FUN WITH FORECASTING USING STOCHASTIC NONLINEAR DYNAMICS

Dexter Barrows

Supervisor: Dr. Benjamin Bolker

A thesis presented for the degree of
Master of Science

Department of Mathematics and Statistics
McMaster University
Canada
March 18, 2016

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Dedication

To Mom and Dad

Acknowledgements

Soooooooo many people

Contents

List of Figures

Chapter 1

Introduction

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Chapter 2

Hamiltonian MCMC

2.1 Intro

Markov Chain Monte Carlo (MCMC) is part of a general class of methods designed to sample from the posterior distribution of model parameters. It is an algorithm used when we wish to fit a model M that depends on some parameter (or more typically vector of parameters) θ to observed data D. MCMC works by constructing a Markov Chain whose stationary or equilibrium distribution is used to approximate the desired posterior distribution.

2.2 Markov Chains

Consider a finite state machine with 3 states $S = \{x_1, x_2, x_3\}$, where the probability of transitioning from one particular state to another is shown as a transition graph in Figure [2.1].

The transition probabilities can be summarized as a matrix as

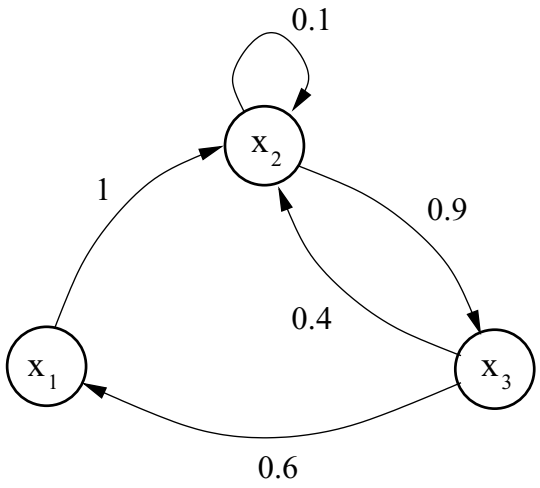

Figure 2.1: Finite state machine. *(Andrieu et al., 2003)*

$$T = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0.1 & 0.9 \\ 0.6 & 0.4 & 0 \end{bmatrix}.$$

The probability vector $\mu(x^{(1)})$ for a state $x^{(1)}$ can be evolved using T by evaluating $\mu(x^{(1)})T$, then again by evaluating $\mu(x^{(1)})T^2$, and so on. If we take the limit as the number of transitions approaches infinity, we find

$$\lim_{t\to\infty} \mu(x^{(1)})T^t = (27/122, 50/122, 45/122).$$

This indicates that no matter what we pick for the initial probability distribution $\mu(x^{(1)})$, the chain will always stabilize at the equilibrium distribution.

Note that this property holds when the chain satisfies the following conditions

- *Irreducible* Any state A can be reached from any other state B with non-zero probability
- *Positive Recurrent* The number of steps required for the chain to reach state A from state B must be finite

- *Aperiodic* The chain must be able to explore the parameter space without becoming trapped in a cycle

Note that MCMC sampling generates a Markov chain $(\theta^{(1)}, \theta^{(2)}, ..., \theta^{(N)})$ that does indeed satisfy these conditions, and uses the chain's equilibrium distribution to approximate the posterior distribution of the parameter space.

2.3 Likelihood

MCMC and similar methods hinge on the idea that the weight or support bestowed upon a particular set of parameters θ should be proportional to the probability of observing the data D given the model output using that set of parameters $M(\theta)$. In order to do this we need a way to evaluate whether or not $M(\theta)$ is a good fit for D; this is done by specifying a likelihood function $\mathcal{L}(\theta)$ such that

$$\mathcal{L}(\theta) \propto P(D|\theta).$$

In standard Maximum Likelihood approaches, $\mathcal{L}(\theta)$ is searched to find a value of θ that maximizes $\mathcal{L}(\theta)$, then this θ is taken to be the most likely true value. Here our aim is to not just maximize the likelihood but to also explore the area around it.

2.4 Prior distribution

Another significant component of MCMC is the user-specified prior distribution for θ or distributions for the individual components of θ (Priors). Priors serve as a way for us to tell the MCMC algorithm what we think consist of good values for the parameters.

Note that if very little is known about the parameters, or we are worried about biasing our estimate of the posterior, we can simply use a a wide uniform distribution. However, this handicaps the algorithm in two ways: convergence of the chain may become exceedingly slow, and more pressure is put on the likelihood function to be as good as possible – it will now be the only thing informing the algorithm of what constitutes a "good" set of parameters, and what should be considered poor.

2.5 Proposal distribution

As part of the MCMC algorithm, when we find a state in the parameter space that is accepted as part of the Markov chain construction process, we need a good way of generating a good next step to try. Unlike basic rejection sampling in which we would just randomly sample from our prior distribution, MCMC attempts to optimise our choices by choosing a step that is close enough to the last accepted step so as to stand a decent chance of also being accepted, but far enough away that it doesn't get "trapped" in a particular region of the parameter space.

This is done through the use of a proposal or candidate distribution. This will usually be a distribution centred around our last accepted step and with a dispersion potential narrower than that of out prior distribution.

Choice of this distribution is theoretically not of the utmost importance, but in practice becomes important so as to not waste computer time.

2.6 Algorithm

Now that we have all the pieces necessary, we can discuss the details of the MCMC algorithm.

We will denote the previously discussed quantities as

- $p(\cdot)$ - the prior distribution
- $q(\cdot|\cdot)$ - the proposal distribution
- $\mathcal{L}(\cdot)$ - the Likelihood function
- $\mathcal{U}(\cdot,\cdot)$ - the uniform distribution

and the define the acceptance ratio, r, as

$$r = \frac{\mathcal{L}(\theta^*)p(\theta^*)q(\theta^*|\theta)}{\mathcal{L}(\theta)p(\theta)q(\theta|\theta^*)},$$

where θ^* is the proposed sample to draw from the posterior, and θ is the last accepted sample.

In the special case of the Metropolis Hastings variation of MCMC, the proposal distribution is symmetric, meaning $q(\theta^*|\theta) = q(\theta|\theta^*)$, and so the acceptance ratio simplifies to

$$r = \frac{\mathcal{L}(\theta^*)p(\theta^*)}{\mathcal{L}(\theta)p(\theta)}.$$

Thus, the MCMC algorithm is as follows.

Algorithm 1: Metropolis-Hastings MCMC

```
/* Select a starting point                                          */
Input : Initialize θ^(1)
for i = 2 : N do
    /* Sample                                                       */
    θ* ~ q(·|θ^(i−1))
    u ~ U(0, 1)
    /* Evaluate acceptance ratio                                    */
    r ← L(θ*)p(θ*) / L(θ)p(θ)
    /* Step acceptance criterion                                    */
    if u < min{1, r} then
        θ^(i) = θ*
    else
        θ^(i) = θ^(i−1)
/* Samples from approximated posterior distribution                 */
Output: Chain of samples (θ^(1), θ^(2), ..., θ^(N))
```

In this way we are ensuring that steps that lead to better likelihood outcomes are likely to be accepted, but steps that do not will not be accepted as frequently. Note that these less "advantageous" moves will still occur but that this is by design – it ensures that as much of the parameter space as possible will be explored but more efficiently than using pure brute force.

2.7 Burn-in

One critical aspect of MCMC-based algorithms has yet to be discussed. The algorithm requires an initial starting point θ to be selected, but as the proposal distribution is supposed to restrict moves to an area close to the last accepted state, then the posterior distribution will be biased towards this starting point. This issue is avoided through the use of a Burn-in period.

Burning in a chain is the act of running the MCMC algorithm normally without saving first M samples. As we are seeking a chain of length N, the total computation will be equivalent to generating a chain of length $M + N$.

2.8 Thinning

Some models will require very long chains to get a good approximation of the posterior, which will consequently require a non-trivial amount of computer storage. One way to reduce the burden of storing so many samples is by thinning. This involves saving only every n^{th} step, which should still give a decent approximate of the posterior (since the chain has time to explore a large portion of the parameter space), but require less room to store.

2.9 Hamiltonian Monte Carlo

The Metropolis-Hastings algorithm has a primary drawback in that the parameter space may not be explored efficiently – a consequence of the rudimentary proposal mechanism. Instead, smarter moves can be proposed through the use of Hamiltonian dynamics, leading to a better exploration of the target distribution and a decrease in overall computational complexity.

From physics, we will borrow the ideas of potential and kinetic energy. Here potential energy is analogous to the negative log likelihood of the parameter selection given the data, formally

$$U(\theta) = -log(\mathcal{L}(\theta)p(\theta)). \tag{2.1}$$

Kinetic energy will serve as a way to "nudge" the parameters along a different moment for each component of θ. We introduce n auxiliary variables $r = (r_1, r_1, ..., r_n)$, where n is the number of components in θ. Note that the samples drawn for r are not of interest, they are only used to inform the evolution of the Hamiltonian dynamics of the system. We can now define the kinetic energy as

$$K(r) = \frac{1}{2} r^T M^{-1} r, \tag{2.2}$$

where M is an $n \times n$ matrix. In practice M can simply be chosen as the identity matrix of size n, however it can also be used to account for correlation between components of θ.

The Hamiltonian of the system is defined as

$$H(\theta, r) = U(\theta) + K(r), \tag{2.3}$$

Where the Hamiltonian dynamics of the combined system can be simulated using the following system of ODEs.

$$\begin{aligned} \frac{d\theta}{dt} &= M^{-1} r \\ \frac{dr}{dt} &= -\nabla U(\theta) \end{aligned} \tag{2.4}$$

.

It is tempting to try to integrate this system using the standard Euler evolution scheme, but in practice this leads to instability. Instead the "Leapfrog" scheme is used. This scheme is very similar to Euler scheme, except instead of using a fixed step size h for all evolutions, a step size of ε is used for most evolutions, with a half step size of $\varepsilon/2$ for evolutions of $\frac{dr}{dt}$ at the first step, and last step L. In this way the evolution steps "leapfrog" over each other while using future values from the other set of steps, leading to the scheme's name.

The end product of the Leapfrog steps are the new proposed parameters (θ^*, r^*). These are either accepted or rejected using a mechanism similar to that of standard Metropolis-Hastings MCMC. Now, however, the acceptance ratio r is defined as

$$r = \exp\left[H(\theta, r) - H(\theta^*, r^*)\right], \tag{2.5}$$

where (θ, r) are the last values in the chain.

Together, we have Algorithm 2.

Note that the parameters ε and L have to be tuned in order to maintain stability and maximize efficiency, a sometimes non-trivial process.

2.10 Fitting

Here we will examine a test case in which Hamiltonian MCMC will be used to fit a Susceptible-Infected-Removed (SIR) epidemic model to mock infectious count data.

The synthetic data was produced by taking the solution to a basic SIR ODE model, sampling it at regular intervals, and perturbing those values by adding in observation noise. The SIR model used was

$$\begin{aligned} \frac{dS}{dt} &= -\beta IS \\ \frac{dI}{dt} &= \beta IS - rI \\ \frac{dR}{dt} &= rI \end{aligned} \tag{2.6}$$

where S is the number of individuals susceptible to infection, I is the number of infectious individuals, R is the number of recovered individuals, $\beta = R_0 r/N$ is the force of infection, R_0 is the number of secondary cases per infected individual, r is the recovery rate, and N is the population size.

The solution to this system was obtained using the `ode()` function from the `deSolve` package. The required derivative array function in the format required by `ode()` was specified as

```
1  SIR ← function(Time, State, Pars) {
2
3      with(as.list(c(State, Pars)), {
```

Algorithm 2: Hamiltonian MCMC

/* Select a starting point */
Input : Initialize $\theta^{(1)}$

for $i = 2 : N$ **do**

/* Resample moments */
for $i = 1 : n$ **do**
r(i) $\leftarrow \mathcal{N}(0, 1)$

/* Leapfrog initialization */
$\theta_0 \leftarrow \theta^{(i-1)}$
$r_0 \leftarrow r - \nabla U(\theta_0) \cdot \varepsilon/2$

/* Leapfrog intermediate steps */
for $j = 1 : L - 1$ **do**
$\theta_j \leftarrow \theta_{j-1} + M^{-1} r_{j-1} \cdot \varepsilon$
$r_j \leftarrow r_{j-1} - \nabla U(\theta_j) \cdot \varepsilon$

/* Leapfrog last steps */
$\theta^* \leftarrow \theta_{L-1} + M^{-1} r_{L-1} \cdot \varepsilon$
$r^* \leftarrow \nabla U(\theta_L) \cdot \varepsilon/2 - r_{L-1}$

/* Evaluate acceptance ratio */
$r = \exp\left[H(\theta^{(i-1)}, r) - H(\theta^*, r^*)\right]$

/* Sample */
$u \sim \mathcal{U}(0, 1)$

/* Step acceptance criterion */
if $u < \min\{1, r\}$ **then**
$\theta^{(i)} = \theta^*$
else
$\theta^{(i)} = \theta^{(i-1)}$

/* Samples from approximated posterior distribution */
Output: Chain of samples $(\theta^{(1)}, \theta^{(2)}, ..., \theta^{(N)})$

```
4
5             B   ← R0*r/N     # calculate Beta
6             BSI ← B*S*I      # save product
7             rI  ← r*I        # save product
8
9             dS = -BSI        # change in Susceptible people
10            dI = BSI - rI    # change in Infected people
11            dR = rI          # change in Removed (recovered people)
12
13            return(list(c(dS, dI, dR)))
14
15        })
16
17    }
```

The true parameter values were set to $R_0 = 3.0, r = 0.1, N = 500$ by

```
1     pars  ← c(R0  ← 3.0,  # new infected people per infected person
2                r   ← 0.1,  # recovery rate
3                N   ← 500)  # population size
```

The system was integrated over $[0, 100]$ with infected counts drawn at each integer time step. These timings were set using

```
1     T ← 100                              # total integration time
2     times ← seq(0, T, by = 1)            # times to draw solution
          values
```

The initial conditions were set to 5 infectious individuals, 495 people susceptible to infection, and no one had yet recovered from infection and been removed. These were set using

```
1     y_ini ← c(S = 495, I = 5, R = 0)   # initial conditions
```

The `ode()` function is called as

```
1     odeout ← ode(y_ini, times, SIR, pars)
```

where `odeout` is a $(T + 1) \times 4$ matrix where the rows correspond to solutions at the given times (the first row is the initial condition), and the columns correspond to the solution times and S-I-R counts at those times.

The observation error was taken to be $\varepsilon_{obs} \sim \mathcal{N}(0, \sigma)$, where individual values were drawn for each synthetic data point.

These "true" values were perturbed to mimic observation error by

```
set.seed(1001)  # set RNG seed for reproducibility
sigma ← 5       # observation error standard deviation
infec_counts_raw ← odeout[,3] + rnorm(101, 0, sigma)
infec_counts     ← ifelse(infec_counts_raw < 0, 0, infec_counts)
```

where the last two lines simply set negative observations (impossible) to 0.

Plotting the data using the `ggplot2` package by

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

print(g)
```

we obtain

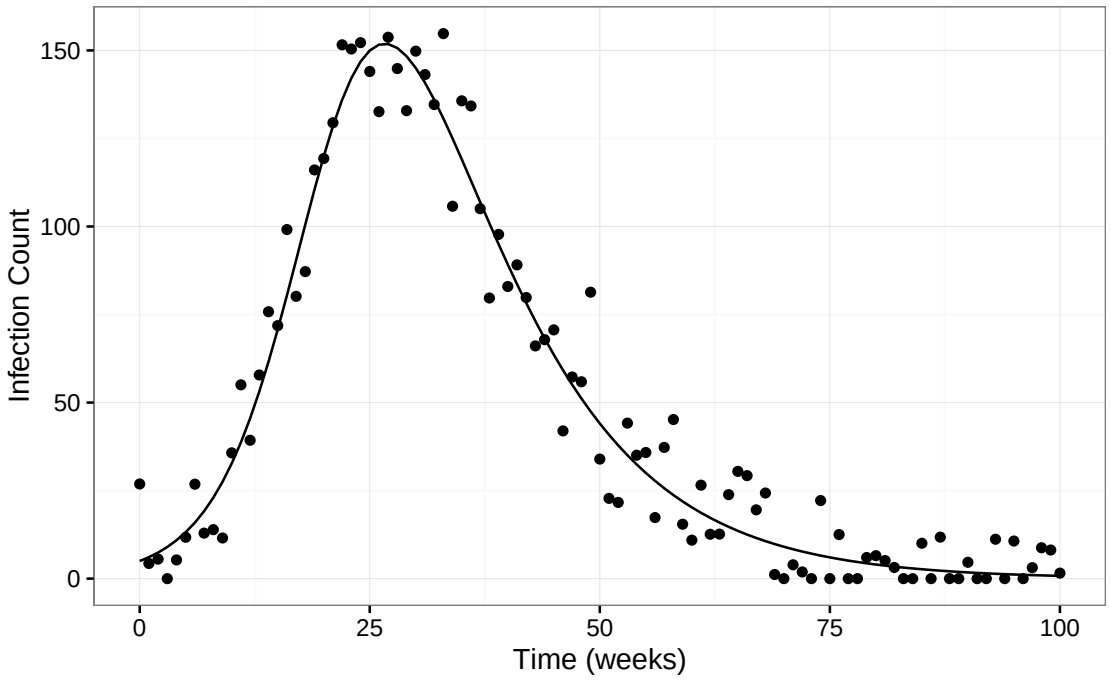

Figure 2.2: True SIR ODE solution infected counts, and with added observation noise

The Hamiltonian MCMC model fitting was done using Stan (`http://mc-stan.org/`), a program written in `C++` that does Baysian statistical inference using Hamil-

tonian MCMC. Stan's R interface (`http://mc-stan.org/interfaces/rstan.html`) was used to ease implementation.

In order to use an Explicit Euler-like stepping method in the later Stan model (both for speed and for integration method homogeneity with other methods against which HMCMC was compared), the synthetic observation counts were treated as weekly observations in which the counts on the other six days of the week were unobserved. For computational and organizational simplicity, these vales were set to -1 (all valid observations are non-negative). This is done in R using

```
sPw ← 7                # steps per week
datlen ← (T-1)*7 + 1   # size of sparse data vector

data ← matrix(data = -1, nrow = T+1, ncol = sPw)
data[,1] ← infec_counts
standata ← as.vector(t(data))[1:datlen]
```

The data to be fed into the R Stan interface is packed as

```
sir_data ← list( T = datlen,     # simulation time
                 y = standata,   # infection count data
                 N = 500,        # population size
                 h = 1/sPw )     # step size per day
```

For efficiency we allow Stan to save compiled code to avoid recompilation, and allow multiple chains to be run simultaneously on separate CPU cores

```
rstan_options(auto_write = TRUE)
options(mc.cores = parallel::detectCores())
```

Now we call the Stan fitting function

```
stan_options ← list(    chains = 4,    # number of chains
                        iter   = 2000, # iterations per chain
                        warmup = 1000, # warmup interations
                        thin   = 2 )   # thinning number
fit ← stan( file    = "d_sirode_euler.stan",
            data    = sir_data,
            chains  = stan_options$chains,
            iter    = stan_options$iter,
            warmup  = stan_options$warmup,
            thin    = stan_options$thin )
```

which fits the model in the file `d_sirode_euler.stan` to the data passed in through `sir_data`. The options here specify that 10 chains will be run, each with a burn in

period of 1000 steps, with 5000 steps to sample over, and only sampling every 10th step. Options are saved so they can be accessed later.

The Stan file contains three blocks that together specify the model. First, the data block specifies the information the model expects to be given. Here, this is

```
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

}
```

where each of the data variables correspond to data passed in through the previously shown R code.

Next the parameters block specifies what Stan is expected to estimate. Here this is

```
parameters {

    real <lower=0, upper=10>    sigma;  // observation error
    real <lower=0, upper=10>    R0;     // R0
    real <lower=0, upper=10>    r;      // recovery rate
    real <lower=0, upper=500>   y0[3];  // initial conditions

}
```

Finally we have the model block. This crucial part of the code specifies the interaction between the parameters and the data. The core component of the model indicates we are fitting an approximation of an ODE model using Euler integration steps (one per day), with the initial conditions and SIR parameters unknown. Further, we can also specify the prior distributions to draw new parameter values from. The initial conditions are taken to be close to the initial data point, with adjustment for observation error, while the other parameters are assumed to be coming from log-normal distributions with relatively small means. Together, we have

```
model {

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

    S[1] <- y0[1];
```

```
        I[1] <- y0[2];
        R[1] <- y0[3];

        y[1] ~ normal(y0[2], sigma);

        for (t in 2:T) {

            S[t] <- S[t-1] + h*( - S[t-1]*I[t-1]*R0*r/N );
            I[t] <- I[t-1] + h*( S[t-1]*I[t-1]*R0*r/N  - I[t-1]*r );
            R[t] <- R[t-1] + h*( I[t-1]*r );

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

        }

        y0[1] ~ normal(N - y[1], sigma);
        y0[2] ~ normal(y[1], sigma);

        theta[1]    ~ lognormal(1,1);
        theta[2]    ~ lognormal(1,1);
        sigma       ~ lognormal(1,1);

    }
```

Examining the traceplot for the the post-warmup chain data returned by the `stan()` function in the `fit` object, we see that the chains are mixing well and convergence has likely been reached. This is shown below in Figure [2.3].

Further, if we look at the chain data including the warmup samples in Figure [2.4], we can see why is is wise to discard these samples (note the scale).

Now if we look at the kernel density estimates for each of the model parameters and the initial number of cases, we see that while the estimates are not perfect, they are fairly decent. This is shown below in Figure [2.5].

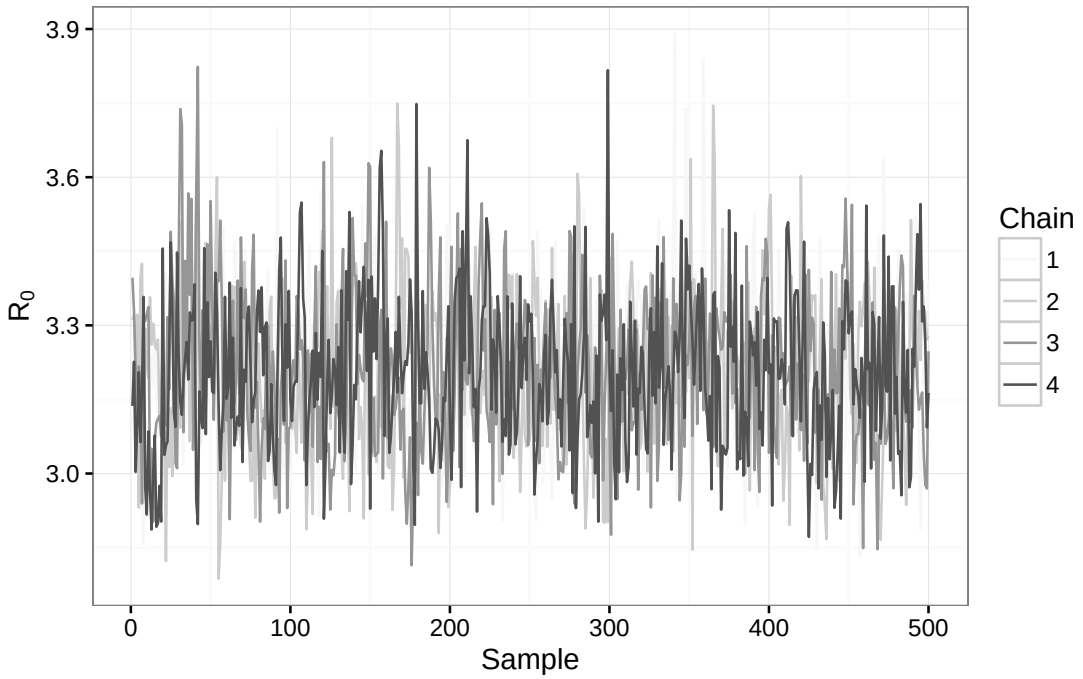

Figure 2.3: Traceplot of samples drawn for parameter R_0, excluding warmup

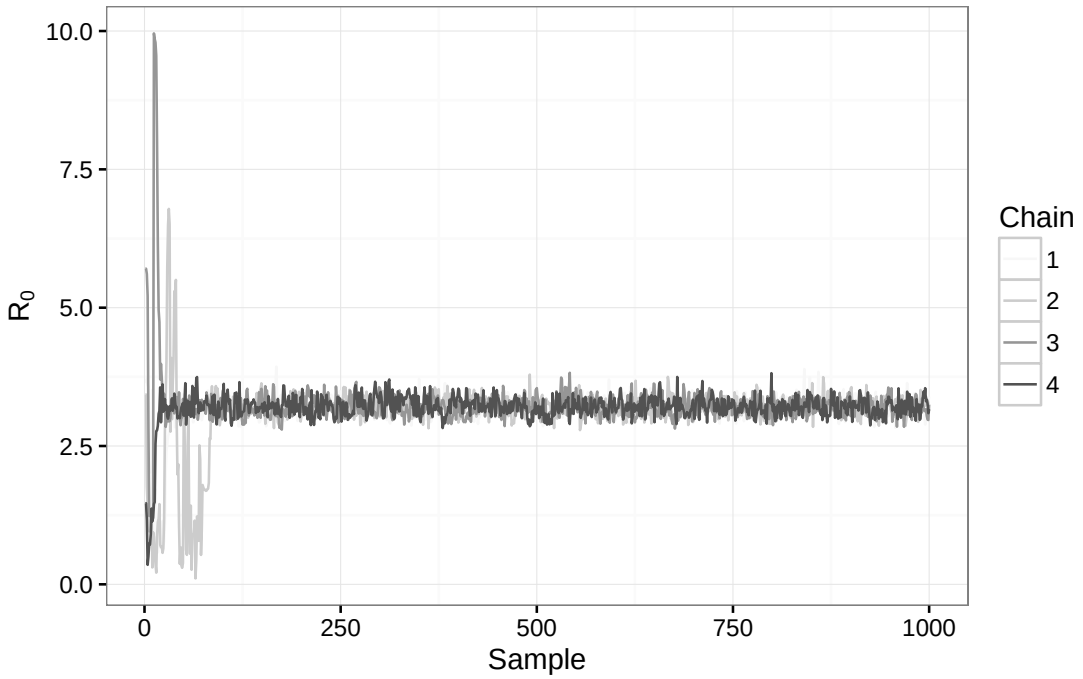

Figure 2.4: Traceplot of samples drawn for parameter R_0, including warmup.

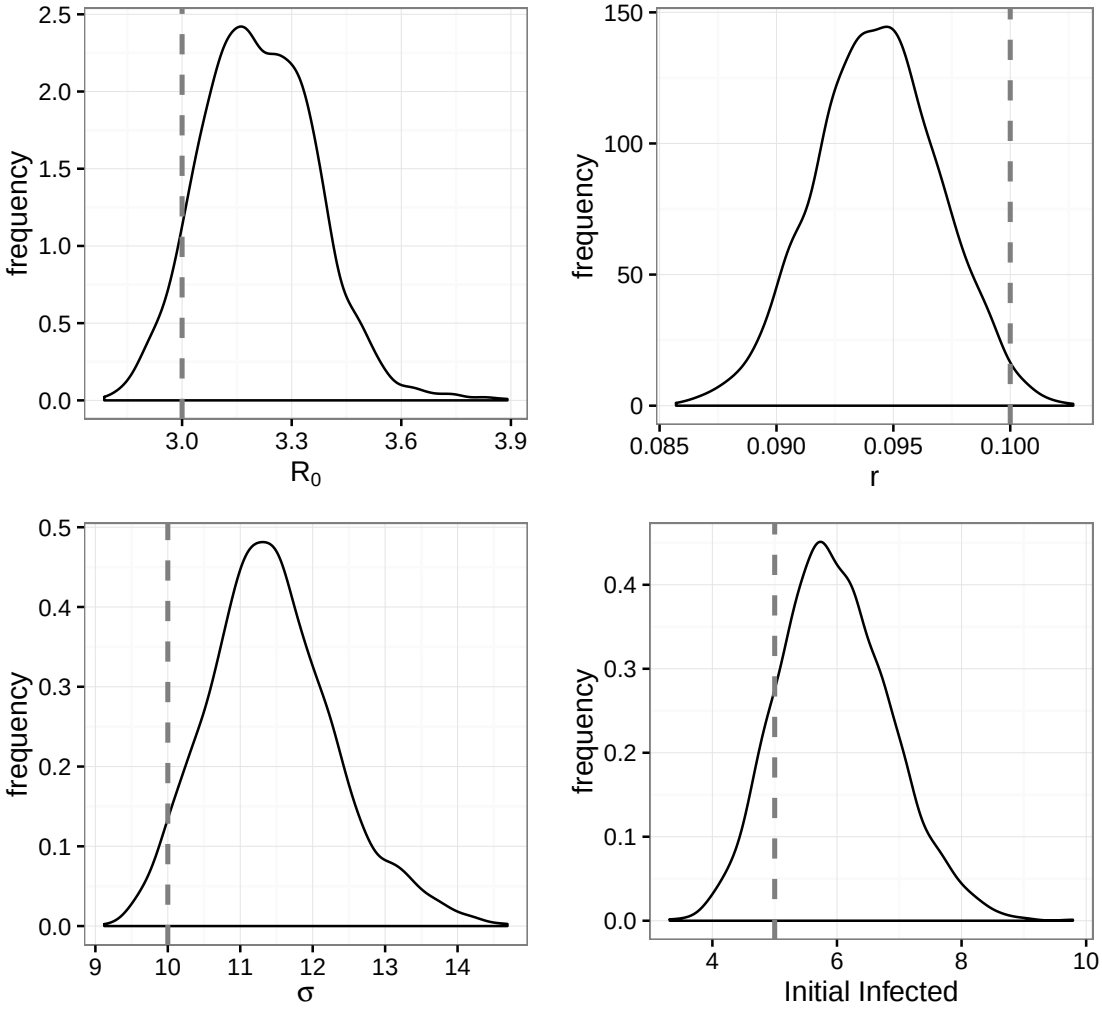

Figure 2.5: Kernel density estimates produced by Stan

Chapter 3

Iterated Filtering

3.1 Intro

Particle filters are similar to MCMC-based methods in that they attempt to draw samples from an approximation of the posterior distribution of model parameters θ given observed data D. Instead of constructing a Markov chain and approximating its stationary distribution, a cohort of "particles" are used to move through the data in an on-line (sequential) fashion with the cohort being culled of poorly-performing particles at each iteration via importance sampling. If the culled particles are not replenished, this will be a Sequential Importance Sampling (SIS) particle filter. If the culled particles are replenished from surviving particles, in a sense setting up a process not dissimilar from Darwinian selection, then this will be a Sequential Importance Resampling (SIR) particle filter.

3.2 Formulation

Particle filters, also called Sequential Monte-Carlo (SMC) or bootstrap filters, feature similar core functionality as the venerable Kalman Filter. As the algorithm moves through the data (sequence of observations), a prediction-update cycle is used to simulate the evolution of the model M with different particular parameter selections, track how closely these predictions approximate the new observed value, and update the current cohort appropriately.

Two separate functions are used to simulate the evolution and observation processes. The "true" state evolution is specified by

$$X_{t+1} \sim f_1(X_t, \theta), \tag{3.1}$$

And the observation process by

$$Y_t \sim f_2(X_t, \theta). \tag{3.2}$$

Note that components of θ can contribute to both functions, but a typical formulation is to have some components contribute to $f_1(\cdot, \theta)$ and others to $f_2(\cdot, \theta)$.

The prediction part of the cycle utilises $f_1(\cdot, \theta)$ to update each particle's current state estimate to the next time step, while $f_2(\cdot, \theta)$ is used to evaluate a weighting w for each particle which will be used to determine how closely that particle is estimating the true underlying state of the system. Note that $f_2(\cdot, \theta)$ could be thought of as a probability of observing a piece of data y_t given the particle's current state estimate and parameter set, $P(y_t|X_t, \theta)$. Then, the new cohort of particles is drawn from the old cohort proportional to the weights. This process is repeated until the set of observations D is exhausted.

3.3 Algorithm

Now we will formalize the particle filter.

We will denote each particle $p^{(j)}$ as the j^{th} particle consisting of a state estimate at time t, $X_t^{(j)}$, a parameter set $\theta^{(j)}$, and a weight $w^{(j)}$. Note that the state estimates will evolve with the system as the cohort traverses the data.

The algorithm for a Sequential Importance Resampling particle is shown in Algorithm 3.

Algorithm 3: SIR particle filter

/* Select a starting point */

Input : Observations $D = y_1, y_2, ..., y_T$, initial particle distribution P_0 of size J

/* Setup */

Initialize particle cohort by sampling $(p^{(1)}, p^{(2)}, ..., p^{(J)})$ from P_0

for $t = 1 : T$ **do**

  /* Evolve */

  for $j = 1{:}J$ **do**

    $X_t^{(j)} \leftarrow f_1(X_{t-1}^{(j)}, \theta^{(j)})$

  /* Weight */

  for $j = 1{:}J$ **do**

    $w^{(j)} \leftarrow P(y_t | X_t^{(j)}, \theta^{(j)}) = f_2(X_t^{(j)}, \theta^{(j)})$

  /* Normalize */

  for $j = 1{:}J$ **do**

    $w^{(j)} \leftarrow w^{(j)} / \sum_1^J w^{(j)}$

  /* Resample */

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

/* Samples from approximated posterior distribution */

Output: Cohort of posterior samples $(\theta^{(1)}, \theta^{(2)}, ..., \theta^{(J)})$

3.4 Particle Collapse

Not uncommonly, a situation may arise in which a single particle is assigned a normalized weight very close to 1 and all the other particles are assigned weights very close to 0. When this occurs, the next generation of the cohort will overwhelmingly consist of descendants of the heavily-weighted particle, termed particle collapse or degeneracy.

Since the basic SIR particle filter does not perturb either the particle system states or system parameter values, the cohort will quickly consist solely of identical particles, effectively halting further exploration of the parameter space as new data is introduced.

A similar situation occurs when a small number of particles (but not necessarily a single particle) split almost all of the normalized weight between them, then jointly dominate the resampling process for the remainder of the iterations. This again halts the exploration of the parameter space with new data.

In either case, the hallmark feature used to detect collapse is the same – at some point the cohort will consist of particles with very similar or identical parameter sets which will consequently result in their assigned weights being extremely close together.

Mathematically, we are interested in the number of effective particles, N_{eff}, which represents the number of particles that are acceptably dissimilar. This is estimated by evaluating

$$N_{eff} = \frac{1}{\sum_1^J (w^{(j)})^2}. \tag{3.3}$$

This can be used to diagnose not only when collapse has occurred, but can also indicate when it is near.

3.5 Iterated Filtering and Data Cloning

A particle filter hinges on the idea that as it progresses through the data set D, its estimate of the posterior carried in the cohort of particles approaches maximum

likelihood. However, this convergence may not be fast enough so that the estimate it produces is of quality before the data runs out. One way around this problem is to "clone" the data and make multiple passes through it as if it were a continuation of the original time series. Note that the system state contained in each particle will have to be reset with each pass.

Rigorous proofs have been developed (references to Ionides et. al. work) that show that by treating the parameters as stochastic processes instead of fixed values, the multiple passes through the data will indeed force convergence of the process mean toward maximum likelihood, and the process variance toward 0.

3.6 IF2

The successor to Iterated Filtering 1 (reference), Iterated Filtering 2 is simpler, faster, and demonstrated better convergence toward maximum likelihood (reference). The core concept involves a two-pronged approach. First, Data cloning is used to allow more time for the parameter stochastic process means to converge to maximum likelihood, and frequent cooled perturbation of the particle parameters allow better exploration of the parameter space while still allowing convergence to good point estimates.

It is worth noting that IF2 is not designed to estimate the full posterior distribution, but in practice can be used to do so within reason. Further, IF2 thwarts the problem of particle collapse by keeping at least some perturbation in the system at all times. It is important to note that while true particle collapse will not occur, there is still risk of a pseudo-collapse in which all particles will be extremely close to one another so as to be virtually indistinguishable. However this will only occur with the use of overly-aggressive cooling strategies or by specifying an excessive number of passes through the data.

An important new quantity is the particle perturbation density denoted $h(\theta|, \sigma)$. Typically this is multi-normal with σ being a vector of variances proportional to the expected values of θ. In practice the proportionality can be derived from current means or specified ahead of time. Further, these intensities must decrease over time. This can be done via exponential or geometric cooling, a decreasing step function, a combination of these, or though some other similar scheme.

The algorithm for IF2 can be seen in Algorithm 4.

Algorithm 4: IF2

/* Select a starting point */

Input : Observations $D = y_1, y_2, ..., y_T$, initial particle distribution P_0 of size J, decreasing sequence of perturbation intensity vectors $\sigma_1, \sigma_2, ..., \sigma_M$

/* Setup */

Initialize particle cohort by sampling $(p^{(1)}, p^{(2)}, ..., p^{(J)})$ from P_0

/* Particle seeding distribution */

$\Theta \leftarrow P_0$

for $m = 1 : M$ **do**

  /* Pass perturbation */

  for $j = 1{:}J$ **do**

    $p^{(j)} \sim h(\Theta^{(j)}, \sigma_m)$

  for $t = 1 : T$ **do**

    for $j = 1{:}J$ **do**

      /* Iteration perturbation */

      $p^{(j)} \sim h(p^{(j)}, \sigma_m)$

      /* Evolve */

      $X_t^{(j)} \leftarrow f_1(X_{t-1}^{(j)}, \theta^{(j)})$

      /* Weight */

      $w^{(j)} \leftarrow P(y_t | X_t^{(j)}, \theta^{(j)}) = f_2(X_t^{(j)}, \theta^{(j)})$

    /* Normalize */

    for $j = 1{:}J$ **do**

      $w^{(j)} \leftarrow w^{(j)} / \sum_1^J w^{(j)}$

    /* Resample */

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

  /* Collect particles for next pass */

  for $j = 1 : J$ **do**

    $\Theta^{(j)} \leftarrow p^{(j)}$

/* Samples from approximated posterior distribution */

Output: Cohort of posterior samples $(\theta^{(1)}, \theta^{(2)}, ..., \theta^{(J)})$

3.7 Fitting

Here we will examine a test case in which IF2 will be used to fit a Susceptible-Infected-Removed (SIR) epidemic model to mock infectious count data.

The synthetic data was produced by taking the solution to a basic SIR ODE model, sampling it at regular intervals, and perturbing those values by adding in observation noise. The SIR model used was

$$\begin{aligned} \frac{dS}{dt} &= -\beta IS \\ \frac{dI}{dt} &= \beta IS - rI \\ \frac{dR}{dt} &= rI \end{aligned} \tag{3.4}$$

where S is the number of individuals susceptible to infection, I is the number of infectious individuals, R is the number of recovered individuals, $\beta = R_0 r/N$ is the force of infection, R_0 is the number of secondary cases per infected individual, r is the recovery rate, and N is the population size.

The solution to this system was obtained using the `ode()` function from the `deSolve` package. The required derivative array function in the format required by `ode()` was specified as

```
SIR ← function(Time, State, Pars) {

    with(as.list(c(State, Pars)), {

        B   ← R0*r/N    # calculate Beta
        BSI ← B*S*I     # save product
        rI  ← r*I       # save product

        dS = -BSI       # change in Susceptible people
        dI = BSI - rI   # change in Infected people
        dR = rI         # change in Removed (recovered people)

        return(list(c(dS, dI, dR)))

    })

}
```

The true parameter values were set to $R_0 = 3.0, r = 0.1, N = 500$ by

```
pars  ← c(R0  = 3.0,  # new infected people per infected person
          r   = 0.1,  # recovery rate
          N   = 500)  # population size
```

The initial conditions were set to 5 infectious individuals, 495 people susceptible to infection, and no one had yet recovered from infection and been removed. These were set using

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

The `ode()` function is called as

```
odeout ← ode(y = true_init_cond, times = 0:(T-1), func = SIR,
   parms = true_pars)
```

where `odeout` is a $T \times 4$ matrix where the rows correspond to solutions at the given times (the first row is the initial condition), and the columns correspond to the solution times and S-I-R counts at those times.

The observation error was taken to be $\varepsilon_{obs} \sim \mathcal{N}(0, \sigma)$, where individual values were drawn for each synthetic data point.

These "true" values were perturbed to mimic observation error by

```
set.seed(1001)  # set RNG seed for reproducibility
sigma ← 10      # observation error standard deviation
infec_counts_raw ← odeout[,3] + rnorm(101, 0, sigma)
infec_counts     ← ifelse(infec_counts_raw < 0, 0, infec_counts)
```

where the last two lines simply set negative observations (impossible) to 0.

Plotting the data using the `ggplot2` package by

```
plotdata ← data.frame(times=1:T,true=trueTraj,data=infec_counts)

g ← ggplot(plotdata, aes(times)) +
      geom_line(aes(y = true, colour = "True")) +
      geom_point(aes(y = data, color = "Data")) +
      labs(x = "Time", y = "Infection count", color = "") +
      scale_color_brewer(palette="Paired") +
      theme(panel.background = element_rect(fill = "#F0F0F0"))
```

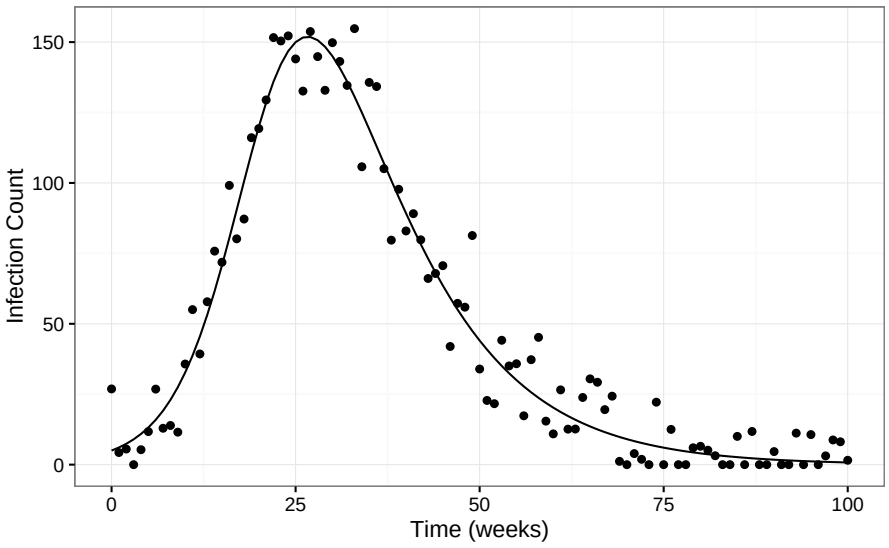

Figure 3.1: True SIR ODE solution infected counts, and with added observation noise.

we obtain Figure [7.1].

The IF2 algorithm was implemented in C++ for speed, and integrated into the R workflow using the Rcpp package. The C++ code is compiled using

```
sourceCpp(paste(getwd(),"if2.cpp",sep="/"))
```

Then run and packed into a data frame using

```
paramdata ← data.frame(if2(infec_counts[1:Tlim], Tlim, N))
colnames(paramdata) ← c("R0", "r", "sigma", "Sinit", "Iinit", "
    Rinit")
```

The final kernel estimates for four of the key parameters are shown in Figure [3.2].

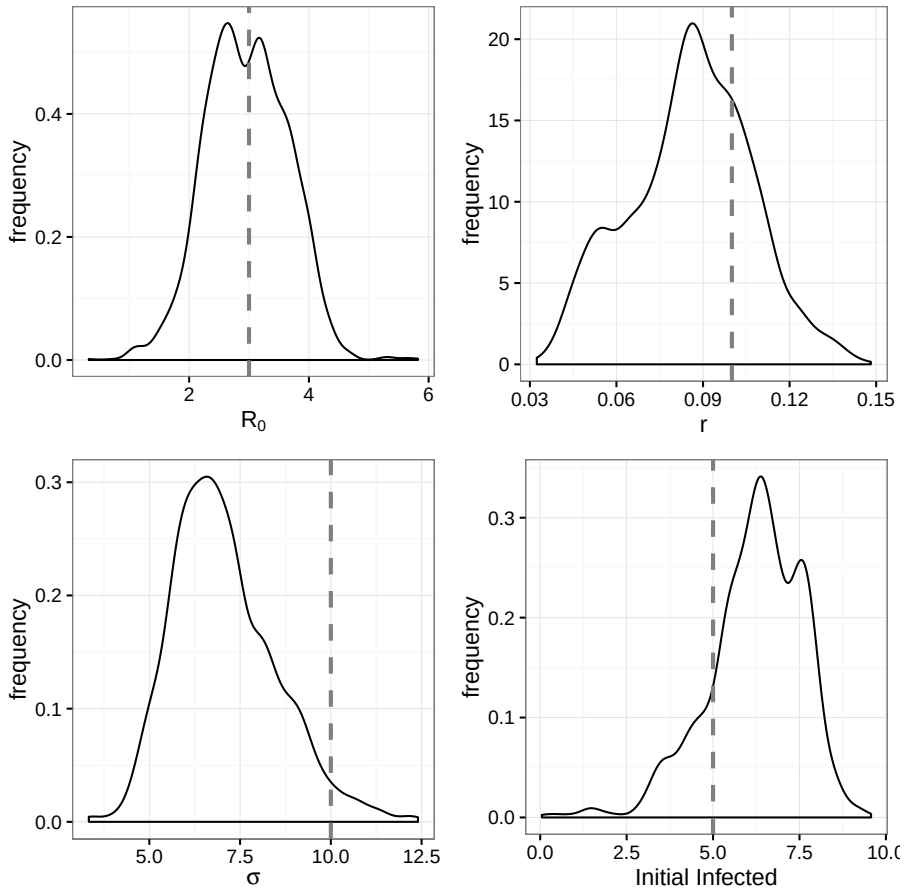

Figure 3.2: Kernel estimates for four essential system parameters. True values are indicated by solid vertical lines, sample means by dashed lines.

Chapter 4

Parameter Fitting

4.1 Fitting Setup

Now that we have established which methods we wish to evaluate the efficacy of for epidemic forecasting, it is prudent to see how they perform when fitting parameters for a known epidemic model. We have already seen how they perform when fitting parameters for a model with a deterministic evolution process and observation noise, but a more realistic model will have both process and observation noise.

To form such a model, we will take a deterministic SIR ODE model given by

$$\begin{aligned} \frac{dS}{dt} &= -\beta SI \\ \frac{dI}{dt} &= \beta SI - \gamma I \\ \frac{dR}{dt} &= \gamma I, \end{aligned} \tag{4.1}$$

and add process noise by allowing β to embark on a geometric random walk given by

$$\beta_{t+1} = \exp\left(\log(\beta_t) + \eta(\log(\bar{\beta}) - \log(\beta_t)) + \epsilon_t\right). \tag{4.2}$$

We will take ϵ_t to be normally distributed with standard deviation ρ^2 such that

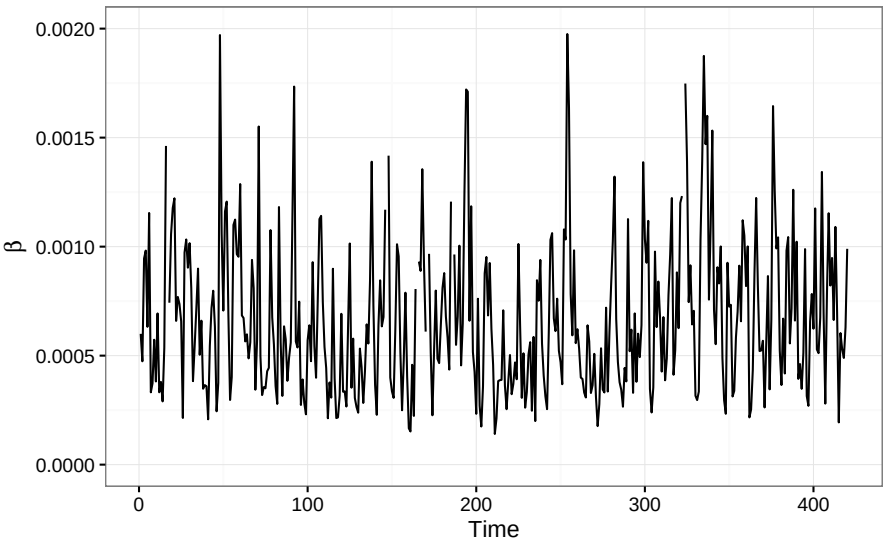

Figure 4.1: Simulated geometric autoregressive process show in Equation [4.2].

$\epsilon_t \sim \mathcal{N}(0, \rho^2)$. The geometric attraction term constrains the random walk, the force of which is $\eta \in [0, 1]$. If we take $\eta = 0$ then the walk will be unconstrained; if we let $\eta = 1$ then all values of β_t will be independent from the previous value (and consequently all other values in the sequence).

When $\eta \in (0, 1)$, we have an autoregressive process of order 1 on the logarithmic scale of the form

$$X_{t+1} = c + \rho X_t + \epsilon_t, \tag{4.3}$$

where ϵ_t is normally distributed noise with mean 0 and standard deviation σ_E. This process has a theoretical expected mean of $\mu = c/(1 - \rho)$ and variance $\sigma = \sigma_E^2/(1 - \rho^2)$. If we choose $\eta = 0.5$, the resulting log-normal distribution has a mean of 6.80×10^{-4} and standard deviation of 4.46×10^{-4}.

Simulating the process in Equation [4.2] with $\eta = 0.5$ gives us the plot in Figure [4.1] below.

We can obtain the corresponding density plot of the values in Figure [4.1], shown in Figure [4.2].

We see a density plot similar in shape to the desired density, and the geometric

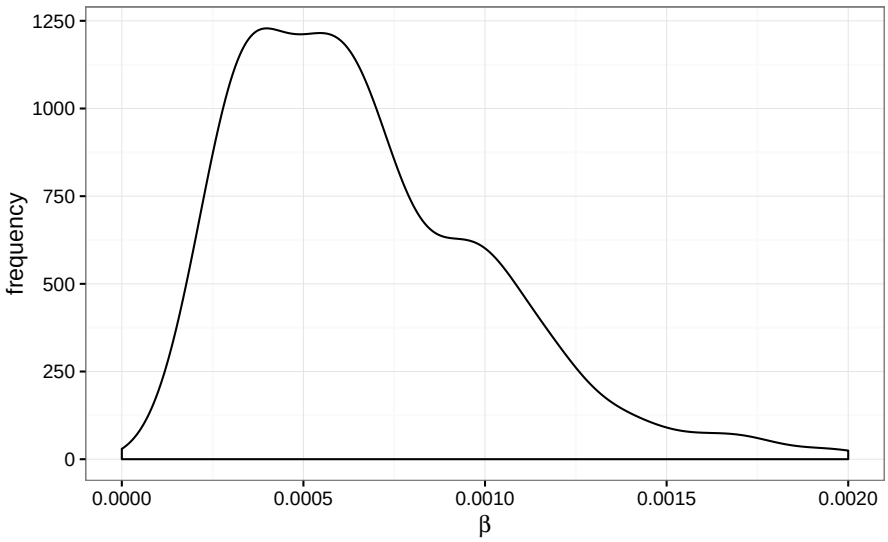

Figure 4.2: Density plot of values shown if Figure[4.1].

random walk displays dependence on previous values. Further the mean of this distribution was calculated to be 6.92×10^{-4} and standard the deviation to be 3.99×10^{-4}, which are very close to the theoretical values.

If we take the full stochastic SIR system and evolve it using an Euler stepping scheme with a step size of $h = 1/7$, for 1 step per day, we obtain the plot in Figure [4.3].

4.2 Calibrating Samples

In order to compare HMCMC and IF2 we need to set up a fair and theoretically justified way to select the number of samples to draw for the HMCMC iterations and the number of particles to use for IF2. We assume that we are working with a problem that has an unknown real solution, so we use the Monte Carlo Standard Error (MCSE).

Suppose we are using a Monte-Carlo based method to obtain an estimate $\hat{\mu}_n$ for a quantity μ, where n is the number of samples. Then the Law of Large Numbers says that $\hat{\mu}_n \to \mu$ as $n \to \infty$. Further, the Central Limit Theorem says that the error $\hat{\mu}_n - \mu$ should shrink with number of samples such that $\sqrt{n}(\hat{\mu}_n - \mu) \to \mathcal{N}(0, \sigma^2)$ as $n \to \infty$, where σ^2 is the variance of the samples drawn.

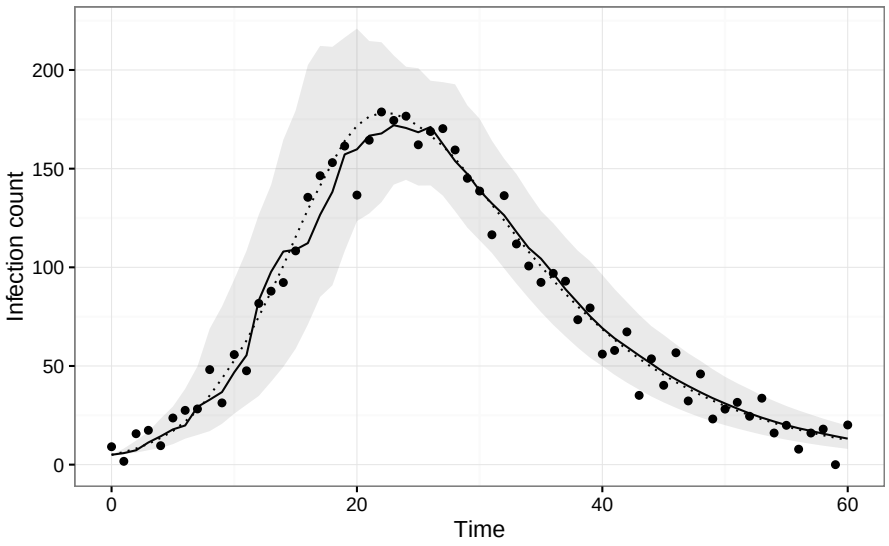

Figure 4.3: Stochastic SIR model simulated using an explicit Euler stepping scheme. The solid line is a single random trajectory, the dots show the data points obtained by adding in observation error defined as $\epsilon_{obvs} = \mathcal{N}(0, 10)$, and the grey ribbon is centre 95th quantile from 100 random trajectories.

We of course do not know μ, but the above allows us to obtain an estimate $\hat{\sigma}_n$ for σ given a number of samples n as

$$\hat{\sigma}_n = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(X_i - \hat{\mu})}, \tag{4.4}$$

which is known as the Monte Carlo Standard Error.

We can modify this formula to account for multiple variables by replacing the single variance measure sum by

$$\Theta^* V (\Theta^*)^T \tag{4.5}$$

where Θ^* is a row vector containing the reciprocals of the means of the parameters of interest, and V is the variance-covariance matrix with respect to the same parameters. This in effect scales the variances with respect to their magnitudes and accounts for covariation between parameters in one fell swoop. We also divide by

the number of parameters, yielding

$$\hat{\sigma}_n = \sqrt{\frac{1}{n}\frac{1}{P}\Theta^* V (\Theta^*)^T} \tag{4.6}$$

where P is the number of particles.

The goal here is to then pick the number of HMCMC samples and IF2 particles to yield similar MCSE values. To do this we picked a combination of parameters for RStan that yielded decent results when applied to the stochastic SIR model specified above, calculated the resulting mean MCSE across several model fits, and isolated the expected number of IF2 particles needed to obtain the same value. This was used as a starting value to "titrate" the IF2 iterations to the same point.

The resulting values were 1000 HMCMC warm-up iterations with 1000 samples drawn post-warm-up, and 2500 IF2 particles sent through 50 passes, each method giving an approximate MCSE of 0.0065.

4.3 IF2 Fitting

Now we will use an implementation of the IF2 algorithm to attempt to fit the stochastic SIR model to the previous data. The goal here is just parameter inference, but since IF2 works by applying a series on particle filters we essentially get the average system state estimates for a very small additional computational cost. Hence, we will will also look at that estimated behaviour in addition the the parameter estimates.

The code used here is a mix of R and C++ implemented using RCpp. The fitting was undertaken using 2500 particles with 50 IF2 passes and a cooling schedule given by a reduction in particle spread determined by 0.975^p, where p is the pass number starting with 0.

The MLE parameter estimates, taken to be the mean of the particle swarm values after the final pass, are shown in the table in Figure [4.4], along with the true values and the relative error.

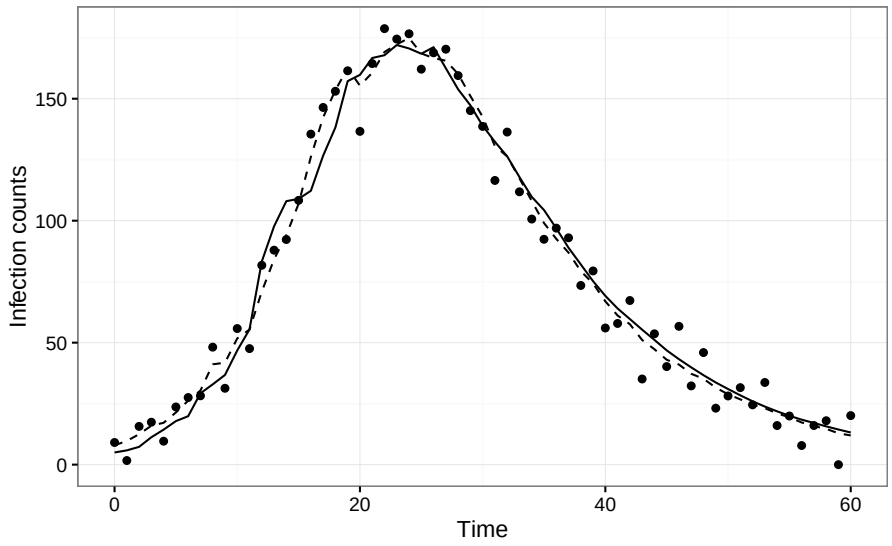

Figure 4.5: True system trajectory (solid line), observed data (dots), and IF2 estimated real state (dashed line).

		IF2		HMCMC	
Name	True	Fit	Error	Fit	Error
R_0	3.0	3.27	9.08×10^{-2}	3.12	1.05×10^{-1}
r	10^{-1}	1.04×10^{-1}	3.61×10^{-2}	9.99×10^{-2}	-7.56×10^{-4}
Initial Infected	5	7.90	5.80×10^{-1}	6.64	3.28×10^{-1}
σ	10	8.84	-1.15×10^{-1}	8.5	-1.50×10^{-1}
η	5×10^{-1}	5.87×10^{-1}	1.73×10^{-1}	4.57×10^{-1}	-8.27×10^{-2}
ε_{err}	5×10^{-1}	1.63×10^{-1}	-6.73×10^{-1}	1.60×10^{-1}	-6.80×10^{-1}

Figure 4.4: Fitting errors.

From last IF2 particle filtering iteration, the mean state values from the particle swarm at each time step are shown with the true underlying state and data in the plot in Figure [4.5].

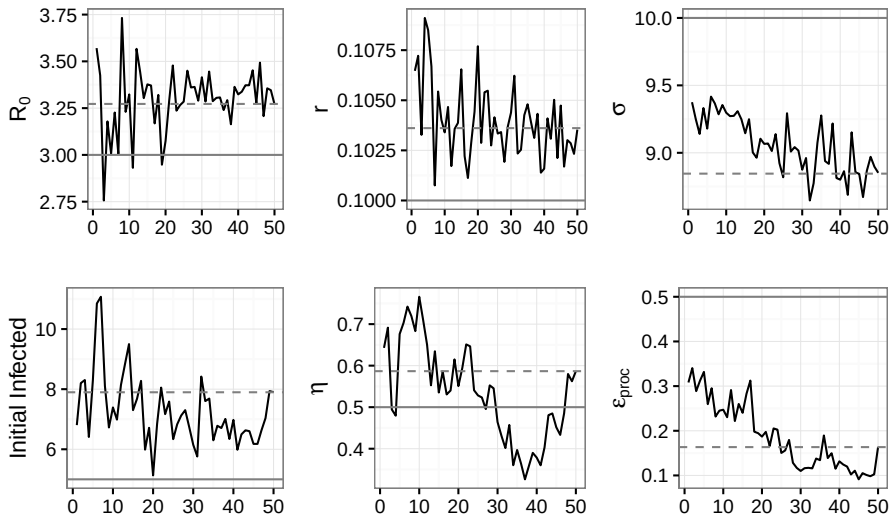

Figure 4.6: The horizontal axis shows the IF2 pass number. The solid black lines show the evolution of the ML estimates, the solid grey lines show the true value, and the dashed grey lines show the mean parameter estimates from the particle swarm after the final pass.

4.4 IF2 Convergence

Since IF2 is an iterative algorithm where each pass through he data is expected to push the parameter estimates towards the MLE, we can see the evolution of these estimates as a function of the pass number. Plots showing evolution of the mean estimates are shown if Figure [4.6] for the six most critical parameters.

Similarly, we can look at the evolution of the standard deviations of the parameter estimates from the particle swarm as a function of the pass number, shown in Figure [4.7] below.

As expected there is a downward trend in all plots, with a very strong trend in all but two of them.

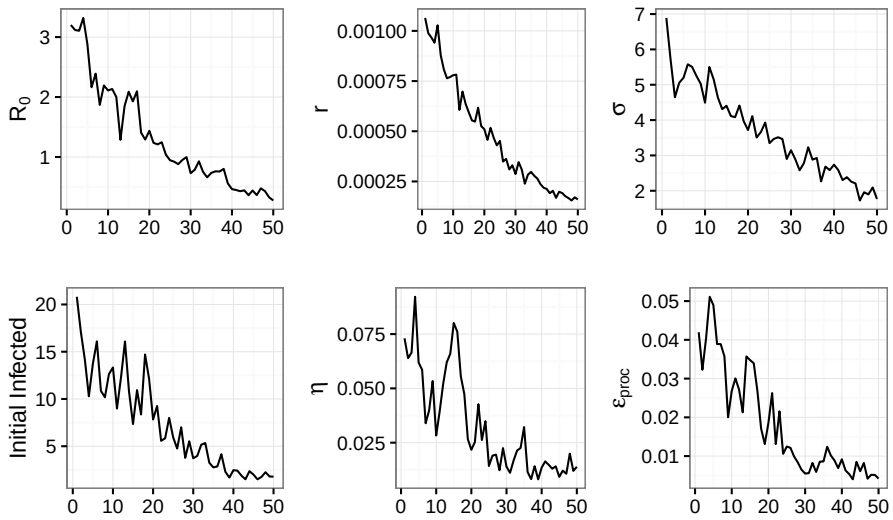

Figure 4.7: The horizontal axis shows the IF2 pass number and the solid black lines show the evolution of the standard deviations of the particle swarm values.

4.5 IF2 Densities

Of diagnostic importance are the densities of the parameter estimates given by the final parameter swarm. These are shown below if Figure [].

It is worth noting that the IF2 parameters chosen were in part chosen so as to not artificially narrow these densities; a more aggressive cooling schedule and/or an increased number of passes would have resulted in much narrower densities, and indeed have the potential to collapse them to point estimates.

4.6 HMCMC Fitting

We can use the Hamiltonian Monte Carlo algorithm implemented in the 'Rstan' package to fit the stochastic SIR model as above. This was done with a single HMC chain of 2000 iterations with 1000 of those being warm-up iterations.

The MLE parameter estimates, taken to be the means of the samples in the chain, were shown in the table in Figure [4.4] along with the true values and relative error.

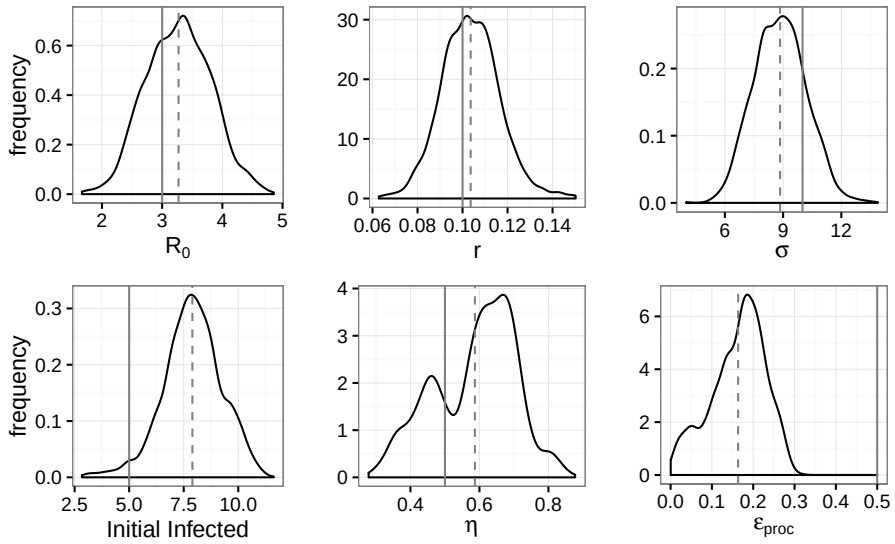

Figure 4.8: As before, the solid grey lines show the true parameter values and the dashed grey lines show the density means.

4.7 HMCMC Densities

The parameter estimation densities from the Stan HMCMC fitting are shown in Figure [4.9] below.

the densities shown here represent a "true" MLE density estimate in that they represent HMC's attempt to directly sample from the parameter space according to the likelihood surface, unlike IF2 which is in theory only trying to get a ML point estimate. Hence, these densities are potentially more robust than those produced by the IF2 implementation.

4.8 HMCMC and Bootstrapping

Unlike particle particle-filtering-based approaches, HMC does not produce state estimates as a by-product of parameter fitting, but we can use information about the stochastic nodes related to the noise in the β geometric random walk to reconstruct state estimates. The results of 100 bootstrap trajectories is shown in Figure [4.10].

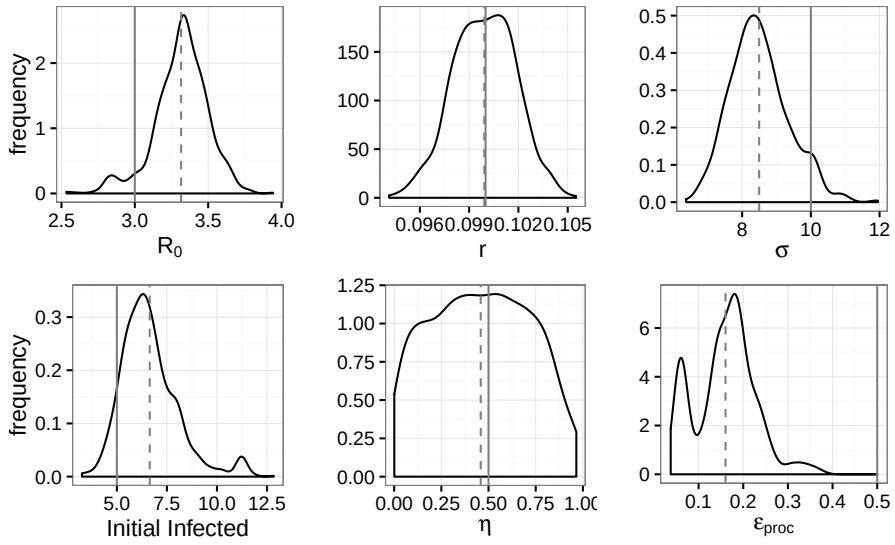

Figure 4.9: As before, the solid grey lines show the true parameter values and the dashed grey lines show the density means.

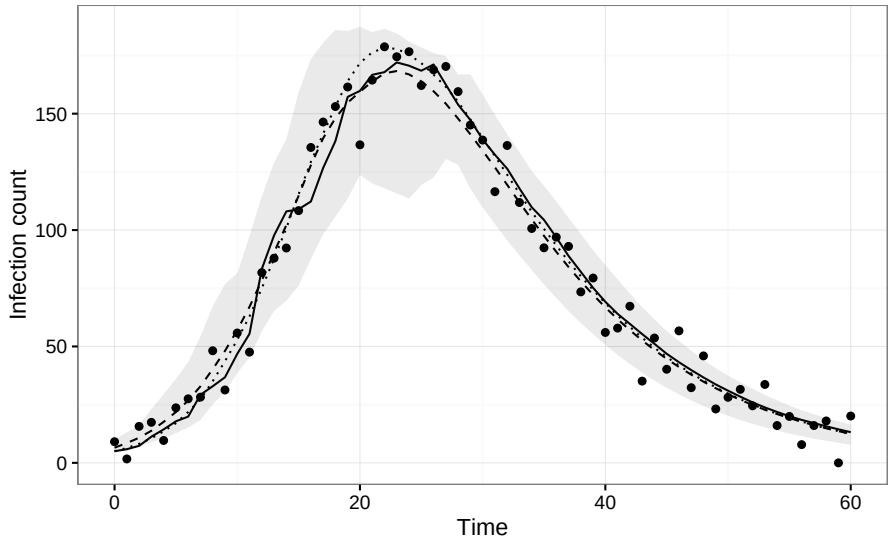

Figure 4.10: Result from 100 HMCMC bootstrap trajectories. The solid line shows the true states, the dots show the data, the dotted line shows the average system behaviour, the dashed line shows the bootstrap mean, and the grey ribbon shows the centre 95th quantile of the bootstrap trajectories.

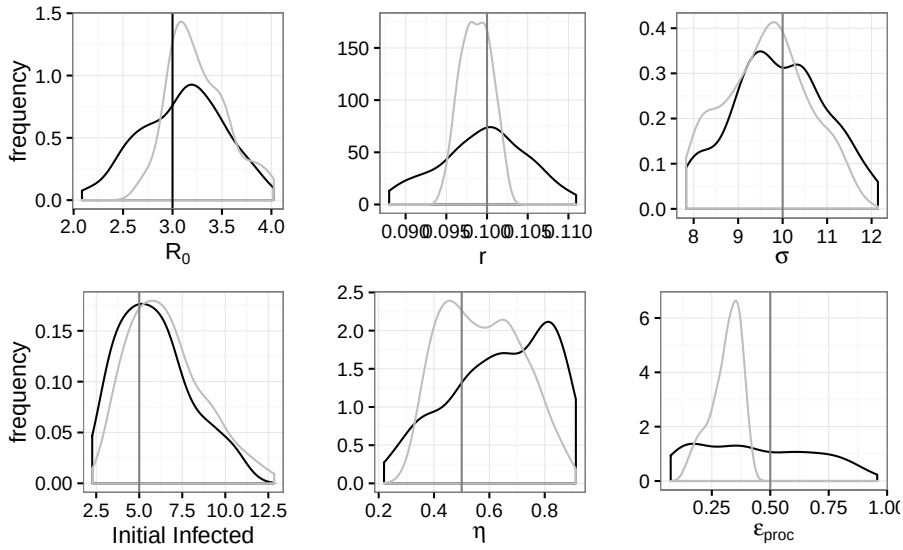

Figure 4.11: IF2 point estimate densities are shown in black and HMCMC point estimate densities are shown in grey. The vertical black lines show the true parameter values.

4.9 Multi-trajectory Parameter Estimation

Here we fit the stochastic SIR model to 200 random independent trajectories using each method and examine the density of the point estimates produced.

The densities by and large display similar coverage, with the IF2 densities for r and ε_{proc} showing slightly wider coverage than the HMCMC densities for the same parameters.

The running times for each algorithm are summarized in Figure [7.4] below.

The average running times were approximately 45.5 seconds and 257.4 seconds for IF2 and HMCMC respectively, representing a 5.7x speedup for IF2 over HMCMC. While IF2 may be able to fit the model to data faster than HMCMC, we are obtaining less information; this will become important in the next section. Further, the results in Figure [7.4] show that while the running time for IF2 is relatively fixed, the times for HMCMC are anything but, showing a wide spread of potential times.

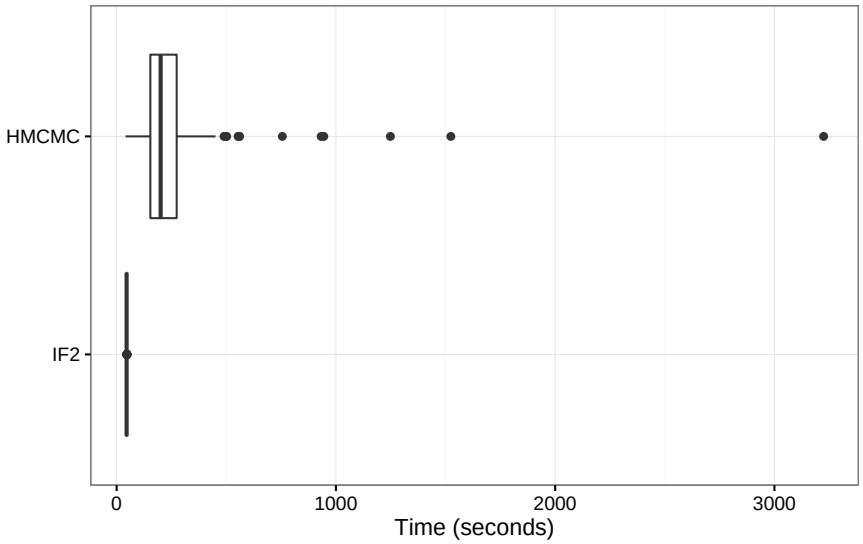

Figure 4.12: Fitting times for IF2 and HMCMC, in seconds. The centre box in each plot shows the centre 50th quantile, with the bold centre line showing the median.

Chapter 5

Forecasting Frameworks

5.1 Data Setup

This section will focus on taking the stochastic SIR model from the previous section, truncating the synthetic data output from realizations of that model, and seeing how well IF2 and HMCMC can reconstruct out-of-sample forecasts.

An example of a simulated system with truncated data can be seen in Figure [7.1].

In essence we want to be able to give either IF2 of HMCMC only the data points and have it reconstruct the entirety of the true system states.

5.2 IF2

For IF2, we will take advantage of the fact that the particle filter will produce state estimates for every datum in the time series given to it, as well as producing parameter maximum likelihood point estimates. Both of these sources of information will be used to produce forecasts by parametric bootstrapping using the final parameter estimates from the particle swarm after the last IF2 pass, then using the newly generated parameter sets along with the system state point estimates from the first fitting to simulate the systems forward into he future.

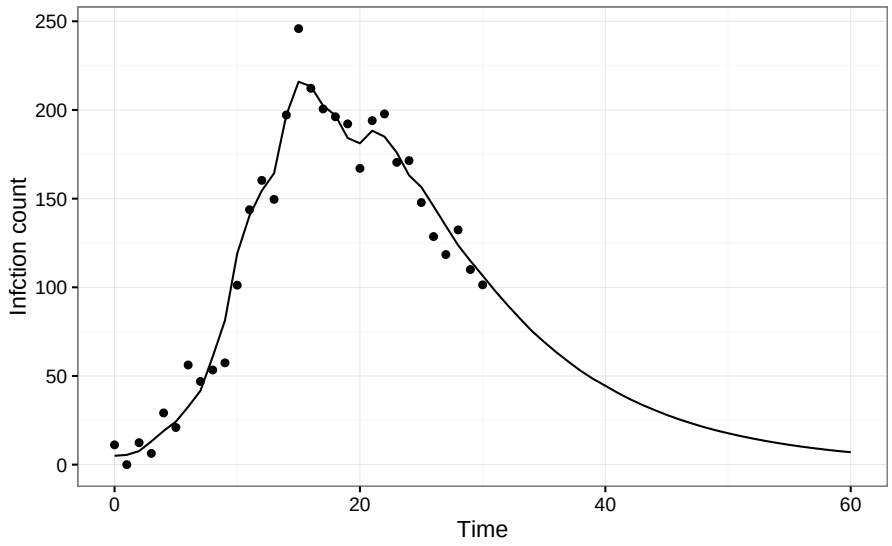

Figure 5.1: Infection count data truncated at $T = 30$. The solid line shows the true underlying system states, and the dots show those states with added observation noise. Parameters used were $R_0 = 3.0$, $r = 0.1$, $\eta = .05$, $\sigma_{proc} = 0.5$, and additive observation noise was drawn from $\mathcal{N}(0, 10)$.

We will truncate the data at half the original time series length (to $T = 30$), and fit the model as previously described.

First, we can see the state estimates for each time point produced by the last IF2 pass in Figure [5.2].

Recall that IF2 is not trying to generate parameter estimation densities, but rather produce a point estimate. Since we wish to determine the approximate distribution of each of the parameters in addition to the point estimate, we must turn to another method, parametric bootstrapping.

5.2.1 Parametric Bootstrapping

The goal of the parametric bootstrap is use an initial density sample θ^* to generate further samples $\theta_1, \theta_2, ..., \theta_M$. It works by using θ to generate artificial data sets $D_1, D_2, ..., D_M$ to which we can refit our model of interest and generate new parameter sets.

[I'm still trying to dig up a good paper that talks about applicability to dynamical

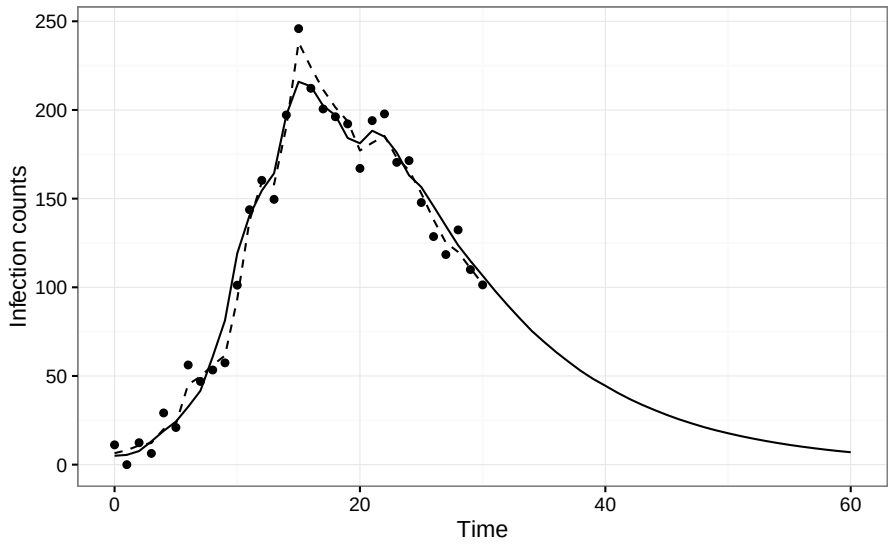

Figure 5.2: Infection count data truncated at $T = 30$ from Figure [7.1] above. The dashed line shows IF2's attempt to reconstruct the true underlying state from the observed data points.

systems, there will be a paragraph here about it.]

An algorithm for parametric bootstrapping using IF2 and our stochastic SIR model is shown in Algorithm [5].

5.2.2 IF2 Forecasts

Using the parameter sets $\theta_1, \theta_2, ..., \theta_M$ and the point estimate of the state provided by the initial IF2 fit, we can use a normal bootstrap to produce estimates of the future state. A plot showing a projection of the data from the previous plots can be seen in Figure [5.3].

We can define a metric to gauge forecast effectiveness by calculating the SSE and dividing that value by the number of values predicted to get the average squared error per point. For the data in Figure [5.3] the value was $\overline{SSE} = 1.67$.

Algorithm 5: Parametric Bootstrap

Input : Forward simulator $S(\theta)$, data set D

/* Initial fit */

$\theta^* \leftarrow IF2(D)$

/* Generate artificial data sets */

for $i = 1 : M$ **do**

  $D_i \leftarrow S(\theta^*)$

/* Fit to new data sets */

for $i = 1 : M$ **do**

  $\theta_i \leftarrow IF2(D_i)$

Output: Distribution samples $\theta_1, \theta_2, ..., \theta_M$

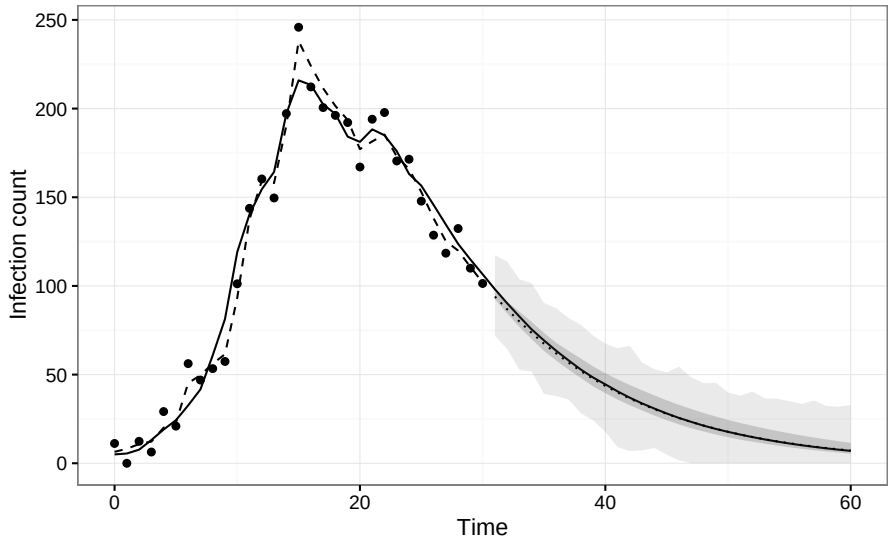

Figure 5.3: Forecast produced by the IF2 / parametric bootstrapping framework. The dotted line shows the mean estimate of the forecasts, the dark grey ribbon shows the centre 95th quantile of the true state estimates, and the lighter grey ribbon shows the centre 95th quantile of the true state estimates with added observation noise drawn from $\mathcal{N}(0, \sigma)$.

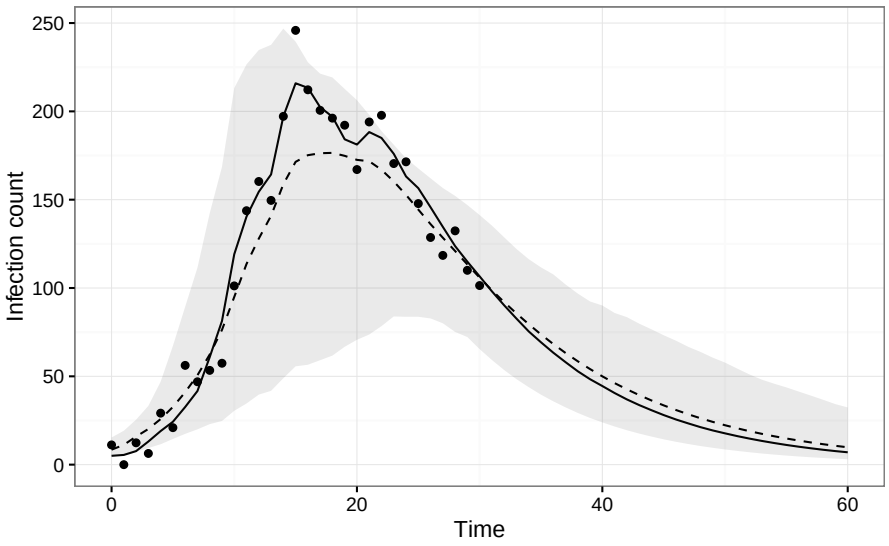

Figure 5.4: Forecast produced by the HMCMC / bootstrapping framework with $M = 200$ trajectories. The dotted line shows the mean estimate of the forecasts, and the grey ribbon shows the centre 95th quantile.

5.3 HMCMC

For HMCMC we can use a simpler bootstrapping approach. We do not get state estimates directly from the RStan fitting due to the way we implemented the model, but we can construct them using the process noise latent variables. Once we've done this we can forward simulate the system from the state estimate into the future.

As before we fit the stochastic SIR model to the partial data, but now perform bootstrapping as described above, and obtain the plot in Figure [5.4].

And as before we can evaluate the averaged SSE of the forecast for the data shown, giving $\overline{SSE} = 20.27$.

5.4 Truncation vs. Error

Of course the above mini-comparison only shows one truncation value for one trajectory. Really, we need to know how each method performs on average given different trajectories and truncation amounts. In effect we wish to "starve" each method

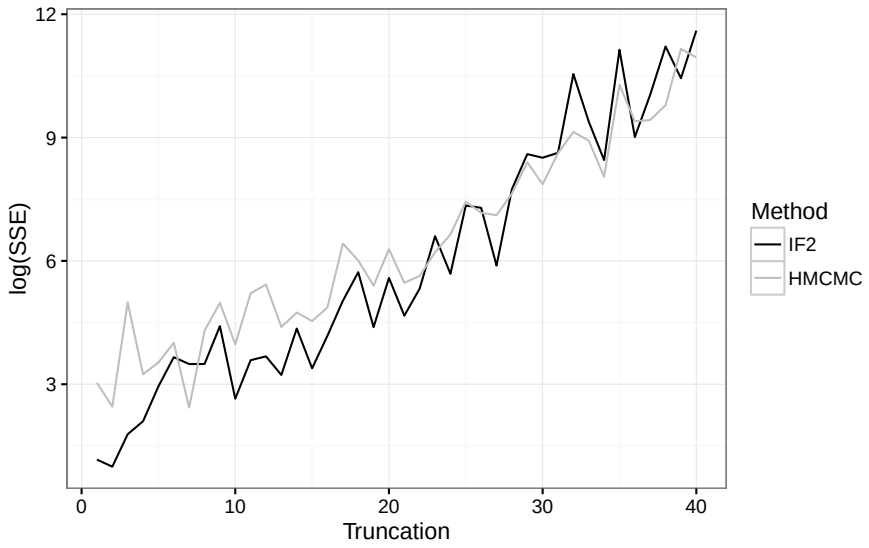

Figure 5.5: Error growth as a function of data truncation amount. Both methods used 200 bootstrap trajectories. Note that the y-axis shows the natural log of the averaged SSE, not the total SSE.

of data and see how poor the estimates become with each successive data point loss.

Using each method, we can fit the stochastic SIR model to successively smaller time series to see the effect of truncation on forecast averaged SSE. This was performed with 10 new trajectories drawn for each of the desired lengths. The results are shown in Figure [5.5].

IF2 and HMCMC perform very closely, with IF2 maintaining a small advantage up to a truncation of about 25-30 data points.

Since the parametric bootstrapping approach used by IF2 requires a significant number of additional fits, its computational cost is significantly higher than the simpler bootstrapping approach used by the HMCMC framework, about 35.5x as expensive. However the now much longer running time can somewhat alleviated by parallelizing the parametric bootstrapping process; as each of the parametric bootstrap fittings in entirely independent, this can be done without a great deal of additional effort. The code used here has this capability, but it was not utilised in the comparison so as to accurately represent total computational cost, not potential running time.

Chapter 6

S-map and SIRS

6.1 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

$$\begin{aligned} A(i,j) &= w(||\mathbf{x_i} - \mathbf{x_t}||)\mathbf{x_i}(j) \\ b(i) &= w(||\mathbf{x_i} - \mathbf{x_t}||)y_i \end{aligned} \tag{6.1}$$

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

$$w(d) = \exp\left(\frac{-\theta d}{\bar{d}}\right), \tag{6.2}$$

where d is the euclidean distance between the predictor vector and library vectors in Equation (6.1) 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

$$\hat{y_t} = \sum_{j=0}^{E} c_t(j)\mathbf{x_t}(j). \tag{6.3}$$

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.

6.2 S-map Algorithm

The above description can be summarized in Algorithm [6].

Algorithm 6: 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 $\hat{y}_t$

6.3 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.

$$\begin{aligned} \frac{dS}{dt} &= -\Gamma(t)\beta SI + \eta R \\ \frac{dI}{dt} &= \Gamma(t)\beta SI - \gamma I \\ \frac{dR}{dt} &= \gamma I - \eta R, \end{aligned} \tag{6.4}$$

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

$$\Gamma(t) = \exp\left(2cos\left(\frac{2\pi}{365}t\right) - 2\right). \tag{6.5}$$

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

$$\beta_{t+1} = \exp\left(\log(\beta_t) + \eta(\log(\bar{\beta}) - \log(\beta_t)) + \epsilon_t\right). \tag{6.6}$$

When simulated for the equivalent of 5 years (260 weeks), and adding noise drawn from $\mathcal{N}(0, \sigma)$we obtain Figure [7.1].

We can see how the S-map can reconstruct the next cycle in the time series in Figure [6.2].

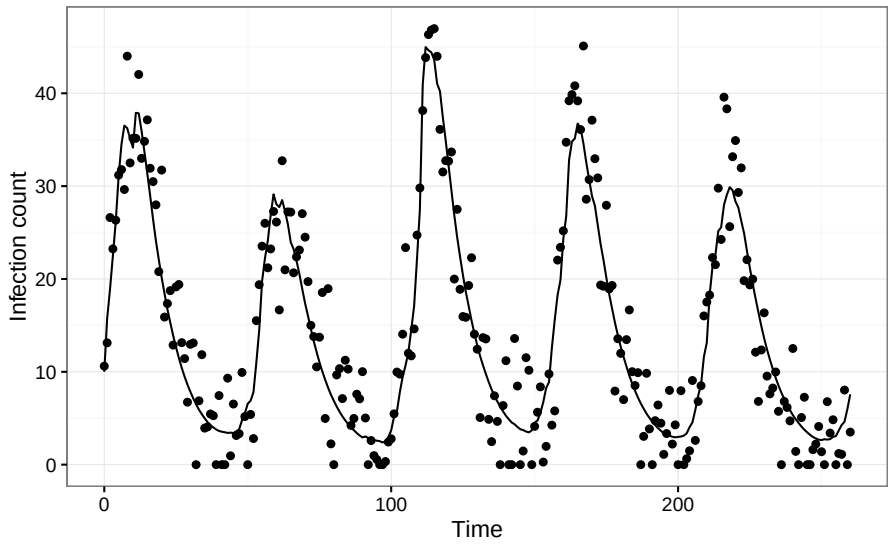

Figure 6.1: Five cycles generated by the SIRS function. The solid line the the true number of cases, dots show case counts with added observation noise. The Parameter values were $R0 = 3.0$, $\gamma = 0.1$, $\eta = 1$, $\sigma = 5$, and 10 initial cases.

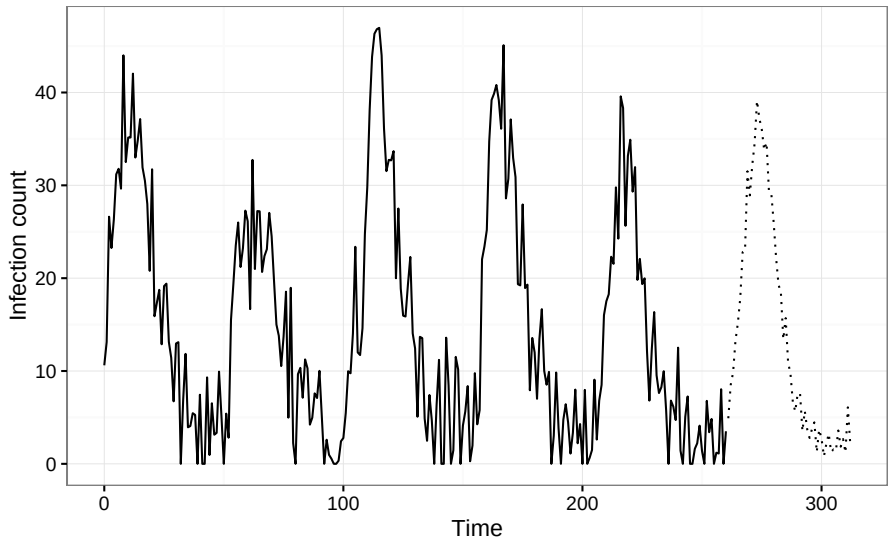

Figure 6.2: S-map applied to the data from the previous figure. The solid line shows the infection counts with observation noise form the previous plot, and the dotted line is the S-map forecast. Parameters chosen were $E = 14$ and $\theta = 3$.

The parameters used in the S-map algorithm to obtain the forecast used in Figure

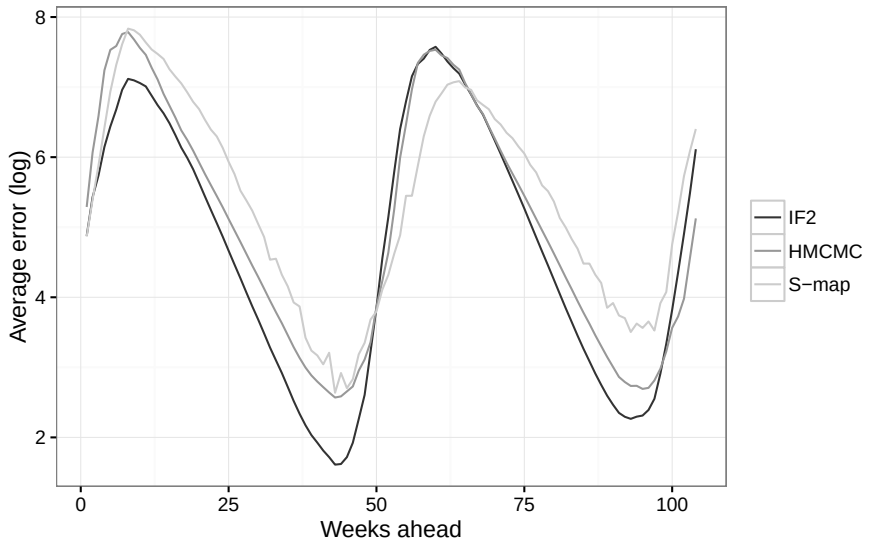

Figure 6.3: Error as a function of forecast length.

[6.2] were obtained using a grid search of potential parameters outlined in (Sugihara ref). The script is included in the appendices.

6.4 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 [7.3].

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.

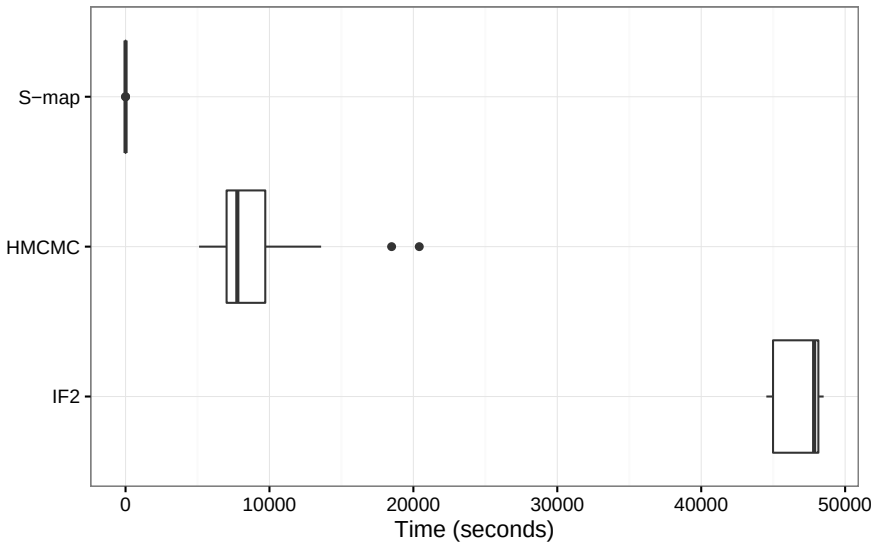

Figure 6.4: Runtimes for producing SIRS forecasts. The box shows the middle 50th quantile, the bold line is the median, and the dots are outliers.

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 [7.4] shows the running times over 20 simulations.

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.

Chapter 7

Spatial Epidemics

7.1 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

$$\begin{aligned}
\frac{dS_i}{dt} &= -\left(1 - \phi\frac{N_i}{N_i+1}\right)\beta_i S_i I_i - \left(\frac{\phi}{N_i+1}\right) S_i \sum_{j=0}^{N_i} \beta_j I_j \\
\frac{dI_i}{dt} &= \left(1 - \phi\frac{N_i}{N_i+1}\right)\beta_i S_i I_i + \left(\frac{\phi}{N_i+1}\right) S_i \sum_{j=0}^{N_i} \beta_j I_j - \gamma I \\
\frac{dR_i}{dt} &= \gamma I,
\end{aligned} \tag{7.1}$$

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

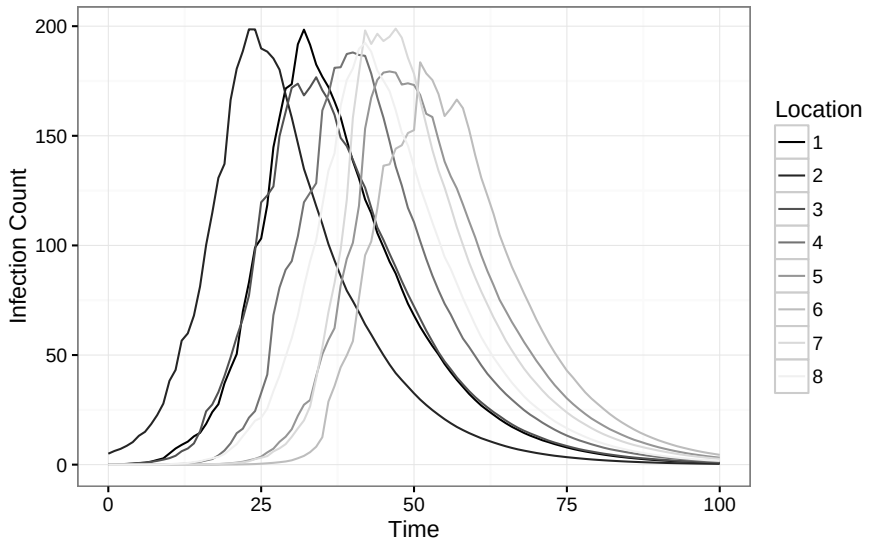

Figure 7.1: Evolution of a spatial epidemic in a ring topology. The outbreak was started with 5 cases in Location 2. Parameters were $R_0 = 3.0$, $\gamma = 0.1$, $\eta = 0.5$, $\sigma_{err} = 0.5$, and $\phi = 0.5$.

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

$$\beta_{i,t+1} = \exp\left(\log(\beta_{i,t}) + \eta(\log(\bar{\beta}) - \log(\beta_{i,t})) + \epsilon_t\right). \tag{7.2}$$

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 [7.1].

If we add noise to the data from Figure [7.1], we obtain Figure [7.2], below.

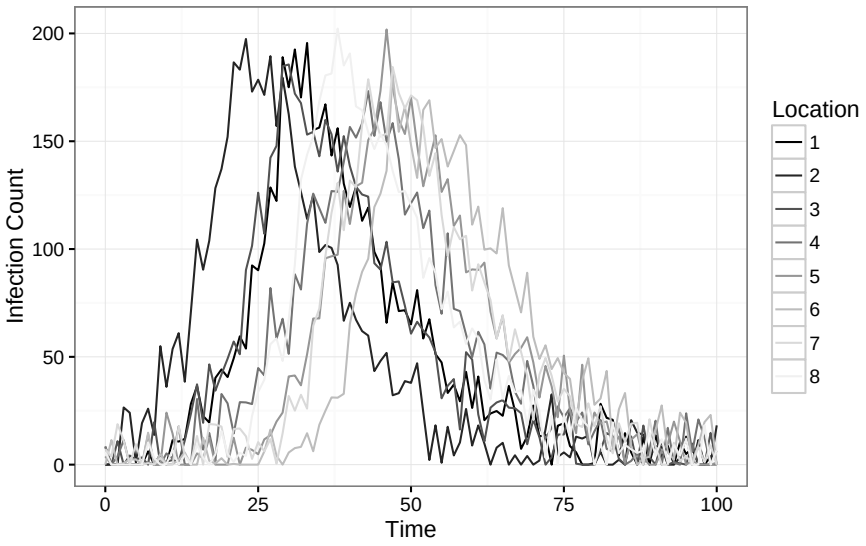

Figure 7.2: Evolution of a spatial epidemic as in Figure [7.1], with added observation noise drawn from $\mathcal{N}(0, 10)$.

7.2 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

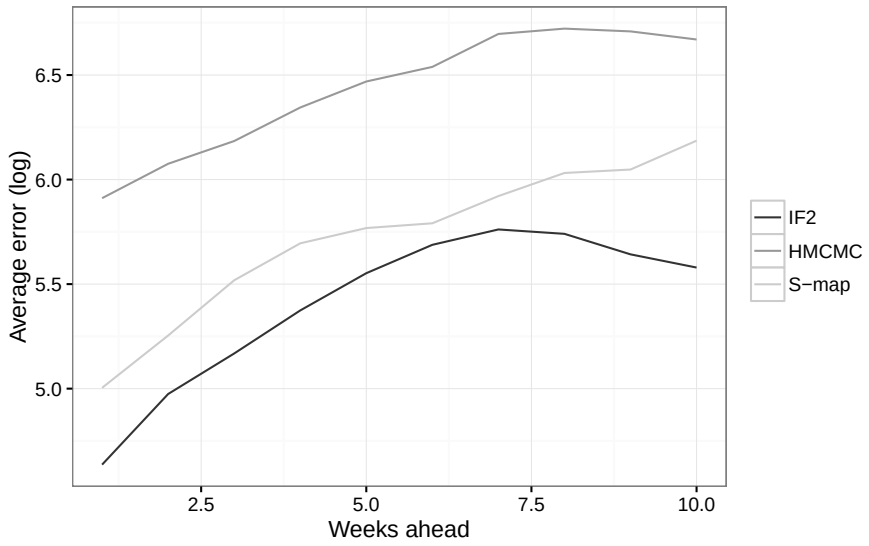

Figure 7.3: Average SSE (log scale) across each location and all trials as a function of the number of weeks ahead in the forecast.

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.

7.3 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 [7.3].

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

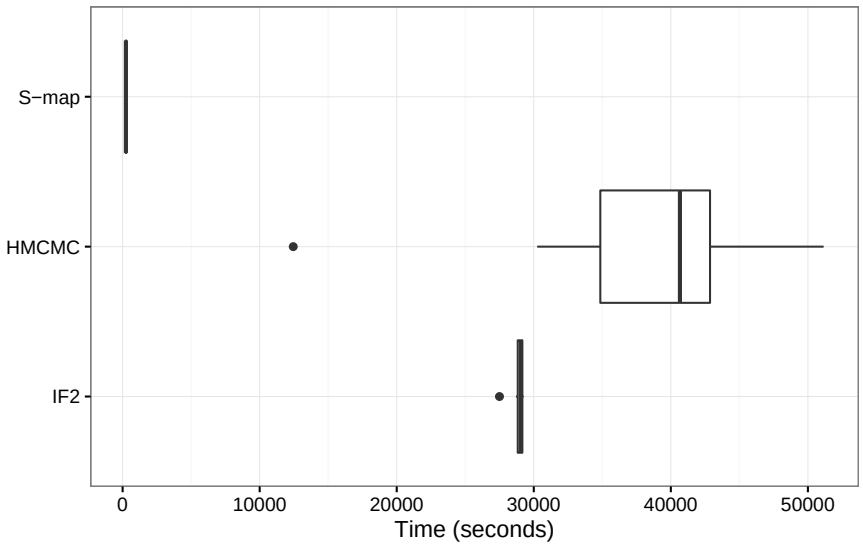

Figure 7.4: Runtimes for producing spatial SIR forecasts. The box shows the middle 50th quantile, the bold line is the median, and the dots are outliers.

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 [7.4].

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 HMCMC 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 HMCMC.

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

Appendix A

Hamiltonian MCMC

A.1 Full R code

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

```
## Dexter Barrows
## McMaster University
## 2016

library(deSolve)
library(rstan)
library(shinystan)
library(ggplot2)
library(RColorBrewer)
library(reshape2)

SIR ← function(Time, State, Pars) {

    with(as.list(c(State, Pars)), {

        B   ← R0*r/N
        BSI ← B*S*I
        rI  ← r*I

        dS = -BSI
        dI = BSI - rI
        dR = rI

        return(list(c(dS, dI, dR)))

    })
```

```
27
28 }
29
30 pars  <- c(R0  <- 3.0,    # average number of new infected individuals
       per infectious person
31            r   <- 0.1,    # recovery rate
32            N   <- 500)    # population size
33
34 T <- 100
35 y_ini <- c(S = 495, I = 5, R = 0)
36 times <- seq(0, T, by = 1)
37
38 odeout <- ode(y_ini, times, SIR, pars)
39
40 set.seed(1001)
41 sigma <- 10
42 infec_counts_raw <- odeout[,3] + rnorm(T+1, 0, sigma)
43 infec_counts     <- ifelse(infec_counts_raw < 0, 0, infec_counts_raw)
44
45 g <- qplot(0:T, odeout[,3], geom = "line", xlab = "Time (weeks)",
      ylab = "Infection Count") +
46     geom_point(aes(y = infec_counts)) +
47     theme_bw()
48
49 print(g)
50 ggsave(g, filename="dataplot.pdf", height=4, width=6.5)
51
52 sPw <- 7
53 datlen <- (T-1)*7 + 1
54
55 data <- matrix(data = -1, nrow = T+1, ncol = sPw)
56 data[,1] <- infec_counts
57 standata <- as.vector(t(data))[1:datlen]
58
59 sir_data <- list( T = datlen,   # simulation time
60                   y = standata, # infection count data
61                   N = 500,      # population size
62                   h = 1/sPw )   # step size per day
63
64 rstan_options(auto_write = TRUE)
65 options(mc.cores = parallel::detectCores())
66 stan_options <- list(   chains = 4,    # number of chains
67                         iter   = 2000, # iterations per chain
68                         warmup = 1000, # warmup interations
69                         thin   = 2)   # thinning number
70 fit <- stan(file    = "d_sirode_euler.stan",
71             data    = sir_data,
72             chains  = stan_options$chains,
73             iter    = stan_options$iter,
74             warmup  = stan_options$warmup,
```

```
75              thin    = stan_options$thin )
76
77 exfit <- extract(fit, permuted = TRUE, inc_warmup = FALSE)
78
79 R0points <- exfit$R0
80 R0kernel <- qplot(R0points, geom = "density", xlab = expression(R[0])
       , ylab = "frequency") +
81          geom_vline(aes(xintercept=R0), linetype="dashed", size=1,
               color="grey50") +
82          theme_bw()
83
84 print(R0kernel)
85 ggsave(R0kernel, filename="kernelR0.pdf", height=3, width=3.25)
86
87 rpoints <- exfit$r
88 rkernel <- qplot(rpoints, geom = "density", xlab = "r", ylab = "
       frequency") +
89          geom_vline(aes(xintercept=r), linetype="dashed", size=1,
               color="grey50") +
90          theme_bw()
91
92 print(rkernel)
93 ggsave(rkernel, filename="kernelr.pdf", height=3, width=3.25)
94
95 sigmapoints <- exfit$sigma
96 sigmakernel <- qplot(sigmapoints, geom = "density", xlab = expression
       (sigma), ylab = "frequency") +
97          geom_vline(aes(xintercept=sigma), linetype="dashed", size=1,
                color="grey50") +
98          theme_bw()
99
100 print(sigmakernel)
101 ggsave(sigmakernel, filename="kernelsigma.pdf", height=3, width
       =3.25)
102
103 infecpoints <- exfit$y0[,2]
104 infeckernel <- qplot(infecpoints, geom = "density", xlab = "Initial
       Infected", ylab = "frequency") +
105          geom_vline(aes(xintercept=y_ini[['I']]), linetype="dashed",
               size=1, color="grey50") +
106          theme_bw()
107
108 print(infeckernel)
109 ggsave(infeckernel, filename="kernelinfec.pdf", height=3, width
       =3.25)
110
111 exfit <- extract(fit, permuted = FALSE, inc_warmup = FALSE)
112 plotdata <- melt(exfit[,,"R0"])
113 tracefitR0 <- ggplot() +
114               geom_line(data = plotdata,
```

```
115                         aes(x = iterations,
116                         y = value,
117                         color = factor(chains, labels = 1:stan_
                               options$chains))) +
118               labs(x = "Sample", y = expression(R[0]), color = "
                     Chain") +
119               scale_color_brewer(palette="Greys") +
120               theme_bw()
121
122 print(tracefitR0)
123 ggsave(tracefitR0, filename="traceplotR0.pdf", height=4, width=6.5)
124
125 exfit ← extract(fit, permuted = FALSE, inc_warmup = TRUE)
126 plotdata ← melt(exfit[,,"R0"])
127 tracefitR0 ← ggplot() +
128               geom_line(data = plotdata,
129                         aes(x = iterations,
130                         y = value,
131                         color = factor(chains, labels = 1:stan_
                               options$chains))) +
132               labs(x = "Sample", y = expression(R[0]), color = "
                     Chain") +
133               scale_color_brewer(palette="Greys") +
134               theme_bw()
135
136 print(tracefitR0)
137 ggsave(tracefitR0, filename="traceplotR0_inc.pdf", height=4, width
      =6.5)
138
139 sso ← as.shinystan(fit)
140 sso ← launch_shinystan(sso)
```

A.2 Full Stan code

Stan model code to be used with the preceding R code.

```
1  ## Dexter Barrows
2  ## McMaster University
3  ## 2016
4
5  data {
6
7      int     <lower=1>   T;      // total integration steps
8      real                y[T];   // observed number of cases
9      int     <lower=1>   N;      // population size
10     real                h;      // step size
```

```
11
12 }
13
14 parameters {
15
16     real <lower=0, upper=10>    R0;     // R0
17     real <lower=0, upper=10>    r;      // recovery rate
18     real <lower=0, upper=20>    sigma;  // observation error
19     real <lower=0, upper=500>   y0[3];  // initial conditions
20
21 }
22
23 model {
24
25     real S[T];
26     real I[T];
27     real R[T];
28
29     S[1] <- y0[1];
30     I[1] <- y0[2];
31     R[1] <- y0[3];
32
33     y[1] ~ normal(y0[2], sigma);
34
35     for (t in 2:T) {
36
37         S[t] <- S[t-1] + h*( - S[t-1]*I[t-1]*R0*r/N );
38         I[t] <- I[t-1] + h*( S[t-1]*I[t-1]*R0*r/N  - I[t-1]*r );
39         R[t] <- R[t-1] + h*( I[t-1]*r );
40
41         if (y[t] > 0) {
42             y[t] ~ normal( I[t], sigma );
43         }
44
45     }
46
47     y0[1] ~ normal(N - y[1], sigma);
48     y0[2] ~ normal(y[1], sigma);
49
50     R0      ~ lognormal(1,1);
51     r       ~ lognormal(1,1);
52     sigma   ~ lognormal(1,1);
53
54 }
```

Appendix B

Iterated Filtering

B.1 Full R code

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

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

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

SIR ← function(Time, State, Pars) {

    with(as.list(c(State, Pars)), {

        B   ← R0*r/N
        BSI ← B*S*I
        rI  ← r*I

        dS = -BSI
        dI = BSI - rI
        dR = rI

        return(list(c(dS, dI, dR)))

    })

}
```

```
27
28 T     ← 100
29 N     ← 500
30 sigma   ← 10
31 i_infec ← 5
32
33 ## Generate true trajecory and synthetic data
34 ##
35
36 true_init_cond ← c(S = N - i_infec,
37                     I = i_infec,
38                     R = 0)
39
40 true_pars ← c(R0 = 3.0,
41                r = 0.1,
42                N = 500.0)
43
44 odeout ← ode(true_init_cond, 0:T, SIR, true_pars)
45 trueTraj ← odeout[,3]
46
47 set.seed(1001)
48
49 infec_counts_raw ← odeout[,3] + rnorm(T+1, 0, sigma)
50 infec_counts     ← ifelse(infec_counts_raw < 0, 0, infec_counts_raw)
51
52 g ← qplot(0:T, odeout[,3], geom = "line", xlab = "Time (weeks)",
       ylab = "Infection Count") +
53     geom_point(aes(y = infec_counts)) +
54     theme_bw()
55
56 print(g)
57 ggsave(g, filename="dataplot.pdf", height=4, width=6.5)
58
59 ## Rcpp stuff
60 ##
61
62 sourceCpp(paste(getwd(),"d_if2.cpp",sep="/"))
63
64 paramdata ← data.frame(if2(infec_counts, T+1, N))
65 colnames(paramdata) ← c("R0", "r", "sigma", "Sinit", "Iinit", "Rinit
       ")
66
67 ## Parameter density kernels
68 ##
69
70 R0points ← paramdata$R0
71 R0kernel ← qplot(R0points, geom = "density", xlab = expression(R[0])
       , ylab = "frequency") +
72         geom_vline(aes(xintercept=true_pars[["R0"]]), linetype="
              dashed", size=1, color="grey50") +
```

```
        theme_bw()

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

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

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

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

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

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

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

# show grid
grid.arrange(R0kernel, rkernel, sigmakernel, infeckernel, ncol = 2, nrow = 2)

pdf("if2kernels.pdf", height = 6.5, width = 6.5)
grid.arrange(R0kernel, rkernel, sigmakernel, infeckernel, ncol = 2, nrow = 2)
dev.off()
#ggsave(filename="if2kernels.pdf", g2,  height=6.5, width=6.5)
```

B.2 Full C++ code

Stan model code to be used with the preceding R code.

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

    */

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

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

#define Treal   100         // time to simulate over
#define R0true  3.0         // infectiousness
#define rtrue   0.1         // recovery rate
#define Nreal   500.0       // population size
#define merr    10.0        // expected measurement error
#define I0      5.0         // Initial infected individuals

#include <Rcpp.h>
using namespace Rcpp;


struct Particle {
    double R0;
    double r;
    double sigma;
    double S;
    double I;
    double R;
    double Sinit;
    double Iinit;
    double Rinit;
};

struct ParticleInfo {
    double R0mean;      double R0sd;
    double rmean;       double rsd;
    double sigmamean;   double sigmasd;
```

```
46      double Sinitmean;   double Sinitsd;
47      double Iinitmean;   double Iinitsd;
48      double Rinitmean;   double Rinitsd;
49  };
50
51
52  int timeval_subtract (double *result, struct timeval *x, struct
        timeval *y);
53  int check_double(double x,double y);
54  void exp_euler_SIR(double h, double t0, double tn, int N, Particle *
        particle);
55  void copyParticle(Particle * dst, Particle * src);
56  void perturbParticles(Particle * particles, int N, int NP, int
        passnum, double coolrate);
57  bool isCollapsed(Particle * particles, int NP);
58  void particleDiagnostics(ParticleInfo * partInfo, Particle *
        particles, int NP);
59  NumericMatrix if2(NumericVector * data, int T, int N);
60  double randu();
61  double randn();
62
63  // [[Rcpp::export]]
64  NumericMatrix if2(NumericVector data, int T, int N) {
65
66      int     NP          = 2500;
67      int     nPasses     = 50;
68      double  coolrate    = 0.975;
69
70      int     i_infec     = I0;
71
72      NumericMatrix paramdata(NP, 6);
73
74      srand(time(NULL));      // Seed PRNG with system time
75
76      double w[NP];           // particle weights
77
78      Particle particles[NP];     // particle estimates for current
            step
79      Particle particles_old[NP]; // intermediate particle states for
            resampling
80
81      printf("Initializing particle states\n");
82
83      // initialize particle parameter states (seeding)
84      for (int n = 0; n < NP; n++) {
85
86          double R0can, rcan, sigmacan, Iinitcan;
87
88          do {
89              R0can = R0true + R0true*randn();
```

```
90          } while (R0can < 0);
91          particles[n].R0 = R0can;
92
93          do {
94              rcan = rtrue + rtrue*randn();
95          } while (rcan < 0);
96          particles[n].r = rcan;
97
98          do {
99              sigmacan = merr + merr*randn();
100         } while (sigmacan < 0);
101         particles[n].sigma = sigmacan;
102
103         do {
104             Iinitcan = i_infec + i_infec*randn();
105         } while (Iinitcan < 0 || N < Iinitcan);
106         particles[n].Sinit = N - Iinitcan;
107         particles[n].Iinit = Iinitcan;
108         particles[n].Rinit = 0.0;
109
110     }
111
112     // START PASSES THROUGH DATA
113
114     printf("Starting filter\n");
115     printf("---------------\n");
116     printf("Pass\n");
117
118
119     for (int pass = 0; pass < nPasses; pass++) {
120
121         printf("...%d / %d\n", pass, nPasses);
122
123         perturbParticles(particles, N, NP, pass, coolrate);
124
125         // initialize particle system states
126         for (int n = 0; n < NP; n++) {
127
128             particles[n].S = particles[n].Sinit;
129             particles[n].I = particles[n].Iinit;
130             particles[n].R = particles[n].Rinit;
131
132         }
133
134         // between-pass perturbations
135
136         for (int t = 1; t < T; t++) {
137
138             // between-iteration perturbations
139             perturbParticles(particles, N, NP, pass, coolrate);
```

```
140
141             // generate individual predictions and weight
142             for (int n = 0; n < NP; n++) {
143
144                 exp_euler_SIR(1.0/10.0, 0.0, 1.0, N, &particles[n]);
145
146                 double merr_par = particles[n].sigma;
147                 double y_diff   = data[t] - particles[n].I;
148
149                 w[n] = 1.0/(merr_par*sqrt(2.0*M_PI)) * exp( - y_diff
                        *y_diff / (2.0*merr_par*merr_par) );
150
151             }
152
153             // cumulative sum
154             for (int n = 1; n < NP; n++) {
155                 w[n] += w[n-1];
156             }
157
158             // save particle states to resample from
159             for (int n = 0; n < NP; n++){
160                 copyParticle(&particles_old[n], &particles[n]);
161             }
162
163             // resampling
164             for (int n = 0; n < NP; n++) {
165
166                 double w_r = randu() * w[NP-1];
167                 int i = 0;
168                 while (w_r > w[i]) {
169                     i++;
170                 }
171
172                 // i is now the index to copy state from
173                 copyParticle(&particles[n], &particles_old[i]);
174
175             }
176
177         }
178
179     }
180
181     ParticleInfo pInfo;
182     particleDiagnostics(&pInfo, particles, NP);
183
184     printf("Parameter results (mean | sd)\n");
185     printf("-----------------------------\n");
186     printf("R0        %f %f\n", pInfo.R0mean, pInfo.R0sd);
187     printf("r         %f %f\n", pInfo.rmean, pInfo.rsd);
188     printf("sigma     %f %f\n", pInfo.sigmamean, pInfo.sigmasd);
```

```
189     printf("S_init  %f %f\n", pInfo.Sinitmean, pInfo.Sinitsd);
190     printf("I_init    %f %f\n", pInfo.Iinitmean, pInfo.Iinitsd);
191     printf("R_init    %f %f\n", pInfo.Rinitmean, pInfo.Rinitsd);
192 
193     printf("\n");
194 
195 
196 
197     // Get particle results to pass back to R
198 
199     for (int n = 0; n < NP; n++) {
200 
201         paramdata(n, 0) = particles[n].R0;
202         paramdata(n, 1) = particles[n].r;
203         paramdata(n, 2) = particles[n].sigma;
204         paramdata(n, 3) = particles[n].Sinit;
205         paramdata(n, 4) = particles[n].Iinit;
206         paramdata(n, 5) = particles[n].Rinit;
207 
208     }
209 
210     return paramdata;
211 
212 }
213 
214 
215 /*  Use the Explicit Euler integration scheme to integrate SIR model
        forward in time
216     double h    - time step size
217     double t0   - start time
218     double tn   - stop time
219     double * y  - current system state; a three-component vector
            representing [S I R], susceptible-infected-recovered
220 
221     */
222 void exp_euler_SIR(double h, double t0, double tn, int N, Particle *
        particle) {
223 
224     int num_steps = floor( (tn-t0) / h );
225 
226     double S = particle->S;
227     double I = particle->I;
228     double R = particle->R;
229 
230     double R0   = particle->R0;
231     double r    = particle->r;
232     double B    = R0 * r / N;
233 
234     for(int i = 0; i < num_steps; i++) {
235         // get derivatives
```

```
236         double dS = - B*S*I;
237         double dI = B*S*I - r*I;
238         double dR = r*I;
239         // step forward by h
240         S += h*dS;
241         I += h*dI;
242         R += h*dR;
243     }
244 
245     particle->S = S;
246     particle->I = I;
247     particle->R = R;
248 
249 }
250 
251 
252 /*  Particle pertubation function to be run between iterations and
    passes
253 
254     */
255 void perturbParticles(Particle * particles, int N, int NP, int
    passnum, double coolrate) {
256 
257     double coolcoef = pow(coolrate, passnum);
258 
259     double spreadR0     = coolcoef * R0true / 10.0;
260     double spreadr      = coolcoef * rtrue  / 10.0;
261     double spreadsigma  = coolcoef * merr   / 10.0;
262     double spreadIinit  = coolcoef * I0     / 10.0;
263 
264     double R0can, rcan, sigmacan, Iinitcan;
265 
266     for (int n = 0; n < NP; n++) {
267 
268         do {
269             R0can = particles[n].R0 + spreadR0*randn();
270         } while (R0can < 0);
271         particles[n].R0 = R0can;
272 
273         do {
274             rcan = particles[n].r + spreadr*randn();
275         } while (rcan < 0);
276         particles[n].r = rcan;
277 
278         do {
279             sigmacan = particles[n].sigma + spreadsigma*randn();
280         } while (sigmacan < 0);
281         particles[n].sigma = sigmacan;
282 
283         do {
```

```
284             Iinitcan = particles[n].Iinit + spreadIinit*randn();
285         } while (Iinitcan < 0 || Iinitcan > 500);
286         particles[n].Iinit = Iinitcan;
287         particles[n].Sinit = N - Iinitcan;
288 
289     }
290 
291 }
292 
293 
294 /*  Convinience function for particle resampling process
295 
296     */
297 void copyParticle(Particle * dst, Particle * src) {
298 
299     dst->R0     = src->R0;
300     dst->r      = src->r;
301     dst->sigma  = src->sigma;
302     dst->S      = src->S;
303     dst->I      = src->I;
304     dst->R      = src->R;
305     dst->Sinit  = src->Sinit;
306     dst->Iinit  = src->Iinit;
307     dst->Rinit  = src->Rinit;
308 
309 }
310 
311 
312 /*  Checks to see if particles are collapsed
313     This is done by checking if the standard deviations between the
        particles' parameter
314     values are significantly close to one another. Spread threshold
        may need to be tuned.
315 
316     */
317 bool isCollapsed(Particle * particles, int NP) {
318 
319     bool retVal;
320 
321     double R0mean = 0, rmean = 0, sigmamean = 0, Sinitmean = 0,
        Iinitmean = 0, Rinitmean = 0;
322 
323     // means
324 
325     for (int n = 0; n < NP; n++) {
326 
327         R0mean      += particles[n].R0;
328         rmean       += particles[n].r;
329         sigmamean   += particles[n].sigma;
330         Sinitmean   += particles[n].Sinit;
```

```
331         Iinitmean   += particles[n].Iinit;
332         Rinitmean   += particles[n].Rinit;
333 
334     }
335 
336     R0mean      /= NP;
337     rmean       /= NP;
338     sigmamean   /= NP;
339     Sinitmean   /= NP;
340     Iinitmean   /= NP;
341     Rinitmean   /= NP;
342 
343     double  R0sd = 0, rsd = 0, sigmasd = 0, Sinitsd = 0, Iinitsd =
            0, Rinitsd = 0;
344 
345     for (int n = 0; n < NP; n++) {
346 
347         R0sd    += ( particles[n].R0 - R0mean ) * ( particles[n].R0
                - R0mean );
348         rsd     += ( particles[n].r - rmean ) * ( particles[n].r -
                rmean );
349         sigmasd += ( particles[n].sigma - sigmamean ) * ( particles[
                n].sigma - sigmamean );
350         Sinitsd += ( particles[n].Sinit - Sinitmean ) * ( particles[
                n].Sinit - Sinitmean );
351         Iinitsd += ( particles[n].Iinit - Iinitmean ) * ( particles[
                n].Iinit - Iinitmean );
352         Rinitsd += ( particles[n].Rinit - Rinitmean ) * ( particles[
                n].Rinit - Rinitmean );
353 
354     }
355 
356     R0sd        /= NP;
357     rsd         /= NP;
358     sigmasd     /= NP;
359     Sinitsd     /= NP;
360     Iinitsd     /= NP;
361     Rinitsd     /= NP;
362 
363     if ( (R0sd + rsd + sigmasd) < 1e-5)
364         retVal = true;
365     else
366         retVal = false;
367 
368     return retVal;
369 
370 }
371 
372 void particleDiagnostics(ParticleInfo * partInfo, Particle *
        particles, int NP) {
```

```
373
374     double  R0mean      = 0.0,
375             rmean       = 0.0,
376             sigmamean   = 0.0,
377             Sinitmean   = 0.0,
378             Iinitmean   = 0.0,
379             Rinitmean   = 0.0;
380
381     // means
382
383     for (int n = 0; n < NP; n++) {
384
385         R0mean      += particles[n].R0;
386         rmean       += particles[n].r;
387         sigmamean   += particles[n].sigma;
388         Sinitmean   += particles[n].Sinit;
389         Iinitmean   += particles[n].Iinit;
390         Rinitmean   += particles[n].Rinit;
391
392     }
393
394     R0mean      /= NP;
395     rmean       /= NP;
396     sigmamean   /= NP;
397     Sinitmean   /= NP;
398     Iinitmean   /= NP;
399     Rinitmean   /= NP;
400
401     // standard deviations
402
403     double  R0sd    = 0.0,
404             rsd     = 0.0,
405             sigmasd = 0.0,
406             Sinitsd = 0.0,
407             Iinitsd = 0.0,
408             Rinitsd = 0.0;
409
410     for (int n = 0; n < NP; n++) {
411
412         R0sd    += ( particles[n].R0 - R0mean ) * ( particles[n].R0
                        - R0mean );
413         rsd     += ( particles[n].r - rmean ) * ( particles[n].r -
                        rmean );
414         sigmasd += ( particles[n].sigma - sigmamean ) * ( particles[
                        n].sigma - sigmamean );
415         Sinitsd += ( particles[n].Sinit - Sinitmean ) * ( particles[
                        n].Sinit - Sinitmean );
416         Iinitsd += ( particles[n].Iinit - Iinitmean ) * ( particles[
                        n].Iinit - Iinitmean );
417         Rinitsd += ( particles[n].Rinit - Rinitmean ) * ( particles[
```

```
            n].Rinit - Rinitmean );

    }

    R0sd        /= NP;
    rsd         /= NP;
    sigmasd     /= NP;
    Sinitsd     /= NP;
    Iinitsd     /= NP;
    Rinitsd     /= NP;

    partInfo->R0mean    = R0mean;
    partInfo->R0sd      = R0sd;
    partInfo->sigmamean = sigmamean;
    partInfo->sigmasd   = sigmasd;
    partInfo->rmean     = rmean;
    partInfo->rsd       = rsd;
    partInfo->Sinitmean = Sinitmean;
    partInfo->Sinitsd   = Sinitsd;
    partInfo->Iinitmean = Iinitmean;
    partInfo->Iinitsd   = Iinitsd;
    partInfo->Rinitmean = Rinitmean;
    partInfo->Rinitsd   = Rinitsd;

}

double randu() {

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

}


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

    double x1, x2, w, y1;

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

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

```
466
467     return y1;
468
469 }
```

Appendix C

Parameter Fitting

Appendix D

Forecasting Frameworks

D.1 IF2 Parametric Bootstrapping Function

The parametric bootstrapping machinery used to produce forecasts.

```
# Dexter Barrows
#
# IF2 parametric bootstrapping function

library(foreach)
library(parallel)
library(doParallel)
library(Rcpp)

if2_paraboot ← function(if2data_parent, T, Tlim, steps, N, nTrials,
    if2file, if2_s_file, stoc_sir_file, NP, nPasses, coolrate) {

  source(stoc_sir_file)

  if (nTrials < 2)
    ntrials ← 2

  # unpack if2 first fit data
  # ...parameters
  paramdata_parent ← data.frame( if2data_parent$paramdata )
  names(paramdata_parent) ← c("R0", "r", "sigma", "eta", "berr", "
      Sinit", "Iinit", "Rinit")
  parmeans_parent ← colMeans(paramdata_parent)
  names(parmeans_parent) ← c("R0", "r", "sigma", "eta", "berr", "
      Sinit", "Iinit", "Rinit")
  # ...states
```

```
24  statedata_parent ← data.frame( if2data_parent$statedata )
25  names(statedata_parent) ← c("S","I","R","B")
26  statemeans_parent ← colMeans(statedata_parent)
27  names(statemeans_parent) ← c("S","I","R","B")
28
29
30  ## use parametric bootstrapping to generate forcasts
31  ##
32  trajectories ← foreach( i = 1:nTrials, .combine = rbind, .packages
        = "Rcpp") %dopar% {
33
34    source(stoc_sir_file)
35
36    ## draw new data
37    ##
38
39    pars ← with( as.list(parmeans_parent),
40                 c(R0 = R0,
41                 r = r,
42                 N = N,
43                 eta = eta,
44                 berr = berr) )
45
46    init_cond ← with( as.list(parmeans_parent),
47                      c(S = Sinit,
48                        I = Iinit,
49                        R = Rinit) )
50
51    # generate trajectory
52    sdeout ← StocSIR(init_cond, pars, Tlim + 1, steps)
53    colnames(sdeout) ← c('S','I','R','B')
54
55    # add noise
56    counts_raw ← sdeout[,'I'] + rnorm(dim(sdeout)[1], 0, parmeans_
         parent[['sigma']])
57      counts     ← ifelse(counts_raw < 0, 0, counts_raw)
58
59      ## refit using new data
60      ##
61
62      rm(if2) # because stupid things get done in packages
63      sourceCpp(if2file)
64      if2time ← system.time( if2data ← if2(counts, Tlim+1, N, NP,
           nPasses, coolrate) )
65
66      paramdata ← data.frame( if2data$paramdata )
67    names(paramdata) ← c("R0", "r", "sigma", "eta", "berr", "Sinit",
         "Iinit", "Rinit")
68    parmeans ← colMeans(paramdata)
69    names(parmeans) ← c("R0", "r", "sigma", "eta", "berr", "Sinit",
```

```
        "Iinit", "Rinit")

    ## generate the rest of the trajectory
    ##

    # pack new parameter estimates
    pars ← with(  as.list(parmeans),
                  c(R0 = R0,
                  r = r,
                  N = N,
                  eta = eta,
                  berr = berr) )
    init_cond ← c(S = statemeans_parent[['S']],
                      I = statemeans_parent[['I']],
                      R = statemeans_parent[['R']])

    # generate remaining trajectory part
    sdeout_future ← StocSIR(init_cond, pars, T-Tlim, steps)
    colnames(sdeout_future) ← c('S','I','R','B')

    return ( c( counts = unname(sdeout_future[,'I']),
                parmeans,
                time = if2time[['user.self']]) )


  }

  return(trajectories)

}
```

D.2 RStan Forward Simulator

The code used to reconstruct the state estimates, then project the trajectory forward past data.

```
StocSIRstan ← function(y, pars, T, steps, berrvec, bveclim) {

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

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

```

```
  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) ) {

      if (i <= bveclim) {
        B ← exp( log(B) + eta*(log(B0) - log(B)) + berrvec[i])
      } else {
          B ← exp( log(B) + eta*(log(B0) - log(B)) + rnorm(1, 0,
              berr))
      }

    BSI ← B*S*I
    rI ← r*I

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

    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)

  }

  return(out)

}
```

Appendix E

S-map and SIRS

E.1 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)
```

```
27
28      B ← exp( log(B) + eta*(log(B0) - log(B)) + rnorm(1, 0, berr) )
29
30      BSI ← Bfac*B*S*I
31      rI ← r*I
32          reR ← re*R
33
34      dS ← -BSI + reR
35      dI ← BSI - rI
36      dR ← rI - reR
37
38      S ← S + h*dS  #newInf
39      I ← I + h*dI  #newInf - h*dR
40      R ← R + h*dR  #h*dR
41
42      if (i %% steps == 0)
43        out[i/steps+1,] ← c(S,I,R,B)
44
45    }
46
47    colnames(out) ← c("S","I","R","B")
48    return(out)
49
50  }
51
52  ### suggested parameters
53  #
54  # T     ← 200
55  # i_infec   ← 10
56  # steps   ← 7
57  # N     ← 500
58  # sigma   ← 5
59  #
60  # pars ← c(R0 = 3.0,  # new infected people per infected person
61  #            r = 0.1,  # recovery rate
62  #        N = 500,    # population size
63  #        eta = 0.5,  # geometric random walk
64  #        berr = 0.5, # Beta geometric walk noise
65  #            re = 1)   # resuceptibility rate
```

E.2 SMAP Code

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

```
1  library(pracma)
2
```

```
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

```

```
52          # next predictee
53
54          #predictee ← c( predsum, predictee[-E] )
55          #allPredictees[i,] ← predictee
56
57      }
58
59      return(predictions)
60
61  }
```

E.3 SMAP Parameter Optimization Code

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

```
1  library(deSolve)
2  library(ggplot2)
3  library(RColorBrewer)
4  library(pracma)
5
6  set.seed(1010)
7
8  ## external files
9  ##
10 stoc_sirs_file  ← paste(getwd(), "../sir-functions", "StocSIRS.r",
       sep = "/")
11 smap_file     ← paste(getwd(), "smap.r", sep = "/")
12 source(stoc_sirs_file)
13 source(smap_file)
14
15
16
17 ## parameters
18 ##
19 T     ← 6*52
20 Tlim  ← T - 52
21 i_infec ← 10
22 steps   ← 7
23 N     ← 500
24 sigma   ← 5
25
26 true_pars ← c(  R0 = 3.0,   # new infected people per infected
       person
27              r = 0.1,  # recovery rate
28          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)
```

E.4 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);

}
```

E.5 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;
```

```
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
187             }
188
189             // cumulative sum
190             for (int n = 1; n < NP; n++) {
191                 w[n] += w[n-1];
192             }
193
194             // save particle states to resample from
195             for (int n = 0; n < NP; n++){
196                 copyParticle(&particles_old[n], &particles[n]);
197             }
198
199             // resampling
200             for (int n = 0; n < NP; n++) {
201
202                 double w_r = randu() * w[NP-1];
203                 int i = 0;
204                 while (w_r > w[i]) {
205                     i++;
206                 }
207
208                 // i is now the index to copy state from
209                 copyParticle(&particles[n], &particles_old[i]);
210
211             }
212
213             // between-iteration perturbations, not after last time
                    step
214             if (t < (T-1))
215                 perturbParticles(particles, N, NP, pass, coolrate);
216
217             if (pass == (nPasses-1)) {
218                 State sMeans;
219                 getStateMeans(&sMeans, particles, NP);
220                 statemeans(t,0) = sMeans.S;
221                 statemeans(t,1) = sMeans.I;
222                 statemeans(t,2) = sMeans.R;
223             }
224
```

```
225         }
226
227         ParticleInfo pInfo;
228         particleDiagnostics(&pInfo, particles, NP);
229
230         means(pass, 0) = pInfo.R0mean;
231         means(pass, 1) = pInfo.rmean;
232         means(pass, 2) = pInfo.remean;
233         means(pass, 3) = pInfo.sigmamean;
234         means(pass, 4) = pInfo.etamean;
235         means(pass, 5) = pInfo.berrmean;
236         means(pass, 6) = pInfo.Sinitmean;
237         means(pass, 7) = pInfo.Iinitmean;
238         means(pass, 8) = pInfo.Rinitmean;
239
240         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 );
504         sigmasd += ( particles[n].sigma - sigmamean ) * ( particles[
                        n].sigma - sigmamean );
505         etasd   += ( particles[n].eta - etamean ) * ( particles[n].
                        eta - etamean );
506         berrsd  += ( particles[n].berr - berrmean ) * ( particles[n
                        ].berr - berrmean );
507         Sinitsd += ( particles[n].Sinit - Sinitmean ) * ( particles[
                        n].Sinit - Sinitmean );
```

```
508         Iinitsd += ( particles[n].Iinit - Iinitmean ) * ( particles[
                n].Iinit - Iinitmean );
509         Rinitsd += ( particles[n].Rinit - Rinitmean ) * ( particles[
                n].Rinit - Rinitmean );
510
511     }
512
513     R0sd        /= NP;
514     rsd         /= NP;
515     resd        /= NP;
516     sigmasd     /= NP;
517     etasd       /= NP;
518     berrsd      /= NP;
519     Sinitsd     /= NP;
520     Iinitsd     /= NP;
521     Rinitsd     /= NP;
522
523     partInfo->R0mean    = R0mean;
524     partInfo->R0sd      = R0sd;
525     partInfo->rmean     = rmean;
526     partInfo->rsd       = rsd;
527     partInfo->remean    = remean;
528     partInfo->resd      = resd;
529     partInfo->sigmamean = sigmamean;
530     partInfo->sigmasd   = sigmasd;
531     partInfo->etamean   = etamean;
532     partInfo->etasd     = etasd;
533     partInfo->berrmean  = berrmean;
534     partInfo->berrsd    = berrsd;
535     partInfo->Sinitmean = Sinitmean;
536     partInfo->Sinitsd   = Sinitsd;
537     partInfo->Iinitmean = Iinitmean;
538     partInfo->Iinitsd   = Iinitsd;
539     partInfo->Rinitmean = Rinitmean;
540     partInfo->Rinitsd   = Rinitsd;
541
542 }
543
544 double randu() {
545
546     return (double) rand() / (double) RAND_MAX;
547
548 }
549
550 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 }
```

Appendix F

Spatial Epidemics

F.1 Spatial SIR 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
```

F.2 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
```

```
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, t-1]) ) + Bnoise[loc, t] );
55
56              n ← neinum[loc];
57              sphi ← 1.0 - phi*( n/(n+1.0) );
58              ophi ← phi/(n+1.0);
59
60              nBIsum ← 0.0;
61              for (j in 1:n)
62                  nBIsum ← nBIsum + B[neibmat[loc, j], t-1] * I[
                        neibmat[loc, j], t-1];
```

```
63
64              BSI[loc, t] ← S[loc, t-1]*( sphi*B[loc, t-1]*I[loc, t-1]
                    + ophi*nBIsum );
65              rI[loc, t]  ← r*I[loc, t-1];
66
67              S[loc, t] ← S[loc, t-1] + h*( - BSI[loc, t] );
68              I[loc, t] ← I[loc, t-1] + h*( BSI[loc, t] - rI[loc, t] )
                    ;
69              R[loc, t] ← R[loc, t-1] + h*( rI[loc, t] );
70
71              if (y[loc, t] > 0) {
72                  y[loc, t] ~ normal( I[loc, t], sigma );
73              }
74
75          }
76      }
77
78      R0      ~ lognormal(1,1);
79      r       ~ lognormal(1,1);
80      sigma   ~ lognormal(1,1);
81      for (loc in 1:nloc) {
82          Iinit[loc] ~ normal(y[loc, 1], sigma);
83      }
84
85  }
```

F.3 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[0], particles[0].R[0]);
98      */
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
        forward in time
244     double h    - time step size
245     double t0   - start time
246     double tn   - stop time
247     double * y  - current system state; a three-component vector
            representing [S I R], susceptible-infected-recovered
248
249     */
250 void exp_euler_SSIR(double h, double t0, double tn, int N, Particle
        * particle,
251                     NumericVector neinum, NumericMatrix neibmat, int
                            nloc) {
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 nBIsum = 0.0;
293             for (int j = 0; j < n; j++)
294                 nBIsum += 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*nBIsum );
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|\n", sphi, ophi, BSI, rI, dS, dI, dR, S[1], I
                [1], R[1]);
311 
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 arays 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 /*  Convinience 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 deviation 1
496     Uses the polar form of the Box-Muller transformation
497     From http://www.design.caltech.edu/erik/Misc/Gaussian.html
498     */
499 double randn() {
500
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 }
```