double Rinitmean;   double Rinitsd;
};


int timeval_subtract (double *result, struct timeval *x, struct timeval *y);
int check_double(double x,double y);
void exp_euler_SIRS(double h, double t0, double tn, int N, Particle * particle);
void copyParticle(Particle * dst, Particle * src);
void perturbParticles(Particle * particles, int N, int NP, int passnum, double coolrate);
void particleDiagnostics(ParticleInfo * partInfo, Particle * particles, int NP);
void getStateMeans(State * state, Particle* particles, int NP);
NumericMatrix if2(NumericVector * data, int T, int N);
double randu();
double randn();

// [[Rcpp::export]]
Rcpp::List if2_sirs(NumericVector data, int T, int N, int NP, int nPasses, double coolrate) {

    int npar = 9;

    NumericMatrix paramdata(NP, npar);
    NumericMatrix means(nPasses, npar);
    NumericMatrix sds(nPasses, npar);
    NumericMatrix statemeans(T, 3);
    NumericMatrix statedata(NP, 4);

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

    double w[NP];           // particle weights

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

    printf("Initializing particle states\n");

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

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

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

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

        do {
            recan = retrue + retrue*randn();
        } while (recan < 0);
        particles[n].re = recan;

        particles[n].B = (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 {
            Iinitcan = I0 + I0*randn();
        } while (Iinitcan < 0 || N < Iinitcan);
        particles[n].Sinit = N - Iinitcan;
        particles[n].Iinit = Iinitcan;
        particles[n].Rinit = 0.0;

    }

    // START PASSES THROUGH DATA

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


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

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

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

            particles[n].S = particles[n].Sinit;
            particles[n].I = particles[n].Iinit;
            particles[n].R = particles[n].Rinit;
            particles[n].B = (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_SIRS(1.0/7.0, (double) t-1, (double) t, N, &
                   particles[n]);

                double merr_par = particles[n].sigma;
                double y_diff   = data[t] - particles[n].I;

                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]);
            }

            // 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;
        sds(pass, 1) = pInfo.rsd;
        sds(pass, 2) = pInfo.resd;
        sds(pass, 3) = pInfo.sigmasd;
        sds(pass, 4) = pInfo.etasd;
        sds(pass, 5) = pInfo.berrsd;
        sds(pass, 6) = pInfo.Sinitsd;
        sds(pass, 7) = pInfo.Iinitsd;
        sds(pass, 8) = pInfo.Rinitsd;

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

    }

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

    printf("Parameter results (mean | sd)\n");
    printf("----------------------------\n");
    printf("R0        %f %f\n", pInfo.R0mean, pInfo.R0sd);
    printf("r         %f %f\n", pInfo.rmean, pInfo.rsd);
    printf("re        %f %f\n", pInfo.remean, pInfo.resd);
    printf("sigma     %f %f\n", pInfo.sigmamean, pInfo.sigmasd);
    printf("eta       %f %f\n", pInfo.etamean, pInfo.etasd);
    printf("berr    %f %f\n", pInfo.berrmean, pInfo.berrsd);
    printf("S_init  %f %f\n", pInfo.Sinitmean, pInfo.Sinitsd);
    printf("I_init    %f %f\n", pInfo.Iinitmean, pInfo.Iinitsd);
    printf("R_init    %f %f\n", pInfo.Rinitmean, pInfo.Rinitsd);

    printf("\n");



    // Get particle results to pass back to R

    for (int n = 0; n < NP; n++) {

        paramdata(n, 0) = particles[n].R0;
        paramdata(n, 1) = particles[n].r;
        paramdata(n, 2) = particles[n].re;
        paramdata(n, 3) = particles[n].sigma;
        paramdata(n, 4) = particles[n].eta;
        paramdata(n, 5) = particles[n].berr;
        paramdata(n, 6) = particles[n].Sinit;
        paramdata(n, 7) = particles[n].Iinit;
        paramdata(n, 8) = particles[n].Rinit;

    }

    for (int n = 0; n < NP; n++) {

        statedata(n, 0) = particles[n].S;
        statedata(n, 1) = particles[n].I;
        statedata(n, 2) = particles[n].R;
        statedata(n, 3) = particles[n].B;

    }



    return Rcpp::List::create(  Rcpp::Named("paramdata") = paramdata,
                                Rcpp::Named("means") = means,
                                Rcpp::Named("statemeans") = statemeans,
                                Rcpp::Named("statedata") = statedata,
                                Rcpp::Named("sds") = sds);

}


/*  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_SIRS(double h, double t0, double tn, int N, Particle *
    particle) {

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

    double B = particle->B;

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

        //double Bfac = 0.5 - 0.95*cos( (2.0*M_PI/365)*(t0*num_steps+i) )
           /2.0;
        double Bfac = exp(2*cos((2*M_PI/365)*(t0*num_steps+i)) - 2);
        B = exp( log(B) + eta*(log(B0) - log(B)) + berr*randn() );

        double BSI = Bfac*B*S*I;
        double rI  = r*I;
        double reR = re*R;

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

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

    }

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

}


```

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

    */
void perturbParticles(Particle * particles, int N, int NP, 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 spreadre     = coolcoef * retrue / 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 R0can, rcan, recan, sigmacan, Iinitcan, etacan, berrcan;

    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 {
            recan = particles[n].re + spreadre*randn();
        } while (recan < 0);
        particles[n].re = recan;

        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 {
            Iinitcan = particles[n].Iinit + spreadIinit*randn();
        } while (Iinitcan < 0 || Iinitcan > 500);
        particles[n].Iinit = Iinitcan;
        particles[n].Sinit = N - Iinitcan;

    }

}


/*  Convinience function for particle resampling process

    */
void copyParticle(Particle * dst, Particle * src) {

    dst->R0     = src->R0;
    dst->r      = src->r;
    dst->re     = src->re;
    dst->sigma  = src->sigma;
    dst->eta    = src->eta;
    dst->berr   = src->berr;
    dst->B      = src->B;
    dst->S      = src->S;
    dst->I      = src->I;
    dst->R      = src->R;
    dst->Sinit  = src->Sinit;
    dst->Iinit  = src->Iinit;
    dst->Rinit  = src->Rinit;

}

void particleDiagnostics(ParticleInfo * partInfo, Particle * particles, int
    NP) {

    double  R0mean      = 0.0,
            rmean       = 0.0,
            remean      = 0.0,
            sigmamean   = 0.0,
            etamean     = 0.0,
            berrmean    = 0.0,
            Sinitmean   = 0.0,
            Iinitmean   = 0.0,
            Rinitmean   = 0.0;

    // means

    for (int n = 0; n < NP; n++) {

        R0mean      += particles[n].R0;
        rmean       += particles[n].r;
        remean      += particles[n].re;
        etamean     += particles[n].eta,
        berrmean    += particles[n].berr,
        sigmamean   += particles[n].sigma;
        Sinitmean   += particles[n].Sinit;
        Iinitmean   += particles[n].Iinit;
        Rinitmean   += particles[n].Rinit;

    }

    R0mean      /= NP;
    rmean       /= NP;
    remean      /= NP;
    sigmamean   /= NP;
    etamean     /= NP;
    berrmean    /= NP;
    Sinitmean   /= NP;
    Iinitmean   /= NP;
    Rinitmean   /= NP;

    // standard deviations

    double  R0sd    = 0.0,
            rsd     = 0.0,
            resd    = 0.0,
            sigmasd = 0.0,
            etasd   = 0.0,
            berrsd  = 0.0,
            Sinitsd = 0.0,
            Iinitsd = 0.0,
            Rinitsd = 0.0;

    for (int n = 0; n < NP; n++) {

        R0sd    += ( particles[n].R0 - R0mean ) * ( particles[n].R0 - R0mean );
        rsd     += ( particles[n].r - rmean ) * ( particles[n].r - rmean );
        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) {

    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
    */
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;

}
```