FUN WITH FORECASTING USING STOCHASTIC NON-LINEAR 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 22, 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

Sooooooo many people

Contents

List of Figures

Chapter 1

Introduction

Epidemic forecasting is an important tool that can help inform public policy and decision-making in the face of an infectious disease outbreak. Successful intervention relies on accurate predictions of the number of cases, when they will occur, and where. WIthout this information it is difficult to efficiently allocate resources, a critical step in curbing the size and breadth of an epidemic.

Despite the importance of reliable forecasts, obtaining them remains a challenge both from a theoretical and practical standpoint. Mathematical models can capture the essential drivers in disease dynamics, and extended past the present into the future. However, different epidemics may present with varying dynamics and require different model parameters to be accurately represented. These parameters can be inferred by using statistical model fitting techniques, but this can become computationally intensive, and the modeller risks "overfitting" by attempting to capture too many drivers with too little data. Thus, The modeller must exercise restraint in model selection and fitting technique.

Securing precise, error-free data in the midst of an outbreak can be difficult if not impossible, so uncertainty in what we observe in building mathematical models of disease spread must be accounted for from the get-go. Further, models must differentiate between natural variation in the intensity of disease spread (process error) and error in data collection (observation error) in order to accurately determine the dynamics underlying a data set.

Broadly, there are three primary categories of techniques used in forecasting: phenomenological, pure mechanistic, and semi-mechanistic.

Phenomenological methods operate purely on data, fitting models that do not try to reconstruct disease dynamics, but rather focus purely on trend. A long-standing and widely-used example is the Autoregressive Integrated Moving Average (ARIMA) model. ARIMA assumes a linear underlying process and Gaussian error distributions. It uses three parameters representing the degree of autoregression (p), integration (trend removal) (d), and the moving average (q), where the orders of the autoregression and the moving average are determined through the use of an autocorrelation function (ACF) and partial autocorrelation function (PACF), respectively, applied to the the data *a priori*.

Pure mechanistic approaches simply try to capture the essential drivers in the disease spreading process and use the model alone to generate predictions. For example one could use a compartment model in which individuals are divided into categories based on whether they are susceptible to infection or infected but not yet themselves infectious, infectious, or recovered. These models are referred to as susceptible-infectious-removed (SIR) models and are heavily used in epidemiological study. Typically the transition between compartments is governed by a set of ordinary differential equations, such as

$$\begin{aligned} \frac{dS}{dt} &= -\beta I S \\ \frac{dI}{dt} &= \beta I S - \gamma I \\ \frac{dR}{dt} &= \gamma I, \end{aligned} \tag{1.1}$$

where S, I, and R are the number of individuals in each compartment, β is the “force” of infection acting on the susceptible population, and γ is a recovery rate. As an outbreak progresses, individuals transition from the susceptible compartment, through the infectious compartment, then finish in the removed compartment where they no longer impact the system dynamics. Many extensions of the SIR model exist are are commonly used, such as the SEIR model in which susceptible individuals pass through an exposed class where they have been infected but are not yet themselves infectious, and the SIRS model in which individuals become susceptible again after their immunity wanes.

Combining the phenomenological and mechanistic approaches are the semi-mechanistic techniques. These methods use a model to define the expected underlying dynamics of the system, but integrate data into the model in order to refine estimates of the model parameters and produce more accurate forecasts. Typically the first step in implementing such a technique is fitting the desired model to existing data. There are many ways to do this, most of which fall into two main categories: particle filter-based (PF) methods, and Markov chain Monte Carlo-based (MCMC) methods. From there data can either be integrated into the model by refitting the model to the new longer data set, or in an “on-line” fashion in which data points can be directly integrated without the need to refit the entire model. Normally, MCMC-based machinery must refit the entire model whereas PF-based approaches can sometimes integrate data in an on-line fashion.

Another, broader, distinction among techniques can be drawn between those that rely on assumptions of linearity, and those that make no such assumption. As epidemic dynamics are highly non-linear, it can be questionable as to even consider linear approaches to epidemic forecasting at all. In particular, stalwart approaches such as ARIMA and the venerable Kalman filter face a distinct (at least theoretical) disadvantage in the face of newer PF-based methods. Additionally, these methods are very-well-studied, and further work showing their viability would likely prove extraneous in the modern academic landscape.

Somewhat frustratingly, there exists no “gold standard” in forecasting. As methodology varies widely in theoretical justification, implementation, and operation, it is difficult to compare the state of the art in forecasting methods from a first-principles perspective. Further, published work using any of these methods to forecast uses different prediction accuracy metrics, such as SSE, peak time/duration/intensity, correlation tests, or RMSE, among others. Thus is is difficult to select the best tool for the job when faced with a forecasting problem.

The primary focus of this work is to compare best-in-class methods for forecasting in several epidemically-focused scenarios. These include the a “standard” one-shot forecast outbreak in which the outbreak subsides and does not recur, a seasonal outbreak scenario such as the one we see with influenza each year, and a spatiotemporal scenario in which multiple spatial location are connected and disease is free to spread from one to another.

For techniques we have the following: from MCMC-based methods we have selected Hamiltonian MCMC [*ref*], a less recent but nonetheless highly effective technique, from PF-based methods we have selected IF2 [*Ionides ref*], a newer approach that uses multiple particle filtering rounds to generate MLEs, and from the phenomenological methods we have selected the sequential locally weighted global linear maps (S-map) [*Sugihara ref*].

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

$$T = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0.1 & 0.9 \\ 0.6 & 0.4 & 0 \end{bmatrix}. \tag{2.1}$$

The probability vector $\mu(x^{(1)})$ for a state $x^{(1)}$ can be evolved using T by evaluating

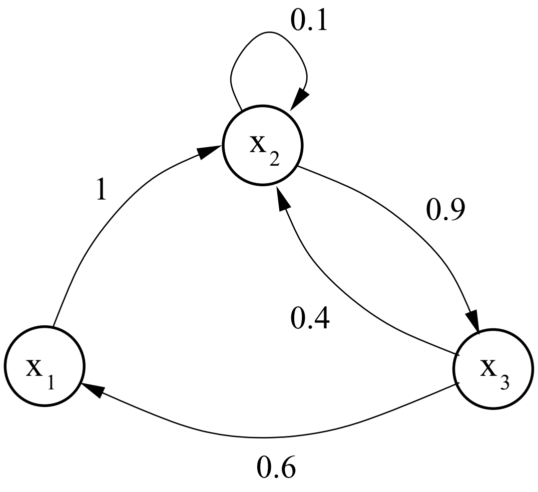

Figure 2.1: Finite state machine. *(Andrieu et al., 2003)*

$\mu(x^{(1)})T$, 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

$$\lim_{t\to\infty} \mu(x^{(1)})T^t = (27/122, 50/122, 45/122). \tag{2.2}$$

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

$$\mathcal{L}(\theta) \propto P(D|\theta). \tag{2.3}$$

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

$$r = \frac{\mathcal{L}(\theta^*)p(\theta^*)q(\theta^*|\theta)}{\mathcal{L}(\theta)p(\theta)q(\theta|\theta^*)}, \tag{2.4}$$

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

$$r = \frac{\mathcal{L}(\theta^*)p(\theta^*)}{\mathcal{L}(\theta)p(\theta)}. \tag{2.5}$$

Thus, the MCMC algorithm shown in Algorithm [1].

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.

Algorithm 1: Metropolis-Hastings MCMC

```
/* Select a starting point                                        */
Input  : Initialize $\theta^{(1)}$
for $i = 2 : N$ do
    /* Sample                                                     */
    $\theta^* \sim q(\cdot|\theta^{(i-1)})$
    $u \sim \mathcal{U}(0,1)$
    /* Evaluate acceptance ratio                                  */
    $r \leftarrow \frac{\mathcal{L}(\theta^*)p(\theta^*)}{\mathcal{L}(\theta)p(\theta)}$
    /* 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 $(\theta^{(1)}, \theta^{(2)}, ..., \theta^{(N)})$
```

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

$$U(\theta) = -log(\mathcal{L}(\theta)p(\theta)). \tag{2.6}$$

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

$$K(r) = \frac{1}{2} r^T M^{-1} r, \tag{2.7}$$

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

$$H(\theta, r) = U(\theta) + K(r), \tag{2.8}$$

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

$$\begin{aligned} \frac{d\theta}{dt} &= M^{-1} r \\ \frac{dr}{dt} &= -\nabla U(\theta) \end{aligned} \tag{2.9}$$

.

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

$$r = \exp\left[H(\theta, r) - H(\theta^*, r^*)\right], \tag{2.10}$$

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.

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 $(\theta^{(1)}, \theta^{(2)}, ..., \theta^{(N)})$
```

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

$$\begin{aligned}\frac{dS}{dt} &= -\beta IS\\ \frac{dI}{dt} &= \beta IS - rI\\ \frac{dR}{dt} &= rI\end{aligned} \tag{2.11}$$

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

```
  T ← 100                              # total integration time
  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

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

The `ode()` function is called as

```
  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].

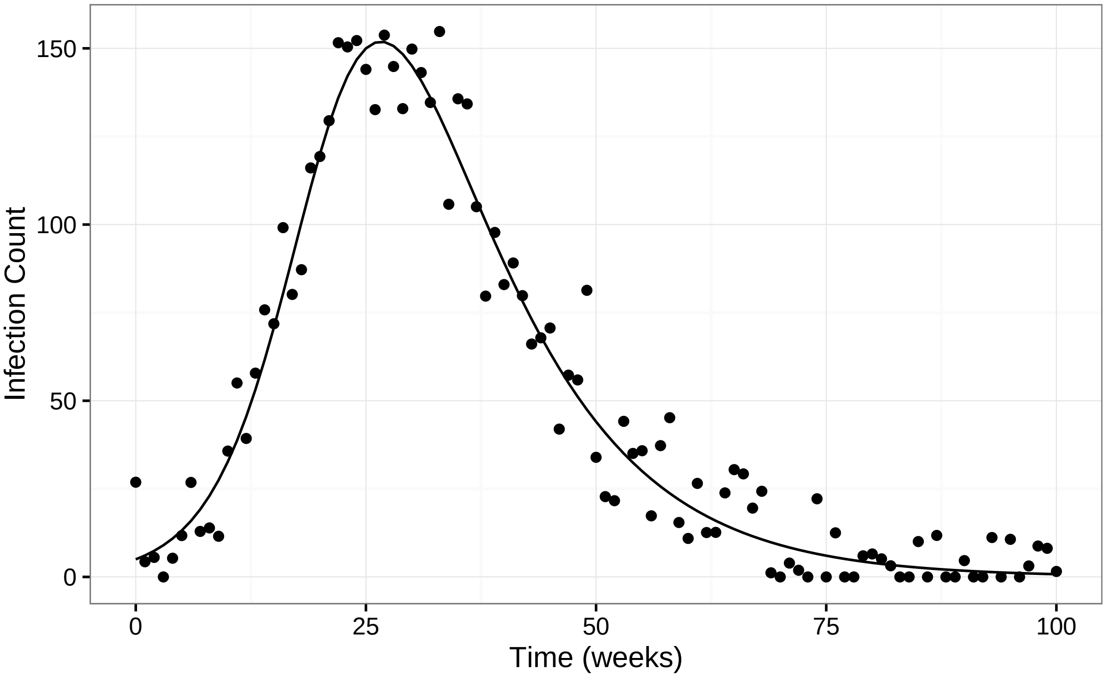

Figure 2.2: True SIR ODE solution infected counts, and with added observation noise

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

    }
```

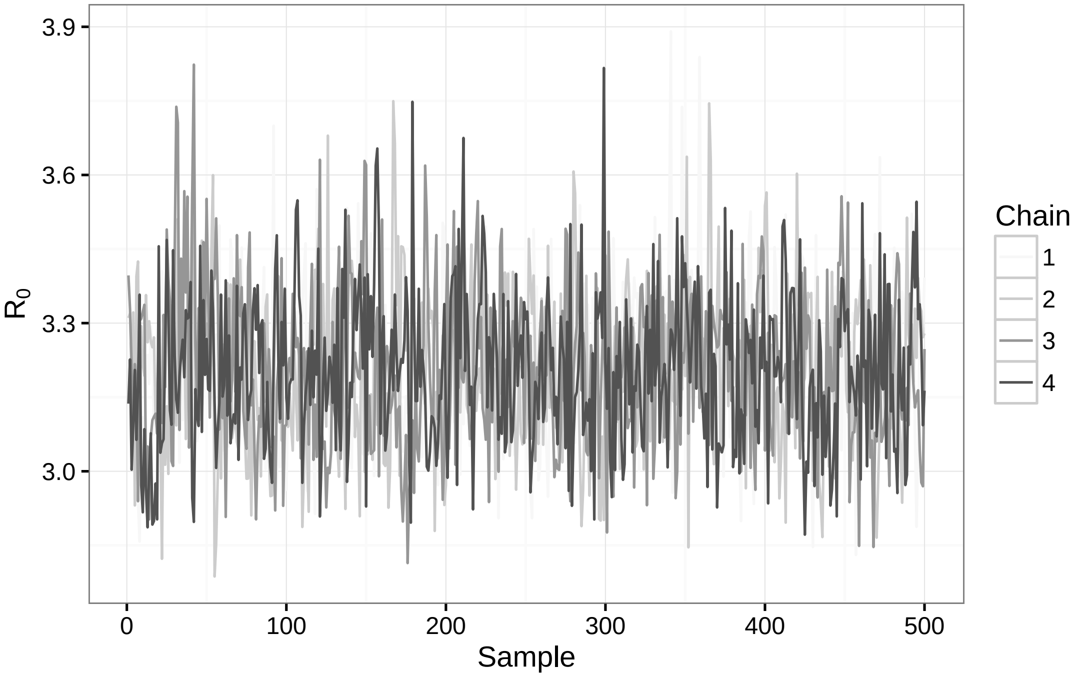

Figure 2.3: Traceplot of samples drawn for parameter R_0, excluding warmup

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 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 in Figure [2.5].

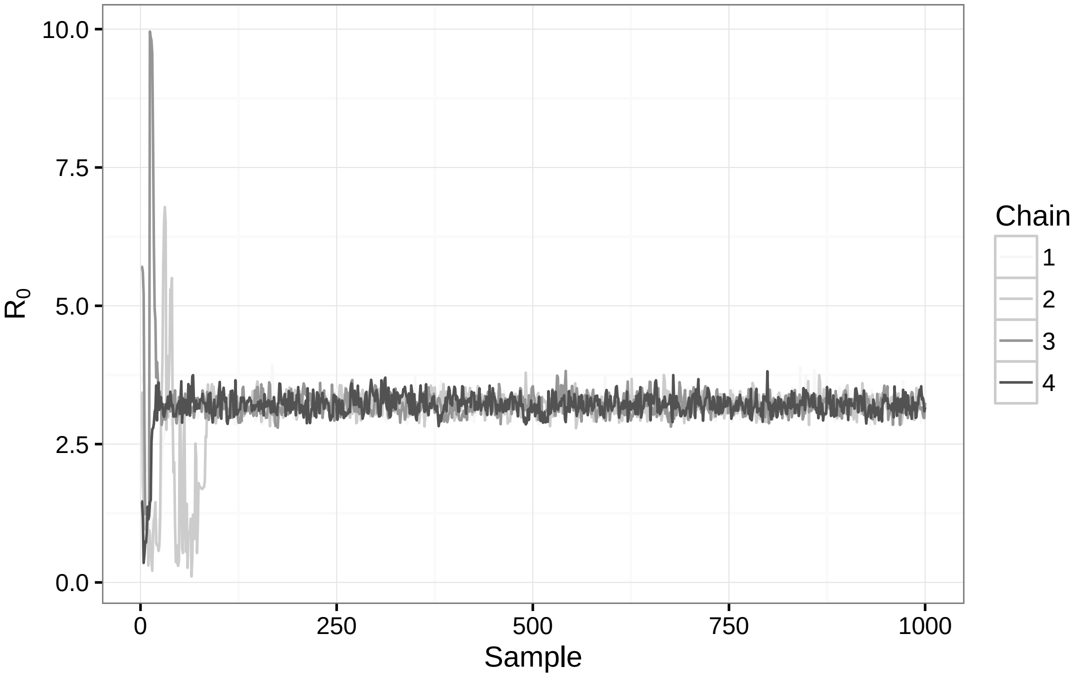

Figure 2.4: Traceplot of samples drawn for parameter R_0, including warmup.

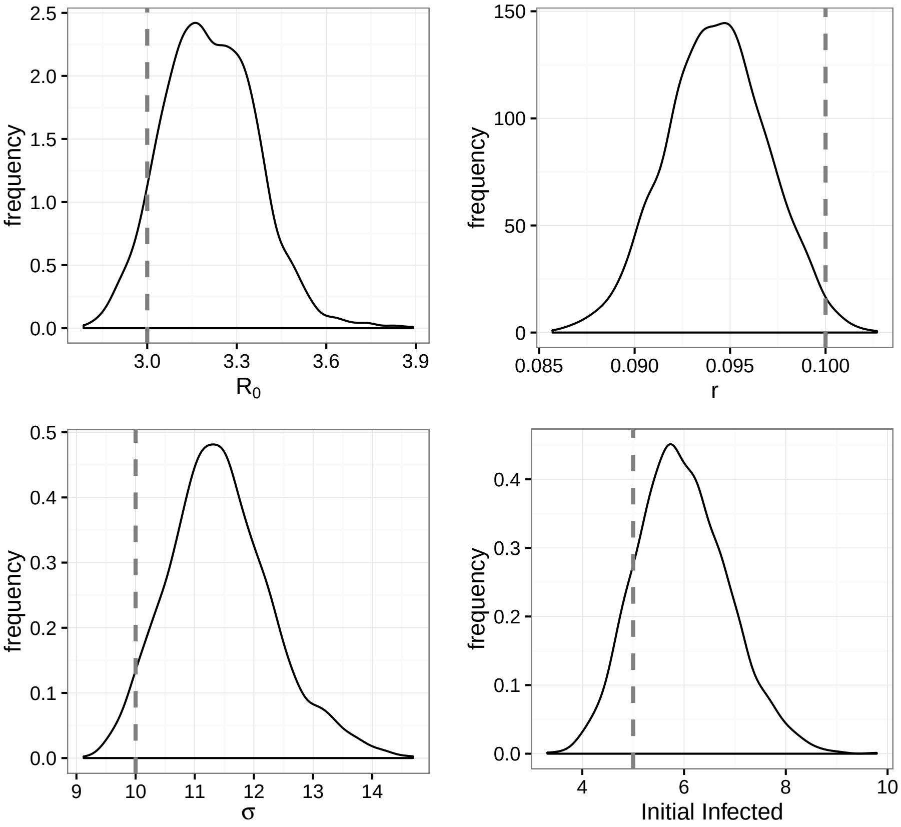

Figure 2.5: Kernel density estimates produced by Stan

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"))
```

we obtain Figure [3.1].

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="/"))
```

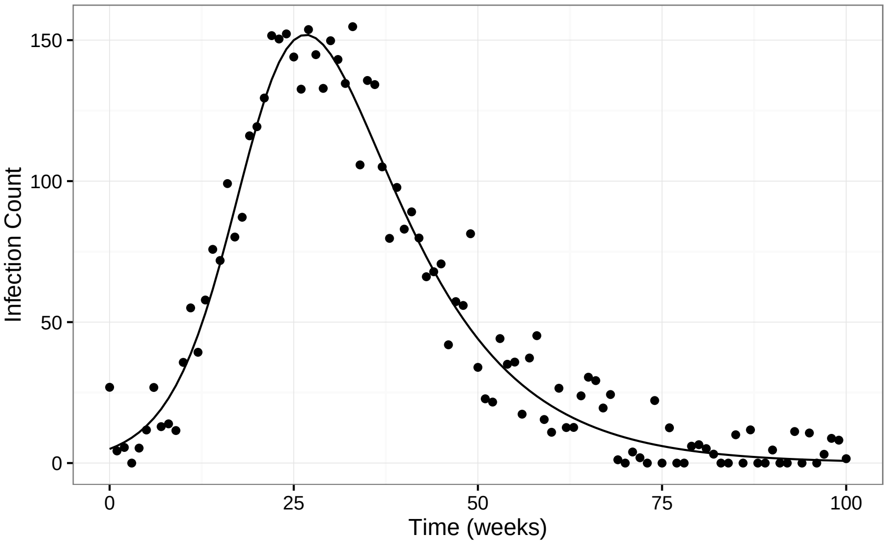

Figure 3.1: True SIR ODE solution infected counts, and with added observation noise.

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].

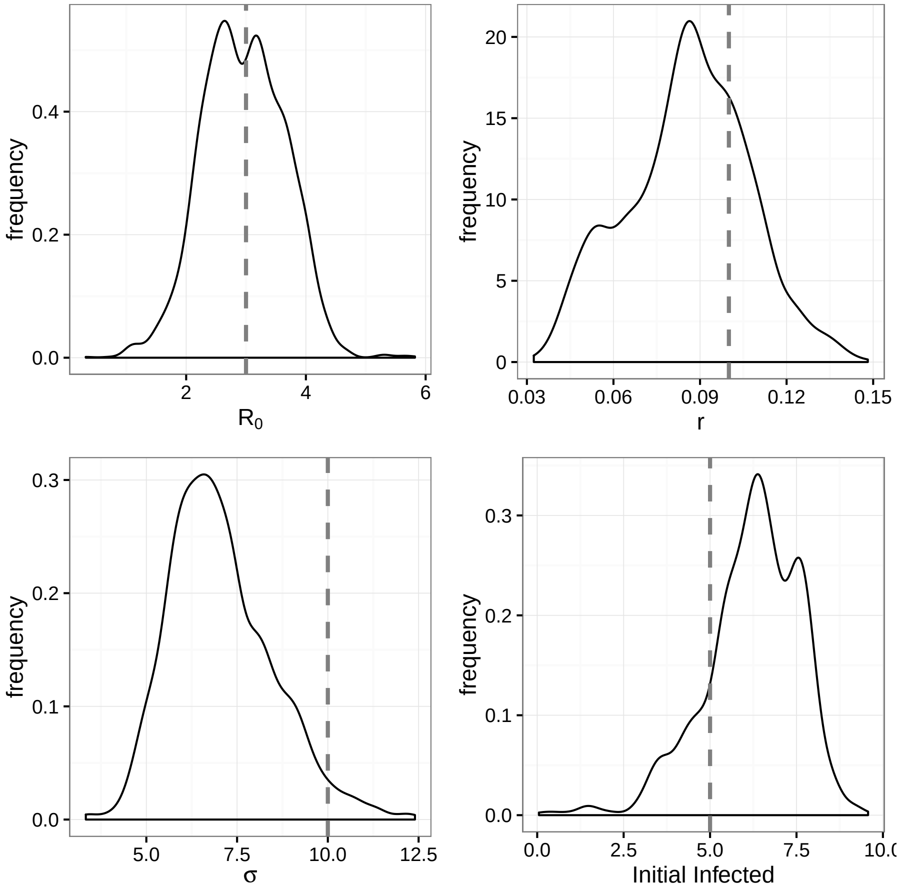

Figure 3.2: Kernel estimates for four essential system parameters. True values are indicated by solid vertical lines, sample means by dashed lines.

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 $\epsilon_t \sim \mathcal{N}(0, \rho^2)$. 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).

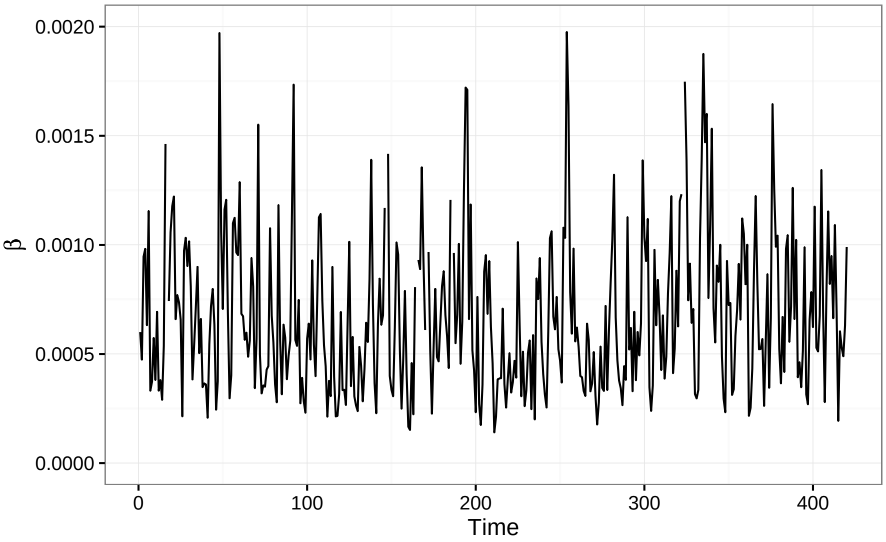

Figure 4.1: Simulated geometric autoregressive process show in Equation [4.2].

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].

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 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].

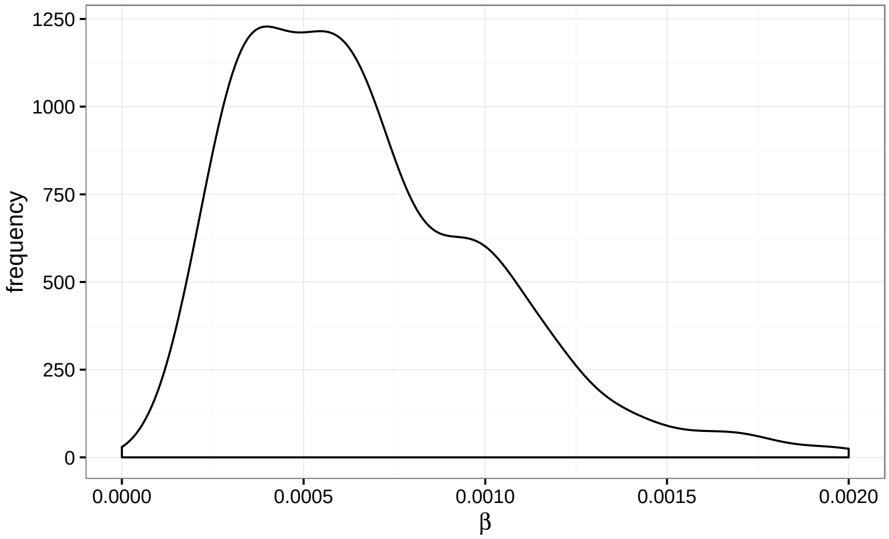

Figure 4.2: Density plot of values shown if Figure[4.1].

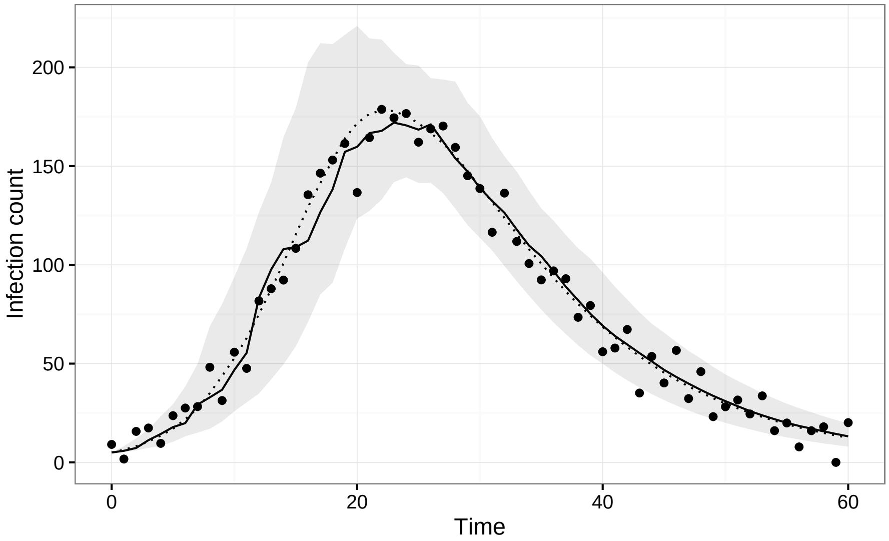

Figure 4.3: Stochastic SIR model simulated using an explicit Euler stepping scheme. The solid line is a single random trajectory, the dots show the data points obtained by adding in observation error defined as $\epsilon_{obvs} = \mathcal{N}(0, 10)$, and the grey ribbon is centre 95th quantile from 100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

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.

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.

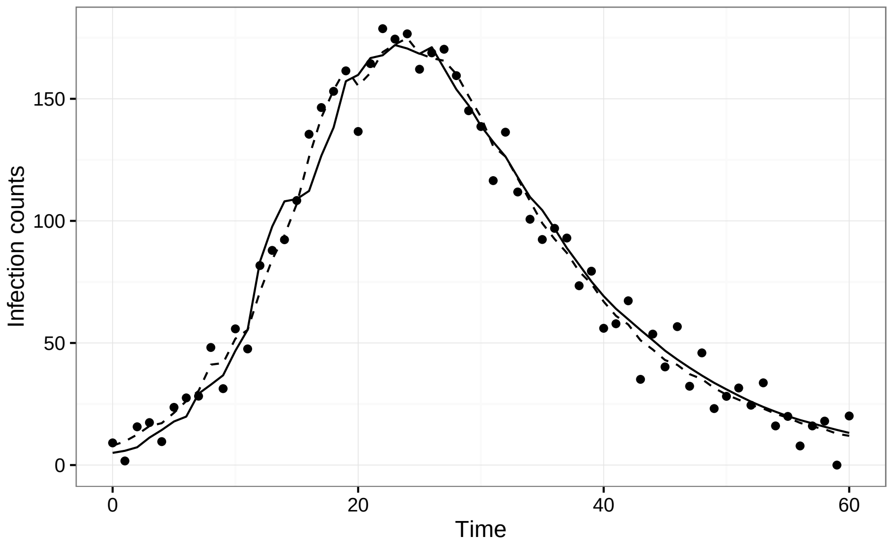

Figure 4.5: True system trajectory (solid line), observed data (dots), and IF2 estimated real state (dashed line).

		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].

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].

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

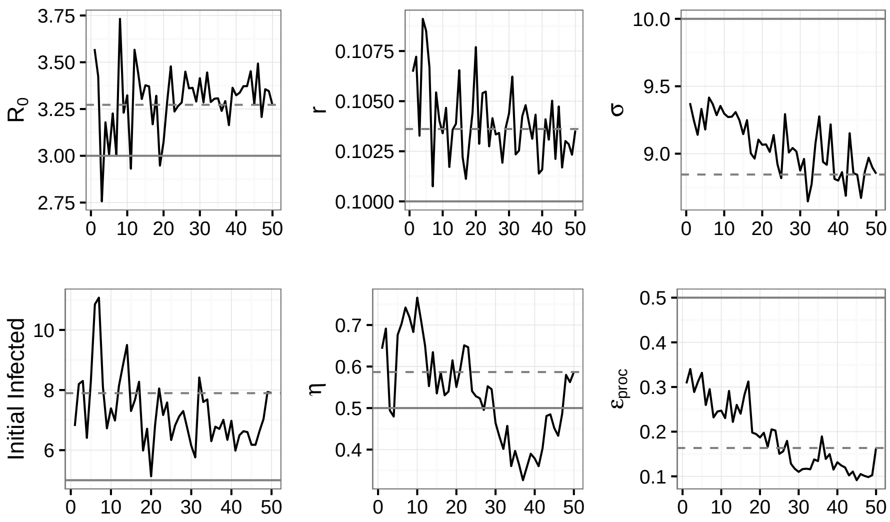

Figure 4.6: The horizontal axis shows the IF2 pass number. The solid black lines show the evolution of the ML estimates, the solid grey lines show the true value, and the dashed grey lines show the mean parameter estimates from the particle swarm after the final pass.

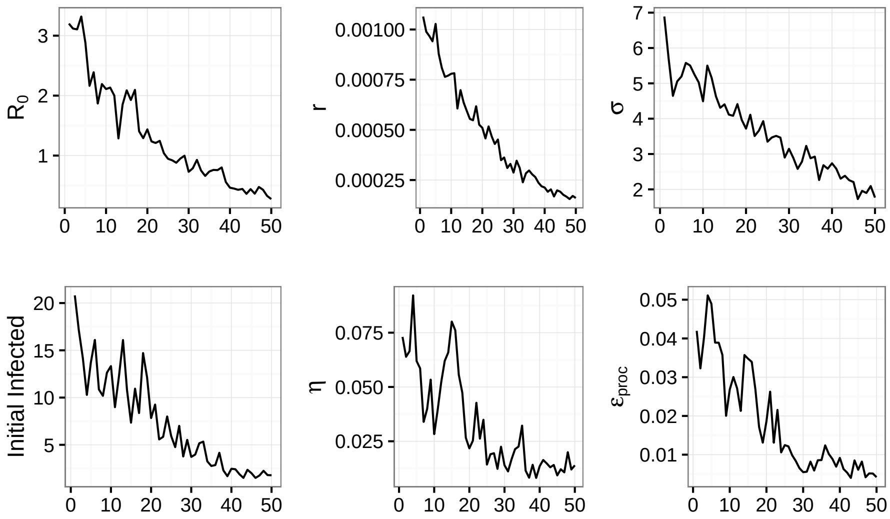

Figure 4.7: The horizontal axis shows the IF2 pass number and the solid black lines show the evolution of the standard deviations of the particle swarm values.

4.5 IF2 Densities

Of diagnostic importance are the densities of the parameter estimates given by the final parameter swarm. These are shown if Figure [4.8].

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.

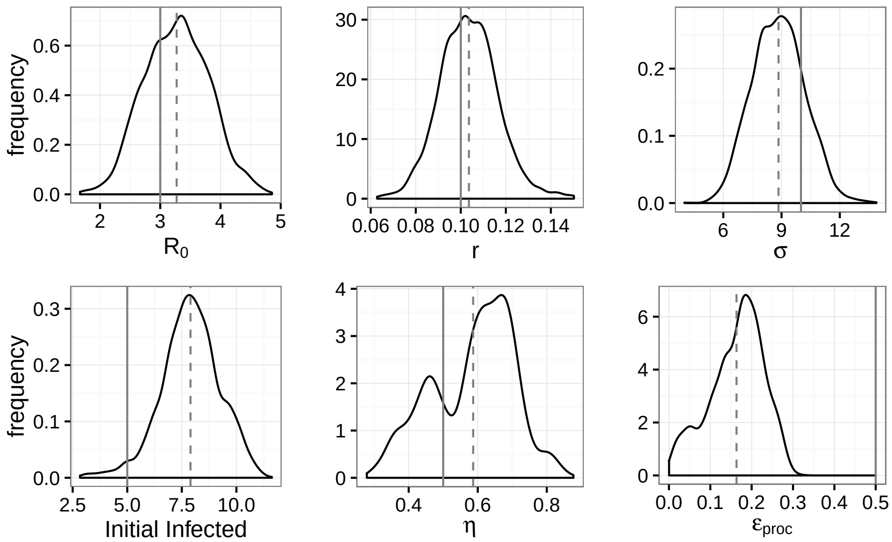

Figure 4.8: As before, the solid grey lines show the true parameter values and the dashed grey lines show the density means.

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.

4.7 HMCMC Densities

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

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.

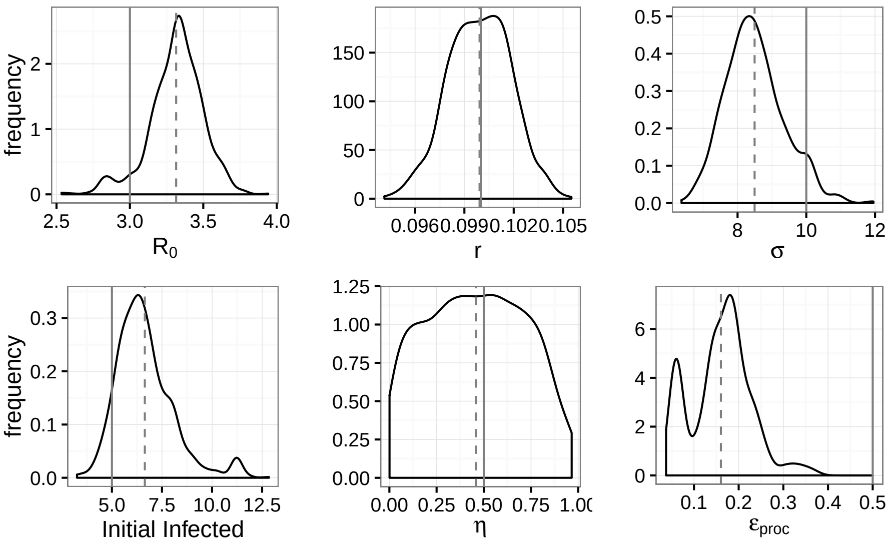

Figure 4.9: As before, the solid grey lines show the true parameter values and the dashed grey lines show the density means.

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].

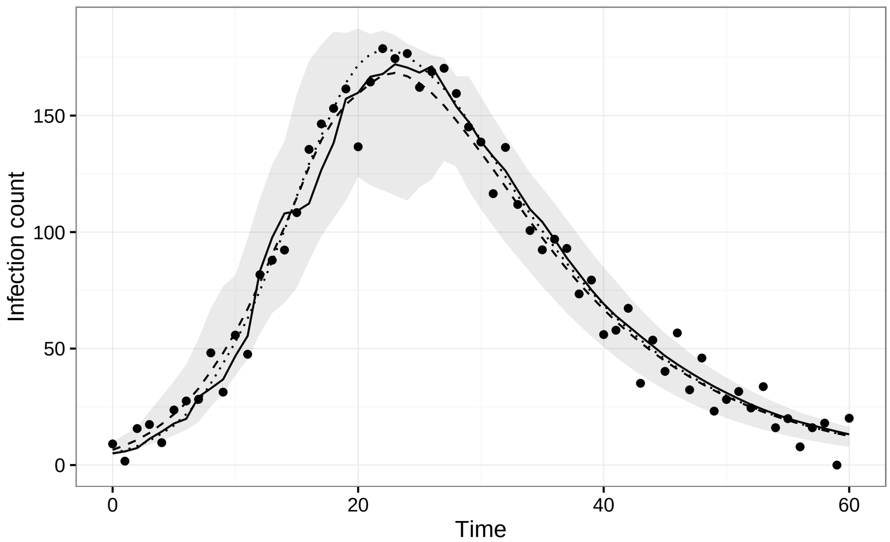

Figure 4.10: Result from 100 HMCMC bootstrap trajectories. The solid line shows the true states, the dots show the data, the dotted line shows the average system behaviour, the dashed line shows the bootstrap mean, and the grey ribbon shows the centre 95th quantile of the bootstrap trajectories.

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 [4.12].

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 [4.12] 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.

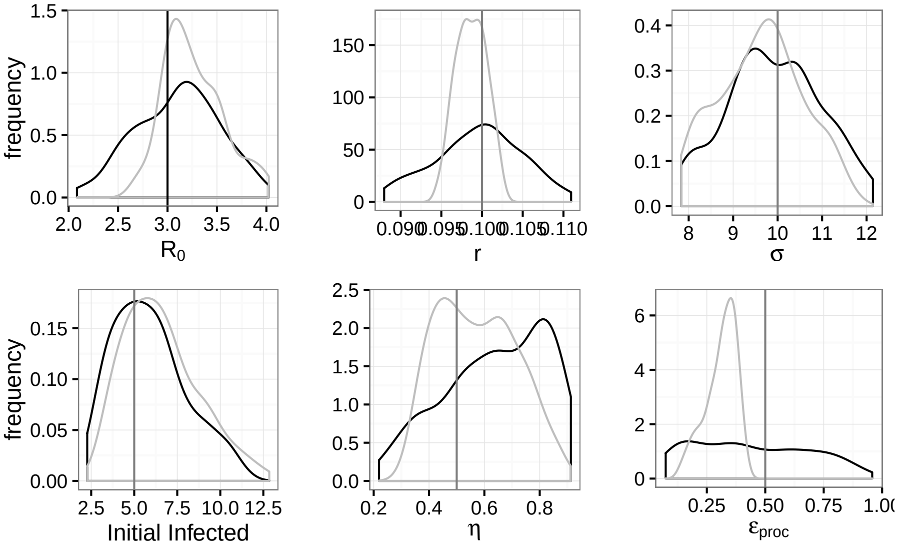

Figure 4.11: IF2 point estimate densities are shown in black and HMCMC point estimate densities are shown in grey. The vertical black lines show the true parameter values.

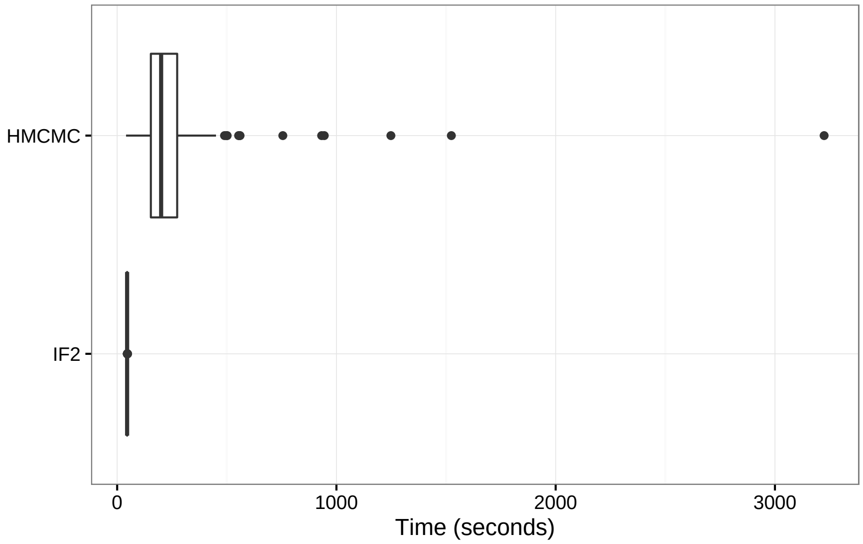

Figure 4.12: Fitting times for IF2 and HMCMC, in seconds. The centre box in each plot shows the centre 50th quantile, with the bold centre line showing the median.

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 [5.1].

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.

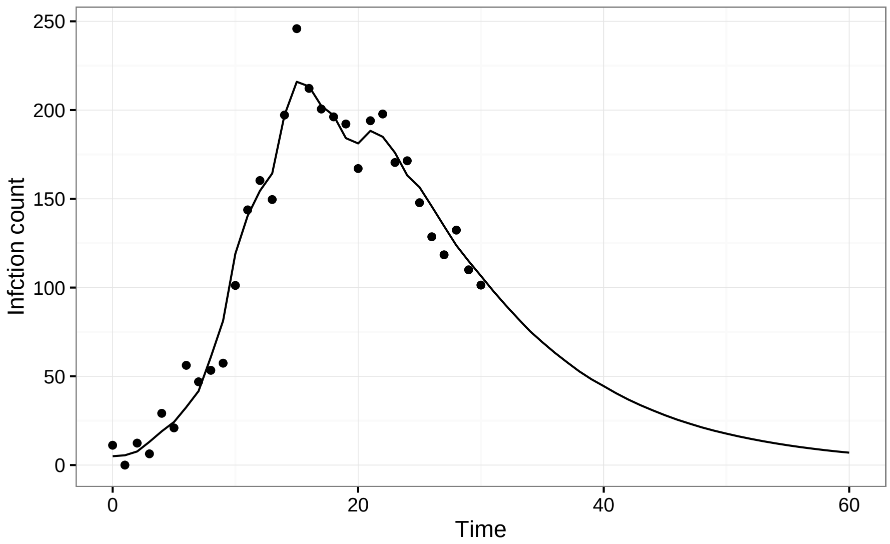

Figure 5.1: Infection count data truncated at $T = 30$. The solid line shows the true underlying system states, and the dots show those states with added observation noise. Parameters used were $R_0 = 3.0$, $r = 0.1$, $\eta = .05$, $\sigma_{proc} = 0.5$, and additive observation noise was drawn from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

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.

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.

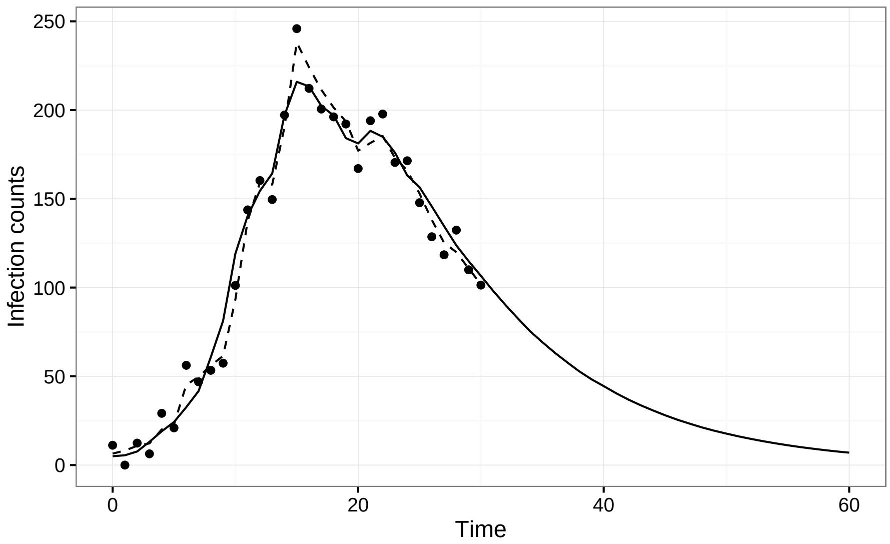

Figure 5.2: Infection count data truncated at $T = 30$ from Figure [5.1]. The dashed line shows IF2's attempt to reconstruct the true underlying state from the observed data points.

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 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 $\theta_1, \theta_2, ..., \theta_M$
```

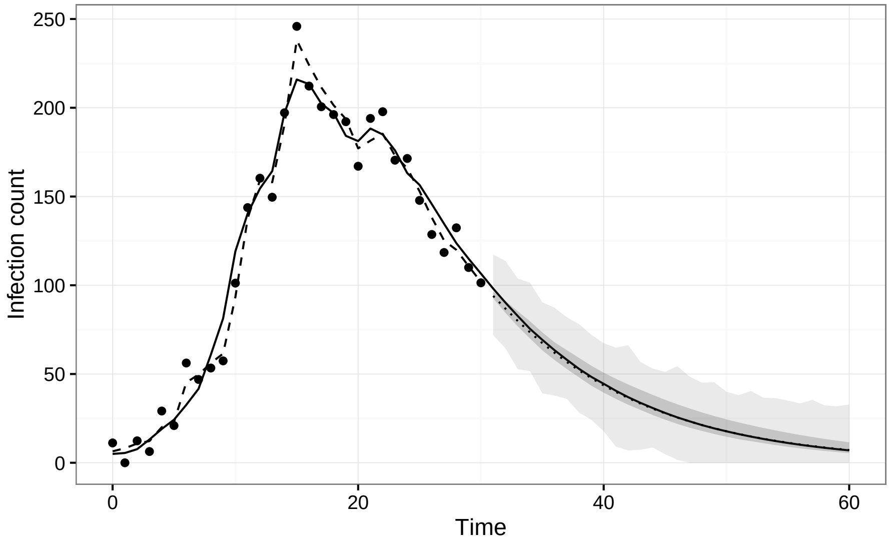

Figure 5.3: Forecast produced by the IF2 / parametric bootstrapping framework. The dotted line shows the mean estimate of the forecasts, the dark grey ribbon shows the centre 95th quantile of the true state estimates, and the lighter grey ribbon shows the centre 95th quantile of the true state estimates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

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 $\overline{SSE} = 20.27$.

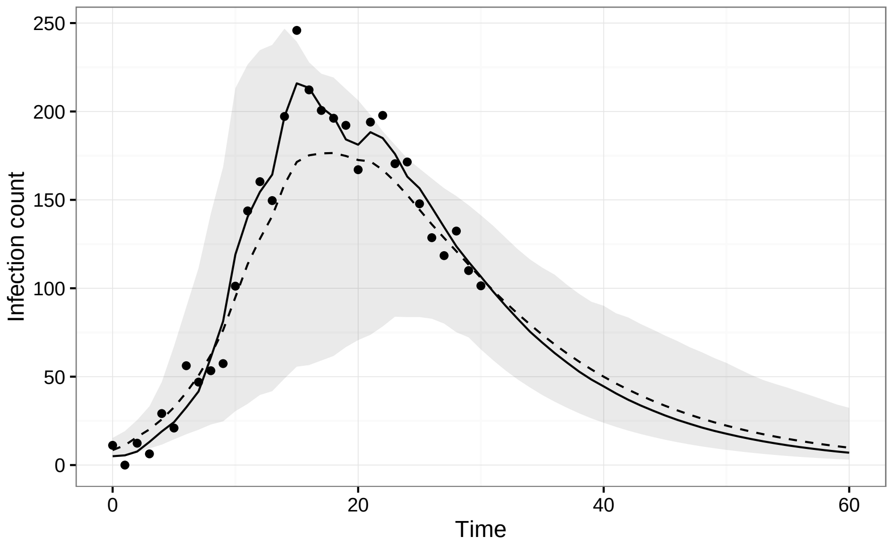

Figure 5.4: Forecast produced by the HMCMC / bootstrapping framework with $M = 200$ trajectories. The dotted line shows the mean estimate of the forecasts, and the grey ribbon shows the centre 95th quantile.

5.4 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 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

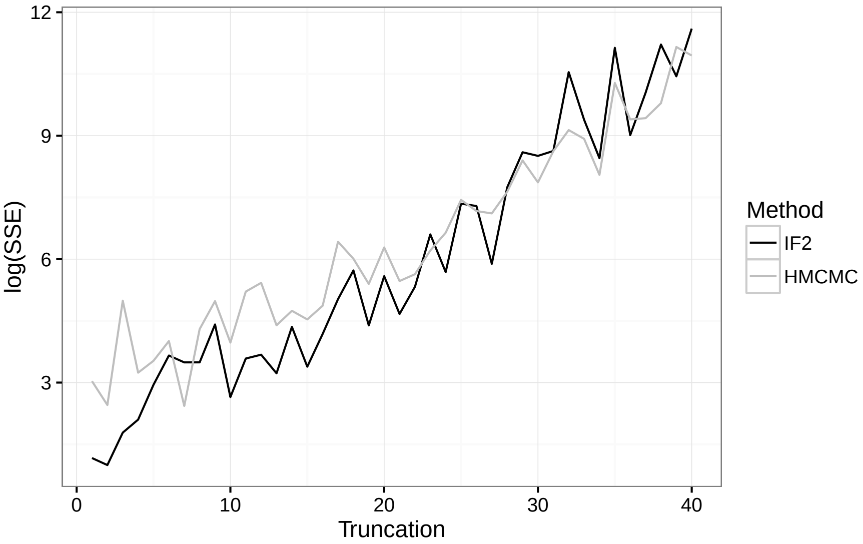

Figure 5.5: Error growth as a function of data truncation amount. Both methods used 200 bootstrap trajectories. Note that the y-axis shows the natural log of the averaged SSE, not the total SSE.

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 $\theta$, 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 [6.1].

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

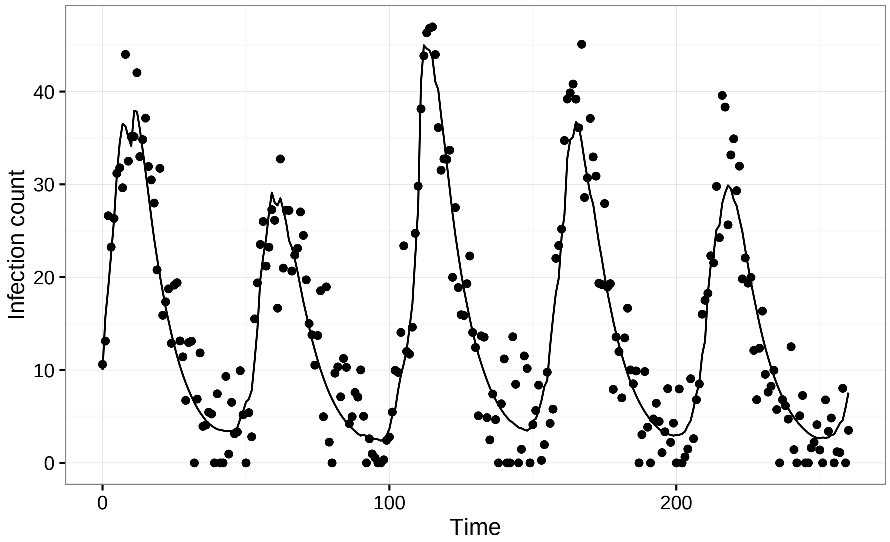

Figure 6.1: Five cycles generated by the SIRS function. The solid line the the true number of cases, dots show case counts with added observation noise. The Parameter values were $R0 = 3.0$, $\gamma = 0.1$, $\eta = 1$, $\sigma = 5$, and 10 initial cases.

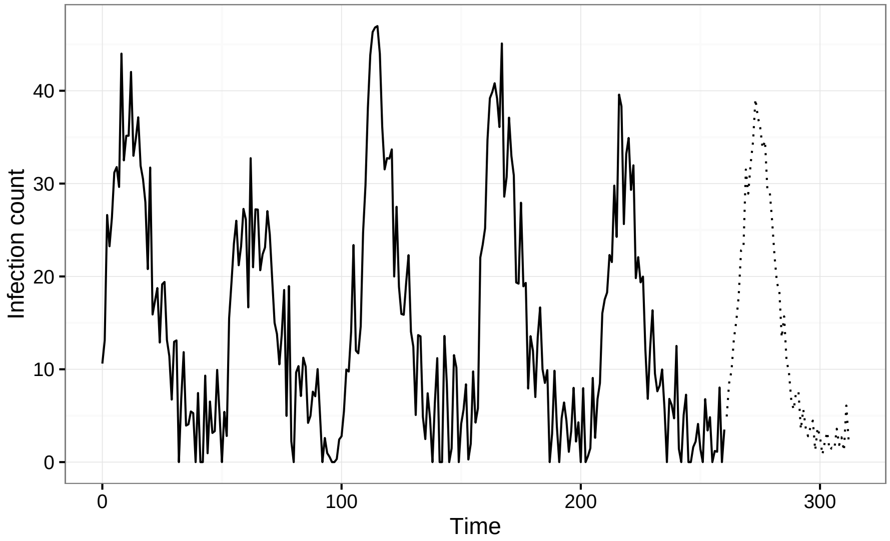

Figure 6.2: S-map applied to the data from the previous figure. The solid line shows the infection counts with observation noise form the previous plot, and the dotted line is the S-map forecast. Parameters chosen were $E = 14$ and $\theta = 3$.

The parameters used in the S-map algorithm to obtain the forecast used in Figure [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 [6.3].

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.

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 [6.4] shows the running times over 20 simulations.

It is clear from Figure [4.12] 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.

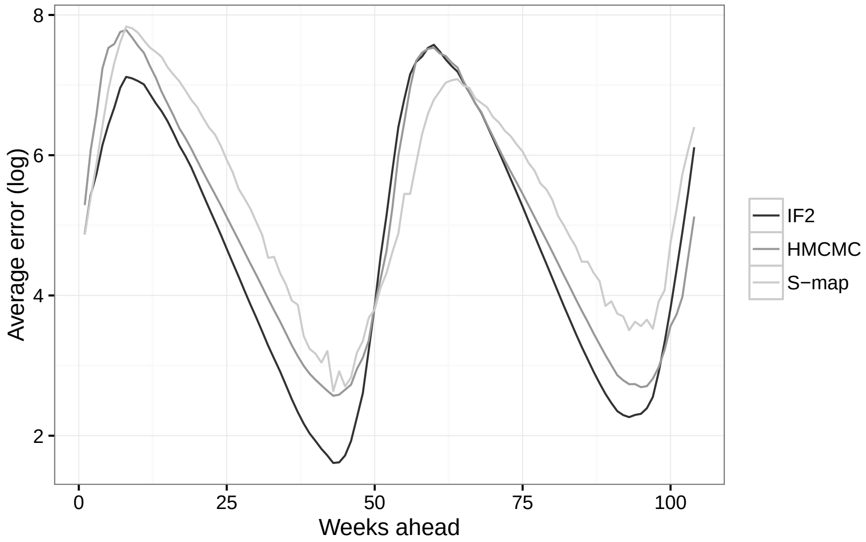

Figure 6.3: Error as a function of forecast length.

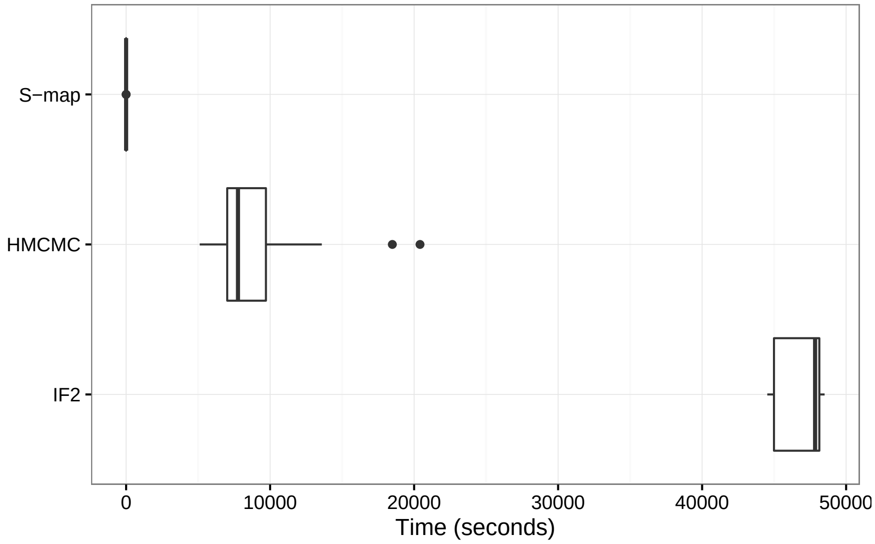

Figure 6.4: Runtimes for producing SIRS forecasts. The box shows the middle 50th quantile, the bold line is the median, and the dots are outliers.

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 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.

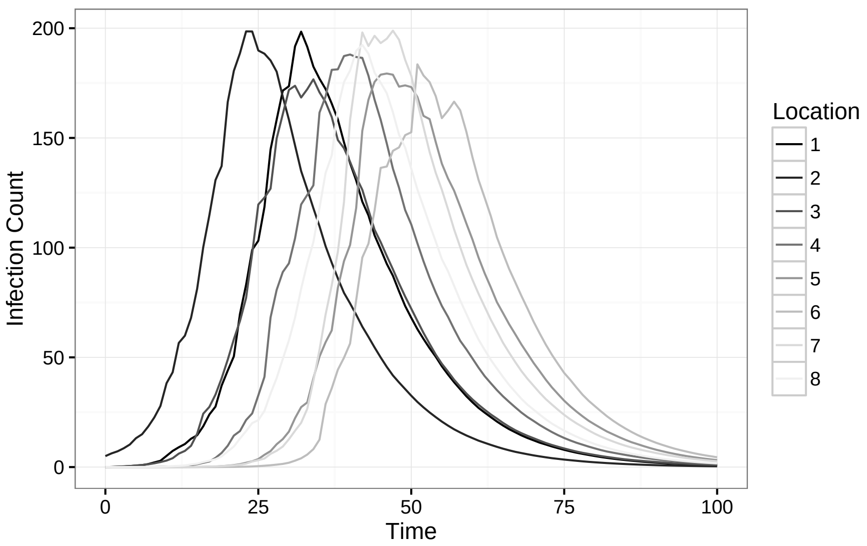

Figure 7.1: Evolution of a spatial epidemic in a ring topology. The outbreak was started with 5 cases in Location 2. Parameters were $R_0 = 3.0$, $\gamma = 0.1$, $\eta = 0.5$, $\sigma_{err} = 0.5$, and $\phi = 0.5$.

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].

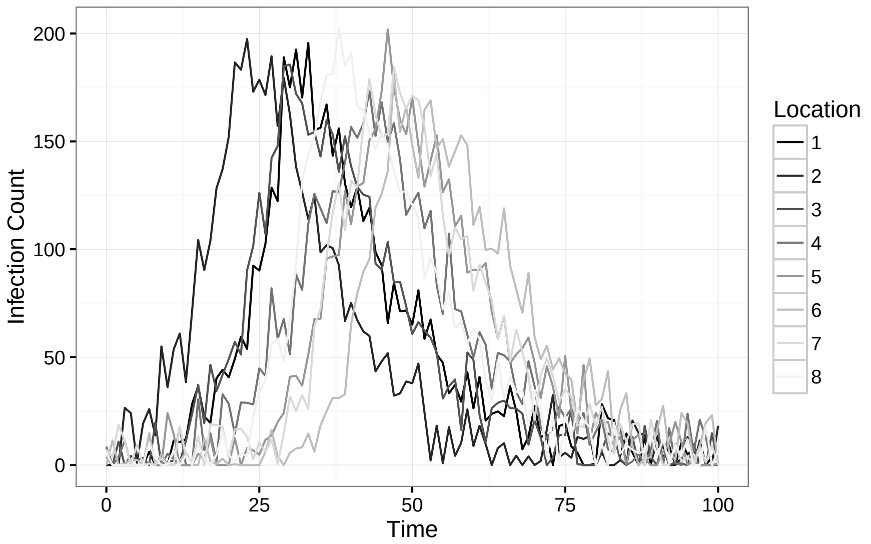

Figure 7.2: Evolution of a spatial epidemic as in Figure [7.1], with added observation noise drawn from $\mathcal{N}(0, 10)$.

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 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 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.

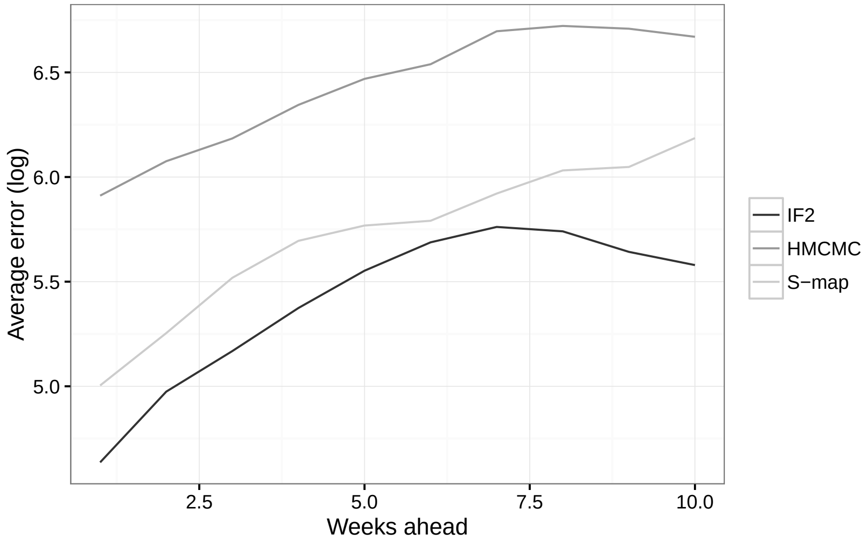

Figure 7.3: Average SSE (log scale) across each location and all trials as a function of the number of weeks ahead in the forecast.

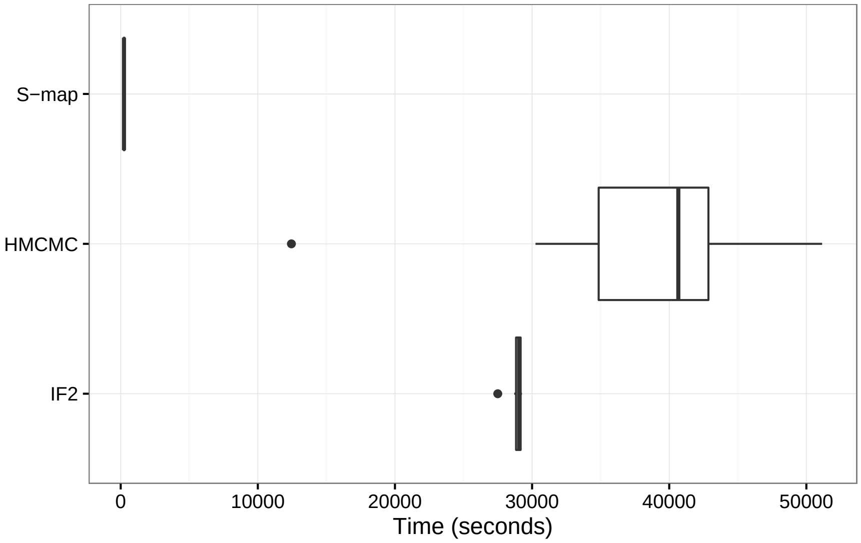

Figure 7.4: Runtimes for producing spatial SIR forecasts. The box shows the middle 50th quantile, the bold line is the median, and the dots are outliers.

Chapter 8

Discussion and Future Directions

8.1 Parallel and Distributed Computing

Whenever running times are discussed, we must consider the current computing landscape and hardware boundaries. In 1965, Intel co-founder Gordon E. Moore published a paper in which he observed that the number of transistors per unit area in integrated circuits double roughly every year. The consequence of this growth is the approximate year-over-year doubling of clock speeds (maximum number of sequential calculations performed per second), equivalent to raw performance of the chip. This forecast was updated in 1975 to double every 2 years and has held steady until the very recent past (Nature ref.).

Recently, transistor growth has begin to falter. This is due to several factors. The size of the transistors themselves has become so small that the next generation of processors would need to use transistors only 10-15 atoms across, at which point their ability to transport electrons becomes unreliable, and their behaviours will start to be affected by quantum uncertainty. Second, denser transistor packing would require aggressive cooling strategies as the Thermal Design Power (TDP), or the heat generated by such chips would increase dramatically.

To compensate for these limitations, chip manufacturers have instead redesigned the internal chip structures to consists to smaller "cores" within a single CPU die. The resulting processing power per processor then stays on track with Moore's Law, but keeps the clock speeds of each individual core, and consequently the thermal dissipation requirement, under control.

Of course this raises many problems on the software and algorithm side of computing.

Using several smaller cores instead of a single large has the distinct disadvantage of lack of cohesion – the cores must execute instructions completely decoupled from each other. This means algorithms have to be redesigned, or at least rewritten at the software level to consists of multiple independent pieces that can be run in parallel. This practice is known as parallelization.

Some compilers can actually detect areas in source code that contain obvious room for parallel execution (for example loop iterations with no dependence), and automatically generate machine code that can run on a multiprocessor with little to no performance overhead. This technology is still nascent and cannot be relied to operate successfully on anything but the most basic algorithms, and so usually we musts identify areas for parallelization and take advantage of them or risk not utilizing the full power of our machines. Further, high-performance computing essentially requires parallelization in its current form as large clusters and supercomputers rely on distributed computing "nodes".

When working with computationally intensive algorithms, particularly iterative methods such those used in this paper, the question of parallelism naturally arises. It may come as no surprise that the potential degrees of parallelism varies between methods.

Hamiltonian MCMC is cursed with high dependence between iterations. While HMCMC has an advantage over "vanilla" MCMC formulations in terms of efficiency of step acceptance and ease of exploration of the parameter per number of samples, each sample still depends entirely on the preceding one, and at a conceptual level the construction of a Markov Chain *requires* iterative dependence. We cannot simply take an accepted step, compute several proposed steps accept/reject them independently – doing so would break the chain construction and could potentially bias our posterior estimate to boot. We can, however, process multiple chains simultaneously and merge the resulting samples. If the required number of samples for a problem were large and the required burn-in time were low, this methods could prove effective. However, the parallel burn-in sampling is still inefficient as it is a duplication of effort with limited pay-off – in the sense that the saved sample to discarded burn-in sample ratio would not be as efficient as running a single long chain. Thus while parallelism via multiple independent chains would help with a reduction in wall clock running times, it would result in an *increase* in total computer time.

With regards to the bootstrapping process we used here, it should be clear that each bootstrap trajectory is completely independent, and thus this component of the forecasting framework can be considered "embarrassingly" parallel. Unfortunately, however, this is the least computationally demanding part of the process by several orders of magnitude, and so working to parallelize it would provide little advantage.

In the case of IF2, we have a decidedly different picture. In IF2 we have 5 primary steps in each data point integration:

- Forward evolution of the particles' internal system state using their parameter state
- Weighting those state estimates against the data point using the observation function
- Particle weight normalizations
- Resampling from the particle weight distribution
- Particle parameter perturbations

Luckily, 4 of the 5 steps can be individually parallelized and run on a per-particle basis. The particle weight normalizations, however, cannot. Summation "reductions" are a well-known problem for parallel algorithms; they can be parallelized to a degree using binary reduction, but that only reduces the approximate running time from $\mathcal{O}(n)$ to $\mathcal{O}(\log(n))$. The normalization process requires the particles' weight sum to be determined, hence the unavoidable obstacle of summation reductions rears its head. However this is in practice a less-taxing step, and its more demanding siblings are more amenable to parallelization.

Further, the full parametric bootstrapping process is incredibly computationally demanding, and also completely parallelizable. Each trajectory requires a fair bit of time to generate, on the order of of the original fitting time, and can be computed completely independently. Hence, IF2 is a very good candidate for a good parallel implementation.

A future offshoot of this project would be a good parallel implementation of both the IF2 fitting process and the parametric bootstrapping framework. And ideal platform for this work would be NVIDIA's Compute Unified Device Architecture (CUDA) Graphics Processing Unit (GPU) computing framework. While a CUDA implementation of a spatial epidemic IF2 parameter fitting algorithm was implemented, it lacked a good front-end implementation, R integration, and a parametric bootstrapping framework and so was not included in the main results of this paper. The code, however, as well as some preliminary results, are included in the appendices.

S-mapping, like the other two methods, is parallelizable to a degree. However, the S-map is already a great deal faster than the other two methods, and in the worst case (paired with Dewdrop Regression and applied to a spatiotemporal data set) still only takes a few minutes to run. Setting this observation aside, if one were investing in developing a faster S-map implementation, this is certainly possible. By far the most computationally expensive component of the algorithm is the SVD decomposition, and algorithms exist to accelerate it via parallelization. Further, each point in the forecast can be computed separately; in the cases similar to the one here with application to spatiotemporal prediction, there can be a significant number of these points.

Further work developing parallel implementations of forecasting frameworks could be advantageous if the goal was to generate accurate forecasts under more stringent time limitations. IF2 seems to have emerged as a leader in forecast accuracy, if not in efficient running times, and demonstrates high potential for parallelism. Expansion of the CUDA IF2 (`cuIF2`) implementation to include a parallel bootstrapping layer and R integration could prove very promising.

8.2 IF2, Bootstrapping, and Forecasting Methodology

The parametric bootstrapping approach used to generate additional parameter posterior samples and produce forecasts has proven effective, but not necessarily computationally efficient.

A recent paper utilising IF2 for forecasting [*King reference*] generated trajectories using IF2, parameter likelihood profiles, weighted quantiles, and the basic particle filter. The parameter profiles were used to construct a bounding box to search for good parameter sets, within which combinations of parameters to generate forecasts were selected using a Sobol sequence. Finally the forecasts were combined using a weighted quantile, taking into account the likelihood of the parameter sets used. Whether this approach would result in higher quality forecasts or lower running times is of interest, and could serve as a future research direction.

Expanding on this, there are other bootstrapping approaches that could be used to produce forecasts. A paper focusing solely on using IF2 with varied bootstrapping approaches and determining a forecast accuracy versus computational time trade-off curve of sorts would be useful.

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)))

    })

}
```

```

pars  ← c(R0  ← 3.0,    # average number of new infected individuals
     per infectious person
           r   ← 0.1,    # recovery rate
           N   ← 500)    # population size

T ← 100
y_ini ← c(S = 495, I = 5, R = 0)
times ← seq(0, T, by = 1)

odeout ← ode(y_ini, times, SIR, pars)

set.seed(1001)
sigma ← 10
infec_counts_raw ← odeout[,3] + rnorm(T+1, 0, sigma)
infec_counts     ← ifelse(infec_counts_raw < 0, 0, infec_counts_raw)

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

print(g)
ggsave(g, filename="dataplot.pdf", height=4, width=6.5)

sPw ← 7
datlen ← (T-1)*7 + 1

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

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

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

```

```
exfit ← extract(fit, permuted = TRUE, inc_warmup = FALSE)

R0points ← exfit$R0
R0kernel ← qplot(R0points, geom = "density", xlab = expression(R[0]), ylab = "frequency") +
        geom_vline(aes(xintercept=R0), linetype="dashed", size=1, color="grey50") +
        theme_bw()

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

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

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

sigmapoints ← exfit$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 ← exfit$y0[,2]
infeckernel ← qplot(infecpoints, geom = "density", xlab = "Initial Infected", ylab = "frequency") +
        geom_vline(aes(xintercept=y_ini[['I']]), linetype="dashed", size=1, color="grey50") +
        theme_bw()

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

exfit ← extract(fit, permuted = FALSE, inc_warmup = FALSE)
plotdata ← melt(exfit[,,"R0"])
tracefitR0 ← ggplot() +
              geom_line(data = plotdata,
                        aes(x = iterations,
                        y = value,
                        color = factor(chains, labels = 1:stan_options$chains))) +
              labs(x = "Sample", y = expression(R[0]), color = "Chain") +
              scale_color_brewer(palette="Greys") +
              theme_bw()

print(tracefitR0)
ggsave(tracefitR0, filename="traceplotR0.pdf", height=4, width=6.5)

exfit ← extract(fit, permuted = FALSE, inc_warmup = TRUE)
plotdata ← melt(exfit[,,"R0"])
tracefitR0 ← ggplot() +
              geom_line(data = plotdata,
                        aes(x = iterations,
                        y = value,
                        color = factor(chains, labels = 1:stan_options$chains))) +
              labs(x = "Sample", y = expression(R[0]), color = "Chain") +
              scale_color_brewer(palette="Greys") +
              theme_bw()

print(tracefitR0)
ggsave(tracefitR0, filename="traceplotR0_inc.pdf", height=4, width=6.5)

sso ← as.shinystan(fit)
sso ← launch_shinystan(sso)
```

A.2 Full Stan code

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

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

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=20>    sigma;  // observation error
    real <lower=0, upper=500>   y0[3];  // initial conditions

}

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);

    R0      ~ lognormal(1,1);
    r       ~ lognormal(1,1);
    sigma   ~ lognormal(1,1);

}
```

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)))

    })

}

T     ← 100
```

```
N      ← 500
sigma   ← 10
i_infec ← 5

## Generate true trajecory and synthetic data
##

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

true_pars ← c(R0 = 3.0,
              r = 0.1,
              N = 500.0)

odeout ← ode(true_init_cond, 0:T, SIR, true_pars)
trueTraj ← odeout[,3]

set.seed(1001)

infec_counts_raw ← odeout[,3] + rnorm(T+1, 0, sigma)
infec_counts     ← ifelse(infec_counts_raw < 0, 0, infec_counts_raw)

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

print(g)
ggsave(g, filename="dataplot.pdf", height=4, width=6.5)

## Rcpp stuff
##

sourceCpp(paste(getwd(),"d_if2.cpp",sep="/"))

paramdata ← data.frame(if2(infec_counts, T+1, N))
colnames(paramdata) ← c("R0", "r", "sigma", "Sinit", "Iinit", "Rinit")

## Parameter density kernels
##

R0points ← paramdata$R0
R0kernel ← qplot(R0points, geom = "density", xlab = expression(R[0]), ylab = "frequency") +
        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;
    double Sinitmean;   double Sinitsd;
    double Iinitmean;   double Iinitsd;
    double Rinitmean;   double Rinitsd;
};


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

// [[Rcpp::export]]
NumericMatrix if2(NumericVector data, int T, int N) {

    int     NP          = 2500;
    int     nPasses     = 50;
    double  coolrate    = 0.975;

    int     i_infec     = I0;

    NumericMatrix paramdata(NP, 6);

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

    double w[NP];           // particle weights

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

    printf("Initializing particle states\n");

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

        double R0can, rcan, sigmacan, Iinitcan;

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

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

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

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

    }

    // START PASSES THROUGH DATA

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


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

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

        perturbParticles(particles, N, NP, pass, coolrate);

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

            particles[n].S = particles[n].Sinit;
            particles[n].I = particles[n].Iinit;
            particles[n].R = particles[n].Rinit;

        }

        // between-pass perturbations

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

            // between-iteration perturbations
            perturbParticles(particles, N, NP, pass, coolrate);

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

                exp_euler_SIR(1.0/10.0, 0.0, 1.0, N, &particles[n]);

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

                w[n] = 1.0/(merr_par*sqrt(2.0*M_PI)) * exp( - y_diff
                   *y_diff / (2.0*merr_par*merr_par) );

            }

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

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

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

                double w_r = randu() * w[NP-1];
                int i = 0;
                while (w_r > w[i]) {
                    i++;
                }

                // i is now the index to copy state from
                copyParticle(&particles[n], &particles_old[i]);

            }

        }

    }

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

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

    printf("\n");



    // Get particle results to pass back to R

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

        paramdata(n, 0) = particles[n].R0;
        paramdata(n, 1) = particles[n].r;
        paramdata(n, 2) = particles[n].sigma;
        paramdata(n, 3) = particles[n].Sinit;
        paramdata(n, 4) = particles[n].Iinit;
        paramdata(n, 5) = particles[n].Rinit;

    }

    return paramdata;

}


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

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

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

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

    double R0   = particle->R0;
    double r    = particle->r;
    double B    = R0 * r / N;

    for(int i = 0; i < num_steps; i++) {
        // get derivatives
        double dS = - B*S*I;
        double dI = B*S*I - r*I;
        double dR = r*I;
        // step forward by h
        S += h*dS;
        I += h*dI;
        R += h*dR;
    }

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

}


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

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

    double coolcoef = pow(coolrate, passnum);

    double spreadR0     = coolcoef * R0true / 10.0;
    double spreadr      = coolcoef * rtrue  / 10.0;
    double spreadsigma  = coolcoef * merr   / 10.0;
    double spreadIinit  = coolcoef * I0     / 10.0;

    double R0can, rcan, sigmacan, Iinitcan;

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

        do {
            R0can = particles[n].R0 + spreadR0*randn();
        } while (R0can < 0);
        particles[n].R0 = R0can;

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

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

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

    }

}


/*  Convinience function for particle resampling process

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

    dst->R0     = src->R0;
    dst->r      = src->r;
    dst->sigma  = src->sigma;
    dst->S      = src->S;
    dst->I      = src->I;
    dst->R      = src->R;
    dst->Sinit  = src->Sinit;
    dst->Iinit  = src->Iinit;
    dst->Rinit  = src->Rinit;

}


/*  Checks to see if particles are collapsed
    This is done by checking if the standard deviations between the
       particles' parameter
    values are significantly close to one another. Spread threshold
       may need to be tuned.

    */
bool isCollapsed(Particle * particles, int NP) {

    bool retVal;

    double R0mean = 0, rmean = 0, sigmamean = 0, Sinitmean = 0,
       Iinitmean = 0, Rinitmean = 0;

    // means

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

        R0mean      += particles[n].R0;
        rmean       += particles[n].r;
        sigmamean   += particles[n].sigma;
        Sinitmean   += particles[n].Sinit;
        Iinitmean   += particles[n].Iinit;
        Rinitmean   += particles[n].Rinit;

    }

    R0mean      /= NP;
    rmean       /= NP;
    sigmamean   /= NP;
    Sinitmean   /= NP;
    Iinitmean   /= NP;
    Rinitmean   /= NP;

    double  R0sd = 0, rsd = 0, sigmasd = 0, Sinitsd = 0, Iinitsd = 0, Rinitsd = 0;

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

        R0sd    += ( particles[n].R0 - R0mean ) * ( particles[n].R0 - R0mean );
        rsd     += ( particles[n].r - rmean ) * ( particles[n].r - rmean );
        sigmasd += ( particles[n].sigma - sigmamean ) * ( particles[n].sigma - sigmamean );
        Sinitsd += ( particles[n].Sinit - Sinitmean ) * ( particles[n].Sinit - Sinitmean );
        Iinitsd += ( particles[n].Iinit - Iinitmean ) * ( particles[n].Iinit - Iinitmean );
        Rinitsd += ( particles[n].Rinit - Rinitmean ) * ( particles[n].Rinit - Rinitmean );

    }

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

    if ( (R0sd + rsd + sigmasd) < 1e-5)
        retVal = true;
    else
        retVal = false;

    return retVal;

}

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

    double  R0mean      = 0.0,
            rmean       = 0.0,
            sigmamean   = 0.0,
            Sinitmean   = 0.0,
            Iinitmean   = 0.0,
            Rinitmean   = 0.0;

    // means

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

        R0mean      += particles[n].R0;
        rmean       += particles[n].r;
        sigmamean   += particles[n].sigma;
        Sinitmean   += particles[n].Sinit;
        Iinitmean   += particles[n].Iinit;
        Rinitmean   += particles[n].Rinit;

    }

    R0mean      /= NP;
    rmean       /= NP;
    sigmamean   /= NP;
    Sinitmean   /= NP;
    Iinitmean   /= NP;
    Rinitmean   /= NP;

    // standard deviations

    double  R0sd    = 0.0,
            rsd     = 0.0,
            sigmasd = 0.0,
            Sinitsd = 0.0,
            Iinitsd = 0.0,
            Rinitsd = 0.0;

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

        R0sd    += ( particles[n].R0 - R0mean ) * ( particles[n].R0 - R0mean );
        rsd     += ( particles[n].r - rmean ) * ( particles[n].r - rmean );
        sigmasd += ( particles[n].sigma - sigmamean ) * ( particles[n].sigma - sigmamean );
        Sinitsd += ( particles[n].Sinit - Sinitmean ) * ( particles[n].Sinit - Sinitmean );
        Iinitsd += ( particles[n].Iinit - Iinitmean ) * ( particles[n].Iinit - Iinitmean );
        Rinitsd += ( particles[n].Rinit - Rinitmean ) * ( particles[n].Rinit - Rinitmean );

    }

    R0sd        /= NP;
    rsd         /= NP;
    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;

    return y1;

}
```

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$paramdata )
  names(paramdata_parent) ← c("R0", "r", "sigma", "eta", "berr", "
     Sinit", "Iinit", "Rinit")
  parmeans_parent ← colMeans(paramdata_parent)
  names(parmeans_parent) ← c("R0", "r", "sigma", "eta", "berr", "
     Sinit", "Iinit", "Rinit")
  # ...states
  statedata_parent ← data.frame( if2data_parent$statedata )
  names(statedata_parent) ← c("S","I","R","B")
```

```
statemeans_parent ← colMeans(statedata_parent)
names(statemeans_parent) ← c("S","I","R","B")


## use parametric bootstrapping to generate forcasts
##
trajectories ← foreach( i = 1:nTrials, .combine = rbind, .packages = "Rcpp") %dopar% {

  source(stoc_sir_file)

  ## draw new data
  ##

  pars ← with( as.list(parmeans_parent),
                c(R0 = R0,
                r = r,
                N = N,
                eta = eta,
                berr = berr) )

  init_cond ← with( as.list(parmeans_parent),
                     c(S = Sinit,
                       I = Iinit,
                       R = Rinit) )

  # generate trajectory
  sdeout ← StocSIR(init_cond, pars, Tlim + 1, steps)
  colnames(sdeout) ← c('S','I','R','B')

  # add noise
  counts_raw ← sdeout[,'I'] + rnorm(dim(sdeout)[1], 0, parmeans_parent[['sigma']])
    counts     ← ifelse(counts_raw < 0, 0, counts_raw)

    ## refit using new data
    ##

    rm(if2) # because stupid things get done in packages
    sourceCpp(if2file)
    if2time ← system.time( if2data ← if2(counts, Tlim+1, N, NP, nPasses, coolrate) )

    paramdata ← data.frame( if2data$paramdata )
  names(paramdata) ← c("R0", "r", "sigma", "eta", "berr", "Sinit", "Iinit", "Rinit")
  parmeans ← colMeans(paramdata)
  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)

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

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

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

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

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

  }

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

}

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

E.2 SMAP Code

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

```
library(pracma)

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

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

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

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

    #allPredictees ← matrix(NA, stepsAhead, E)

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

        # set up weight calculation
        predmat ← repmat(predictee, liblen, 1)
        distances ← sqrt( rowSums( abs(lib - predmat)^2 ) )
        meanDist ← mean(distances)

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

        # construct A, B

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

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

        # solve system for C

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

        # get prediction

        predsum ← sum(C * c(1,predictee))

        # save

        predictions[i] ← predsum

        # next predictee

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

    }

    return(predictions)

}
```

E.3 SMAP Parameter Optimization Code

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

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

set.seed(1010)

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



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

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

```

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

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

ssemat ← matrix(NA, 20, 20)

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

    ssemean ← 0

    for (k in 1:nTrials) {

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

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

      ## perturb to get data
      ##
      infec_counts_raw ← sdeout[1:(Tlim+1),'I'] + rnorm(Tlim+1,0,sigma)
      infec_counts   ← ifelse(infec_counts_raw < 0, 0, infec_counts_raw)

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

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

      ssemean ← ssemean + (sse / nTrials)

    }

    ssemat[i,j] ← ssemean


  }
}

quartz()
image(-ssemat)
```

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

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

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

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

Emin ← Elist[mininds[,’row’]]
thetamin ← thetalist[mininds[,’col’]]

print(Emin)
print(thetamin)
```

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;
    double R;
};

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

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


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

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

    int npar = 9;

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

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

    double w[NP];           // particle weights

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

    printf("Initializing particle states\n");

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

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

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

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

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

        particles[n].B = (double) R0can * rcan / N;

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

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

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

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

    }

    // START PASSES THROUGH DATA

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


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

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

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

            particles[n].S = particles[n].Sinit;
            particles[n].I = particles[n].Iinit;
            particles[n].R = particles[n].Rinit;
            particles[n].B = (double) particles[n].R0 * particles[n
                ].r / N;

        }

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

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

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

                exp_euler_SIRS(1.0/7.0, (double) t-1, (double) t, N,
                     &particles[n]);

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

                w[n] = 1.0/(merr_par*sqrt(2.0*M_PI)) * exp( - y_diff
                    *y_diff / (2.0*merr_par*merr_par) );

            }

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

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

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

                double w_r = randu() * w[NP-1];
                int i = 0;
                while (w_r > w[i]) {
                    i++;
                }

                // i is now the index to copy state from
                copyParticle(&particles[n], &particles_old[i]);

            }

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

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

        }

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

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

        sds(pass, 0) = pInfo.R0sd;
        sds(pass, 1) = pInfo.rsd;
        sds(pass, 2) = pInfo.resd;
        sds(pass, 3) = pInfo.sigmasd;
        sds(pass, 4) = pInfo.etasd;
        sds(pass, 5) = pInfo.berrsd;
        sds(pass, 6) = pInfo.Sinitsd;
        sds(pass, 7) = pInfo.Iinitsd;
        sds(pass, 8) = pInfo.Rinitsd;

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

    }

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

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

    printf("\n");



    // Get particle results to pass back to R

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

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

    }

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

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

    }



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

}


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

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

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

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

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

    double B = particle->B;

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

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

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

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

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

    }

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

}


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

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

    //double coolcoef = exp( - (double) passnum / coolrate );
    double coolcoef = pow(coolrate, passnum);


    double spreadR0     = coolcoef * R0true / 10.0;
    double spreadr      = coolcoef * rtrue / 10.0;
    double spreadre     = coolcoef * retrue / 10.0;
    double spreadsigma  = coolcoef * merr / 10.0;
    double spreadIinit  = coolcoef * I0 / 10.0;
    double spreadeta    = coolcoef * etatrue / 10.0;
    double spreadberr   = coolcoef * berrtrue / 10.0;


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

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

        do {
            R0can = particles[n].R0 + spreadR0*randn();
        } while (R0can < 0);
        particles[n].R0 = R0can;

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

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

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

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

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

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

    }

```

```c
}


/*  Convinience function for particle resampling process

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

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

}

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

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

    // means

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

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

    }

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

    // standard deviations

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

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

        R0sd    += ( particles[n].R0 - R0mean ) * ( particles[n].R0
           - R0mean );
        rsd     += ( particles[n].r - rmean ) * ( particles[n].r -
           rmean );
        resd    += ( particles[n].re - rmean ) * ( particles[n].re -
            rmean );
        sigmasd += ( particles[n].sigma - sigmamean ) * ( particles[
           n].sigma - sigmamean );
        etasd   += ( particles[n].eta - etamean ) * ( particles[n].
           eta - etamean );
        berrsd  += ( particles[n].berr - berrmean ) * ( particles[n
           ].berr - berrmean );
        Sinitsd += ( particles[n].Sinit - Sinitmean ) * ( particles[
           n].Sinit - Sinitmean );
        Iinitsd += ( particles[n].Iinit - Iinitmean ) * ( particles[
           n].Iinit - Iinitmean );
        Rinitsd += ( particles[n].Rinit - Rinitmean ) * ( particles[
           n].Rinit - Rinitmean );

    }

    R0sd        /= NP;
    rsd         /= NP;
    resd        /= NP;
    sigmasd     /= NP;
    etasd       /= NP;
    berrsd      /= NP;
    Sinitsd     /= NP;
    Iinitsd     /= NP;
    Rinitsd     /= NP;

    partInfo->R0mean    = R0mean;
    partInfo->R0sd      = R0sd;
    partInfo->rmean     = rmean;
    partInfo->rsd       = rsd;
    partInfo->remean    = remean;
    partInfo->resd      = resd;
    partInfo->sigmamean = sigmamean;
    partInfo->sigmasd   = sigmasd;
    partInfo->etamean   = etamean;
    partInfo->etasd     = etasd;
    partInfo->berrmean  = berrmean;
    partInfo->berrsd    = berrsd;
    partInfo->Sinitmean = Sinitmean;
    partInfo->Sinitsd   = Sinitsd;
    partInfo->Iinitmean = Iinitmean;
    partInfo->Iinitsd   = Iinitsd;
    partInfo->Rinitmean = Rinitmean;
    partInfo->Rinitsd   = Rinitsd;

}

double randu() {

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

}

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

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

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

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

}
```

```


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

    double x1, x2, w, y1;

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

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

    return y1;

}
```

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

}

model {

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

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

    B0 ← R0 * r / N;

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

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

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

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

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

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

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

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

        }
    }

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

}
```

F.3 IF2 Spatial SIR Code

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

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

    */

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

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

#define Treal       100         // time to simulate over
#define R0true      3.0         // infectiousness
#define rtrue       0.1         // recovery rate
```

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

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

#include <Rcpp.h>
using namespace Rcpp;

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


int timeval_subtract (double *result, struct timeval *x, struct timeval *y);
int check_double(double x,double y);
void initializeParticles(Particle ** particles, int NP, int nloc, int N);
void exp_euler_SSIR(double h, double t0, double tn, int N, Particle * particle,
                    NumericVector neinum, NumericMatrix neibmat, int nloc) ;
void copyParticle(Particle * dst, Particle * src, int nloc);
void perturbParticles(Particle * particles, int N, int NP, int nloc, int passnum, double coolrate);
double randu();
double randn();

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

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

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

    double w[NP];           // particle weights

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

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

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

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

    // START PASSES THROUGH DATA

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


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

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

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

        if (pass == (nPasses-1)) {
            double means[nloc];
            for (int loc = 0; loc < nloc; loc++) {
                means[loc] = 0.0;
                for (int n = 0; n < NP; n++) {
                    means[loc] += particles[n].I[loc] / NP;
                }
                infecmeans(loc, 0) = means[loc];
            }
        }

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

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

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

                double merr_par = particles[n].sigma;

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

            }

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

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

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

                double w_r = randu() * w[NP-1];
                int i = 0;
                while (w_r > w[i]) {
                    i++;
                }

                // i is now the index to copy state from
                copyParticle(&particles[n], &particles_old[i], nloc);

            }

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

            if (pass == (nPasses-1)) {
                double means[nloc];
                for (int loc = 0; loc < nloc; loc++) {
                    means[loc] = 0.0;
                    for (int n = 0; n < NP; n++) {
                        means[loc] += particles[n].I[loc] / NP;
                    }
                    infecmeans(loc, t) = means[loc];
                }
            }

        }

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

    }

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

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

    // Pack final state means data
    double Smeans[nloc], Imeans[nloc], Rmeans[nloc], Bmeans[nloc];
    for (int loc = 0; loc < nloc; loc++) {
        Smeans[loc] = 0.0;
        Imeans[loc] = 0.0;
        Rmeans[loc] = 0.0;
        Bmeans[loc] = 0.0;
        for (int n = 0; n < NP; n++) {
            Smeans[loc] += particles[n].S[loc] / NP;
            Imeans[loc] += particles[n].I[loc] / NP;
            Rmeans[loc] += particles[n].R[loc] / NP;
            Bmeans[loc] += particles[n].B[loc] / NP;
        }
        finalstate(loc, 0) = Smeans[loc];
        finalstate(loc, 1) = Imeans[loc];
        finalstate(loc, 2) = Rmeans[loc];
        finalstate(loc, 3) = Bmeans[loc];
    }


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



}

```

```

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

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

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

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

    // create last state vectors
    double S_last[nloc];
    double I_last[nloc];
    double R_last[nloc];
    double B_last[nloc];

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

    //printf(”sphi \t\t| ophi \t\t| BSI \t\t| rI \t\t| dS \t\t| dI \t\t| dR \t\t| S \t\t| I \t\t| R |\n”);

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

        for (int loc = 0; loc < nloc; loc++) {
            S_last[loc] = S[loc];
            I_last[loc] = I[loc];
            R_last[loc] = R[loc];
            B_last[loc] = B[loc];
        }

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

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

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

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

            double BSI = S_last[loc]*( sphi*B_last[loc]*I_last[loc]
               + ophi*nBIsum );
            double rI  = r*I_last[loc];

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

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

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

        }

    }

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

}

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

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

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

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

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

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

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

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

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

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

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

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

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

    }

}

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

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

    //double coolcoef = exp( - (double) passnum / coolrate );
    double coolcoef = pow(coolrate, passnum);

    double spreadR0     = coolcoef * R0true / 10.0;
    double spreadr      = coolcoef * rtrue / 10.0;
    double spreadsigma  = coolcoef * merr / 10.0;
    double spreadIinit  = coolcoef * I0 / 10.0;
    double spreadeta    = coolcoef * etatrue / 10.0;
    double spreadberr   = coolcoef * berrtrue / 10.0;
    double spreadphi    = coolcoef * phitrue / 10.0;

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

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

        do {
            R0can = particles[n].R0 + spreadR0*randn();
        } while (R0can < 0);
        particles[n].R0 = R0can;

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

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

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

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

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

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

}

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

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

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

}



double randu() {

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

}
```

```

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

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

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

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

}
*/

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

    double x1, x2, w, y1;

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

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

    return y1;

}
```

F.4 CUDA IF2 Spatial Fitting Code

Below is the nascent CUDA code that will be expanded upon in future work.

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

/*  Runs a particle filter on synthetic noisy data and attempts to
    reconstruct underlying true state at each time step. Note that
    this program uses gnuplot to plot the data, so an x11
    environment must be present. Also the multiplier of 1024 in the
    definition of NP below should be set to a multiple of the number
    of multiprocessors of your GPU for optimal results.

    Also, the accompanying "pf.plg" file contains the instructions
    gnuplot will use. It must be present in the same directory as
    the executable generated by compiling this file.

    Compile with:

    nvcc -arch=sm_20 -O2 pf_cuda.cu timer.cpp rand.cpp -o pf_cuda.x

    */

#include <cuda.h>
#include <iostream>
#include <fstream>
#include <curand.h>
#include <curand_kernel.h>
#include <string>
#include <sstream>
#include <cmath>

#include "timer.h"
#include "rand.h"
#include "readdata.h"

#define NP          (2*2500)    // number of particles
#define N           500.0       // population size
#define R0true      3.0         // infectiousness
#define rtrue       0.1         // recovery rate
#define etatrue     0.5         // real drift attraction strength
#define berrtrue    0.5         // real beta drift noise
#define phitrue     0.5         // real connectivity strength
#define merr        10.0        // expected measurement error
#define I0          5.0         // Initial infected individuals
#define PSC         0.5         // sensitive parameter perturbation scaling
#define NLOC        10

#define PI      3.141592654f

// Wrapper for CUDA calls, from CUDA API
// Modified to also print the error code and string
# define CUDA_CALL(x) do { if ((x) != cudaSuccess ) {
        \
    std::cout << " Error at " << __FILE__ << ":" << __LINE__ << std::endl;      \
    std::cout << " Error was " << x << " " << cudaGetErrorString(x) << std::endl;   \
    return EXIT_FAILURE ;}} while (0)          \

typedef struct {
    float R0;
    float r;
    float sigma;
    float eta;
    float berr;
    float phi;
    /*
    float * S;
    float * I;
    float * R;
    float * B;
    float * Iinit;
    */
    float S[NLOC];
    float I[NLOC];
    float R[NLOC];
    float B[NLOC];
    float Iinit[NLOC];
    curandState randState;  // PRNG state
} Particle;

__host__ std::string getHRmemsize (size_t memsize);
__host__ std::string getHRtime (float runtime);

__device__ void exp_euler_SSIR(float h, float t0, float tn, Particle * particle, int * neinum, int * neibmat, int nloc);
__device__ void copyParticle(Particle * dst, Particle * src, int nloc);


/*  Initialize all PRNG states, get starting state vector using initial distribution
    */
__global__ void initializeParticles (Particle * particles, int nloc) {

    int id  = blockIdx.x*blockDim.x + threadIdx.x;  // global thread ID

    if (id < NP) {

        // initialize PRNG state
        curandState state;
        curand_init(id, 0, 0, &state);

        // allocate space for arays inside particle
        //particles[id].S = (float*) malloc(nloc*sizeof(float));
        //particles[id].I = (float*) malloc(nloc*sizeof(float));
        //particles[id].R = (float*) malloc(nloc*sizeof(float));
        //particles[id].B = (float*) malloc(nloc*sizeof(float));
        //particles[id].Iinit = (float*) malloc(nloc*sizeof(float));

        // initialize all parameters

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

        do {
            R0can = R0true + R0true*curand_normal(&state);
        } while (R0can < 0);
        particles[id].R0 = R0can;

        do {
            rcan = rtrue + rtrue*curand_normal(&state);
        } while (rcan < 0);
        particles[id].r = rcan;

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

        do {
            sigmacan = merr + merr*curand_normal(&state);
        } while (sigmacan < 0);
        particles[id].sigma = sigmacan;

        do {
            etacan = etatrue + PSC*etatrue*curand_normal(&state);
        } while (etacan < 0 || etacan > 1);
        particles[id].eta = etacan;

        do {
            berrcan = berrtrue + PSC*berrtrue*curand_normal(&state);
        } while (berrcan < 0);
        particles[id].berr = berrcan;

        do {
            phican = phitrue + PSC*phitrue*curand_normal(&state);
        } while (phican <= 0 || phican >= 1);
        particles[id].phi = phican;

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

        particles[id].randState = state;

    }

}

__global__ void resetStates (Particle * particles, int nloc) {

    int id  = blockIdx.x*blockDim.x + threadIdx.x;  // global thread ID

    for (int loc = 0; loc < nloc; loc++) {
        particles[id].S[loc] = N - particles[id].Iinit[loc];
        particles[id].I[loc] = particles[id].Iinit[loc];
        particles[id].R[loc] = 0.0;
    }

}

__global__ void clobberParams (Particle * particles, int nloc) {

    int id  = blockIdx.x*blockDim.x + threadIdx.x;  // global thread ID

    particles[id].R0 = R0true;
    particles[id].r = rtrue;
    particles[id].sigma = merr;
    particles[id].eta = etatrue;
    particles[id].berr = berrtrue;
    particles[id].phi = phitrue;

    for (int loc = 0; loc < nloc; loc++) {
        particles[id].Iinit[loc] = I0;
    }


}


/*  Project particles forward, perturb, and save weight based on data
    int t - time step number (1,...,T)
    */
__global__ void project (Particle * particles, int * neinum, int *
   neibmat, int nloc) {

    int id = blockIdx.x*blockDim.x + threadIdx.x;   // global id

    if (id < NP) {
        // project forward
        exp_euler_SSIR(1.0/7.0, 0.0, 1.0, &particles[id], neinum,
           neibmat, nloc);
    }

}

__global__ void weight(float * data, Particle * particles, double *
   w, int t, int T, int nloc) {

    int id = blockIdx.x*blockDim.x + threadIdx.x;   // global id

    if (id < NP) {

        float merr_par = particles[id].sigma;

        // Get weight and save
        double w_local = 1.0;
        for (int loc = 0; loc < nloc; loc++) {
            float y_diff = data[loc*T + t] - particles[id].I[loc];
            w_local *= 1.0/(merr_par*sqrt(2.0*PI)) * exp( - y_diff*
               y_diff / (2.0*merr_par*merr_par) );
        }

        w[id] = w_local;

    }

}

__global__ void stashParticles (Particle * particles, Particle *
   particles_old, int nloc) {

    int id = blockIdx.x*blockDim.x + threadIdx.x;   // global id

    if (id < NP) {
        // COPY PARTICLE
        copyParticle(&particles_old[id], &particles[id], nloc);
    }

}


/*  The 0th thread will perform cumulative sum on the weights.
    There may be a faster way to do this, will investigate.
    */
__global__ void cumsumWeights (double * w) {

    int id  = blockIdx.x*blockDim.x + threadIdx.x;  // global thread ID

    // compute cumulative weights
    if (id == 0) {
        for (int i = 1; i < NP; i++)
            w[i] += w[i-1];
    }

}


/*  Resample from all particle states within cell
    */
__global__ void resample (Particle * particles, Particle * particles_old, double * w, int nloc) {

    int id  = blockIdx.x*blockDim.x + threadIdx.x;

    if (id < NP) {

        // resampling proportional to weights
        double w_r = curand_uniform(&particles[id].randState) * w[NP-1];
        int i = 0;
        while (w_r > w[i]) {
            i++;
        }

        // i is now the index of the particle to copy from
        copyParticle(&particles[id], &particles_old[i], nloc);

    }

}

// launch this with probably just nloc threads... block structure/size probably not important
__global__ void reduceStates (Particle * particles, float * countmeans, int t, int T, int nloc) {

    int id  = blockIdx.x*blockDim.x + threadIdx.x;

    if (id < nloc) {

        int loc = id;

        double countmean_local = 0.0;
        for (int n = 0; n < NP; n++) {
            countmean_local += particles[n].I[loc] / NP;
        }

        countmeans[loc*T + t] = (float) countmean_local;

    }

}

__global__ void perturbParticles(Particle * particles, int nloc, int passnum, double coolrate) {

    //double coolcoef = exp( - (double) passnum / coolrate );
    double coolcoef = pow(coolrate, passnum);

    double spreadR0     = coolcoef * R0true / 10.0;
    double spreadr      = coolcoef * rtrue / 10.0;
    double spreadsigma  = coolcoef * merr / 10.0;
    double spreadIinit  = coolcoef * I0 / 10.0;
    double spreadeta    = coolcoef * etatrue / 10.0;
    double spreadberr   = coolcoef * berrtrue / 10.0;
    double spreadphi    = coolcoef * phitrue / 10.0;

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

    int id  = blockIdx.x*blockDim.x + threadIdx.x;

    if (id < NP) {

        do {
            R0can = particles[id].R0 + spreadR0*curand_normal(&particles[id].randState);
        } while (R0can < 0);
        particles[id].R0 = R0can;

        do {
            rcan = particles[id].r + spreadr*curand_normal(&particles[id].randState);
        } while (rcan < 0);
        particles[id].r = rcan;

        do {
            sigmacan = particles[id].sigma + spreadsigma*curand_normal(&particles[id].randState);
        } while (sigmacan < 0);
        particles[id].sigma = sigmacan;

        do {
            etacan = particles[id].eta + PSC*spreadeta*curand_normal
               (&particles[id].randState);
        } while (etacan < 0 || etacan > 1);
        particles[id].eta = etacan;

        do {
            berrcan = particles[id].berr + PSC*spreadberr*
               curand_normal(&particles[id].randState);
        } while (berrcan < 0);
        particles[id].berr = berrcan;

        do {
            phican = particles[id].phi + PSC*spreadphi*curand_normal
               (&particles[id].randState);
        } while (phican <= 0 || phican >= 1);
        particles[id].phi = phican;

        for (int loc = 0; loc < nloc; loc++) {
            do {
                Iinitcan = particles[id].Iinit[loc] + spreadIinit*
                   curand_normal(&particles[id].randState);
            } while (Iinitcan < 0 || Iinitcan > 500);
            particles[id].Iinit[loc] = Iinitcan;
        }

    }

}


int main (int argc, char *argv[]) {


    int T, nloc;

    double restime;
    struct timeval tdr0, tdr1, tdrMaster;

    gettimeofday (&tdr0, NULL);


    // Parse arguments
       **********************************************

    if (argc < 4) {
        std::cout << "Not enough arguments" << std::endl;
        return 0;
    }

    std::string arg1(argv[1]);  // infection counts
    std::string arg2(argv[2]);  // neighbour counts
    std::string arg3(argv[3]);  // neighbour indices

    std::cout << "Arguments:" << std::endl;
    std::cout << "Infection data:    " << arg1 << std::endl;
    std::cout << "Neighbour counts:  " << arg2 << std::endl;
    std::cout << "Neighbour indices: " << arg3 << std::endl;

    //
       ***************************************************************



    // Read count data
       ***********************************************

    std::cout << "Getting count data" << std::endl;
    float * data = getDataFloat(arg1, &T, &nloc);
    size_t datasize = nloc*T*sizeof(float);

    //
       ***************************************************************



    // Read neinum matrix data
       ***************************************

    std::cout << "Getting neighbour count data" << std::endl;
    int * neinum = getDataInt(arg2, NULL, NULL);
    size_t neinumsize = nloc * sizeof(int);

    //
       ***************************************************************



    // Read neibmat matrix data
       **************************************

    std::cout << "Getting neighbour count data" << std::endl;
    int * neibmat = getDataInt(arg3, NULL, NULL);
    size_t neibmatsize = nloc * nloc * sizeof(int);

    //
       ***************************************************************



    gettimeofday (&tdr1, NULL);
    timeval_subtract (&restime, &tdr1, &tdr0);

    std::cout << "\t" << getHRtime(restime) << std::endl;

    // ****************************************************************************

    // CUDA data ***************************************************

    std::cout << "Allocating device storage" << std::endl;

    gettimeofday (&tdr0, NULL);

    float       * d_data;           // device copy of data
    Particle    * particles;        // particles
    Particle    * particles_old;    // intermediate particle states
    double      * w;                // weights
    int         * d_neinum;         // device copy of adjacency matrix
    int         * d_neibmat;        // device copy of neighbour counts matrix
    float       * countmeans;       // host copy of reduced infection count means from last pass
    float       * d_countmeans;     // device copy of reduced infection count means from last pass

    CUDA_CALL( cudaMalloc( (void**) &d_data         , datasize )            );
    CUDA_CALL( cudaMalloc( (void**) &particles      , NP*sizeof(Particle))  );
    CUDA_CALL( cudaMalloc( (void**) &particles_old  , NP*sizeof(Particle))  );
    CUDA_CALL( cudaMalloc( (void**) &w              , NP*sizeof(double))    );
    CUDA_CALL( cudaMalloc( (void**) &d_neinum       , neinumsize)           );
    CUDA_CALL( cudaMalloc( (void**) &d_neibmat      , neibmatsize)          );
    CUDA_CALL( cudaMalloc( (void**) &d_countmeans   , nloc*T*sizeof(float)) );


    gettimeofday (&tdr1, NULL);
    timeval_subtract (&restime, &tdr1, &tdr0);

    std::cout << "\t" << getHRtime(restime) << std::endl;

    size_t avail, total;
    cudaMemGetInfo( &avail, &total );
    size_t used = total - avail;

    std::cout << "\t[" << getHRmemsize(used) << "] used of [" <<
        getHRmemsize(total) << "]" <<std::endl;

    std::cout << "Copying data to device" << std::endl;

    gettimeofday (&tdr0, NULL);

    CUDA_CALL( cudaMemcpy(d_data    , data      , datasize      ,
        cudaMemcpyHostToDevice)   );
    CUDA_CALL( cudaMemcpy(d_neinum  , neinum    , neinumsize    ,
        cudaMemcpyHostToDevice)   );
    CUDA_CALL( cudaMemcpy(d_neibmat , neibmat   , neibmatsize   ,
        cudaMemcpyHostToDevice)   );

    gettimeofday (&tdr1, NULL);
    timeval_subtract (&restime, &tdr1, &tdr0);

    std::cout << "\t" << getHRtime(restime) << std::endl;

    //
        **************************************************************




    // Initialize particles
        *****************************************

    std::cout << "Initializing particles" << std::endl;

    gettimeofday (&tdr0, NULL);

    int nThreads    = 32;
    int nBlocks     = ceil( (float) NP / nThreads);

    initializeParticles <<< nBlocks, nThreads >>> (particles, nloc);
    CUDA_CALL( cudaGetLastError() );
    CUDA_CALL( cudaDeviceSynchronize() );

    initializeParticles <<< nBlocks, nThreads >>> (particles_old,
        nloc);
    CUDA_CALL( cudaGetLastError() );
    CUDA_CALL( cudaDeviceSynchronize() );

    gettimeofday (&tdr1, NULL);
    timeval_subtract (&restime, &tdr1, &tdr0);

    std::cout << "\t" << getHRtime(restime) << std::endl;

    cudaMemGetInfo( &avail, &total );
    used = total - avail;
    std::cout << "\t[" << getHRmemsize(used) << "] used of [" << getHRmemsize(total) << "]" <<std::endl;

    // **************************************************************

    // Starting filtering *******************************************

    for (int pass = 0; pass < 50; pass++) {

        std::cout << "pass = " << pass << std::endl;

        // ** TEMP **
        //clobberParams <<< nBlocks, nThreads >>> (particles, nloc);
        // ** TEMP **

        nThreads    = 32;
        nBlocks     = ceil( (float) NP / nThreads);

        resetStates <<< nBlocks, nThreads >>> (particles, nloc);
        CUDA_CALL( cudaGetLastError() );
        CUDA_CALL( cudaDeviceSynchronize() );

        std::cout << "Filtering over [1," << Tlim << "]"<< std::endl;

        gettimeofday (&tdrMaster, NULL);

        gettimeofday (&tdr0, NULL);

        nThreads = 1;
        nBlocks  = 10;

        if (pass == 49) {
            reduceStates <<< nBlocks, nThreads >>> (particles, d_countmeans, 0, T, nloc);
            CUDA_CALL( cudaGetLastError() );
            CUDA_CALL( cudaDeviceSynchronize() );
        }

        gettimeofday (&tdr1, NULL);
        timeval_subtract (&restime, &tdr1, &tdr0);
        std::cout << "Reduction        " << getHRtime(restime) << std::endl;

        int Tlim = T;

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

            // Projection ************************************************

            nThreads    = 32;
            nBlocks     = ceil( (float) NP / nThreads);

            //if (t == 1)
            //  gettimeofday (&tdr0, NULL);

            project <<< nBlocks, nThreads >>> (particles, d_neinum, d_neibmat, nloc);
            CUDA_CALL( cudaGetLastError() );
            CUDA_CALL( cudaDeviceSynchronize() );

            //if (t == 1) {
            //  gettimeofday (&tdr1, NULL);
            //  timeval_subtract (&restime, &tdr1, &tdr0);
            //  std::cout << "\tProjection " << getHRtime(restime) << std::endl;
            //}

            // Weighting *************************************************

            nThreads    = 32;
            nBlocks     = ceil( (float) NP / nThreads);

            weight <<< nBlocks, nThreads >>>(d_data, particles, w, t, T, nloc);
            CUDA_CALL( cudaGetLastError() );
            CUDA_CALL( cudaDeviceSynchronize() );

            // Cumulative sum ********************************************

            nThreads    = 1;
            nBlocks     = 1;

            if (t == 1)
                gettimeofday (&tdr0, NULL);

            cumsumWeights <<< nBlocks, nThreads >>> (w);
            CUDA_CALL( cudaGetLastError() );
            CUDA_CALL( cudaDeviceSynchronize() );

            if (t == 1) {
                gettimeofday (&tdr1, NULL);
            timeval_subtract (&restime, &tdr1, &tdr0);
            std::cout << "Cumulative sum   " << getHRtime(restime) << std::endl;
        }

        // Save particles for resampling from *************************

        nThreads    = 32;
        nBlocks     = ceil( (float) NP / nThreads);

        stashParticles <<< nBlocks, nThreads >>> (particles, particles_old, nloc);
        CUDA_CALL( cudaGetLastError() );
        CUDA_CALL( cudaDeviceSynchronize() );


        // Resampling *************************************************

        nThreads    = 32;
        nBlocks     = ceil( (float) NP/ nThreads);

        if (t == 1)
            gettimeofday (&tdr0, NULL);

        resample <<< nBlocks, nThreads >>> (particles, particles_old, w, nloc);
        CUDA_CALL( cudaGetLastError() );
        CUDA_CALL( cudaDeviceSynchronize() );

        if (t == 1) {
            gettimeofday (&tdr1, NULL);
            timeval_subtract (&restime, &tdr1, &tdr0);
            std::cout << "\tResampling " << getHRtime(restime) << std::endl;
        }

        // Reduction **************************************************

        //if (t == (Tlim-1)) {

        if (pass == 49) {

            if (t == 1)
                gettimeofday (&tdr0, NULL);

            nThreads = 1;
            nBlocks  = 10;
```

```

                reduceStates <<< nBlocks, nThreads >>> (particles,
                   d_countmeans, t, T, nloc);
                CUDA_CALL( cudaGetLastError() );
                CUDA_CALL( cudaDeviceSynchronize() );

                if (t == 1) {
                    gettimeofday (&tdr1, NULL);
                    timeval_subtract (&restime, &tdr1, &tdr0);
                    std::cout << "Reduction        " << getHRtime(
                       restime) << std::endl;
                }

            }

            // Perturb particles
               *****************************************

            nThreads    = 32;
            nBlocks     = ceil( (float) NP/ nThreads);

            perturbParticles <<< nBlocks, nThreads >>> (particles,
               nloc, pass, 0.975);
            CUDA_CALL( cudaGetLastError() );
            CUDA_CALL( cudaDeviceSynchronize() );

            //}
            /*
            nThreads    = RB_DIM;
            nBlocks     = nCells;



            reduce <<< nBlocks, nThreads >>> (d_E, t, particles,
               Beta_last, nCells);
            CUDA_CALL( cudaGetLastError() );
            CUDA_CALL( cudaDeviceSynchronize() );

            if (t == 1) {
                gettimeofday (&tdr1, NULL);
                timeval_subtract (&restime, &tdr1, &tdr0);
                std::cout << "Reduction        " << getHRtime(
                   restime) << std::endl;
            }
            */


        } // end time

    } // end pass

    std::cout.precision(10);

    countmeans = (float*) malloc (nloc*T*sizeof(float));
    cudaMemcpy(countmeans, d_countmeans, nloc*T*sizeof(float),
        cudaMemcpyDeviceToHost);

    std::string filename = "cuIF2states.dat";

    std::cout << "Writing results to file '" << filename << "' ..."
        << std::endl;

    std::ofstream outfile;
    outfile.open(filename.c_str());

    for(int loc = 0; loc < nloc; loc++) {
        for (int t = 0; t < T; t++) {
            outfile << countmeans[loc*T + t] << " ";
        }
        outfile << std::endl;
    }

    /*
    double * h_w = (double*) malloc (NP*sizeof(double));
    cudaMemcpy(h_w, w, NP*sizeof(double), cudaMemcpyDeviceToHost);

    for (int n = 0; n < NP; n++) {
        std::cout << h_w[n] << " ";
    }
    */

    /*
    for (int i = 0; i < nCells; i++) {
        outfile << trueCounts[t*nCells + i];
        if (i % dim == 0)
            outfile << std::endl;
        else
            outfile << " ";
    }
    */

    outfile.close();

    gettimeofday (&tdr1, NULL);
    timeval_subtract (&restime, &tdr1, &tdrMaster);
    std::cout << "Total PF time (excluding setup) " << getHRtime(
        restime) << std::endl;

    cudaFree(d_data);
    cudaFree(particles);
    cudaFree(particles_old);
    cudaFree(w);
    cudaFree(d_neinum);
    cudaFree(d_neibmat);
    cudaFree(d_countmeans);

    exit (EXIT_SUCCESS);

}


/*  Use the Explicit Euler integration scheme to integrate SIR model forward in time
    float h     - time step size
    float t0    - start time
    float tn    - stop time
    float * y   - current system state; a three-component vector representing [S I R], susceptible-infected-recovered
    */
__device__ void exp_euler_SSIR(float h, float t0, float tn, Particle * particle, int * neinum, int * neibmat, int nloc) {

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

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

    // create last state vectors
    float * S_last = (float*) malloc (nloc*sizeof(float));
    float * I_last = (float*) malloc (nloc*sizeof(float));
    float * R_last = (float*) malloc (nloc*sizeof(float));
    float * B_last = (float*) malloc (nloc*sizeof(float));

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

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

        for (int loc = 0; loc < nloc; loc++) {
            S_last[loc] = S[loc];
            I_last[loc] = I[loc];
            R_last[loc] = R[loc];
            B_last[loc] = B[loc];
        }

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

            B[loc] = exp( log(B_last[loc]) + eta*(log(B0) - log(B_last[loc])) + berr*curand_normal(&(particle->randState)) );

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

            float nBIsum = 0.0;
            for (int j = 0; j < n; j++)
                nBIsum += B_last[neibmat[nloc*loc + j]-1] * I_last[neibmat[nloc*loc + j]-1];

            float BSI = S_last[loc]*( sphi*B_last[loc]*I_last[loc] + ophi*nBIsum );
            float rI  = r*I_last[loc];

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

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

        }

    }

    free(S_last);
    free(I_last);
    free(R_last);
    free(B_last);

}

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

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

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

}

/*  Convert memory size in bytes to human-readable format
    */
std::string getHRmemsize (size_t memsize) {

    std::stringstream ss;
    std::string valstring;

    int kb = 1024;
    int mb = kb*1024;
    int gb = mb*1024;

    if (memsize <= kb)
        ss << memsize << " B";
    else if (memsize > kb && memsize <= mb)
        ss << (float) memsize/ kb << " KB";
    else if (memsize > mb && memsize <= gb)
        ss << (float) memsize/ mb << " MB";
    else
        ss << (float) memsize/ gb << " GB";

    valstring = ss.str();

    return valstring;

}


/*  Convert time in seconds to human readable format
    */
std::string getHRtime (float runtime) {

    std::stringstream ss;
    std::string valstring;

    int mt = 60;
    int ht = mt*60;
    int dt = ht*24;

    if (runtime <= mt)
        ss << runtime << " s";
    else if (runtime > mt && runtime <= ht)
        ss << runtime/mt << " m";
    else if (runtime > ht && runtime <= dt)
        ss << runtime/dt << " h";
    else
        ss << runtime/ht << " d";

    valstring = ss.str();

    return valstring;

}
```

The parameter estimation means as compared to IF2 and HMCMC are shown in Figure [].

The running times for parameter fitting as compared to IF2 and HMCMC are shown in Figure[].