ESTIMATION AND INFERENCE OF NONLINEAR STOCHASTIC TIME SERIES

A COMPARATIVE STUDY OF TECHNIQUES FOR ESTIMATION AND
INFERENCE OF NONLINEAR STOCHASTIC TIME SERIES

By
DEXTER BARROWS, B.SC.

A Thesis Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements for the Degree
Master of Science

MASTER OF SCIENCE (2016) McMaster University
(Mathematics) Hamilton, Ontario, Canada

TITLE A Comparative Study of Techniques for Estimation and
Inference of Nonlinear Stochastic Time Series

AUTHOR Dexter Barrows, B.Sc. (Honours), Ryerson University

SUPERVISOR Dr. Benjamin Bolker

NUMBER OF PAGES x, 147

Abstract

Forecasting tools play an important role in public response to epidemics. Despite this, limited work has been done in comparing best-in-class techniques across the broad spectrum of time series forecasting methodologies. Forecasting frameworks were developed that utilised three methods designed to work with nonlinear dynamics: Iterated Filtering (IF) 2, Hamiltonian MCMC (HMC), and S-mapping. These were compared in several forecasting scenarios including a seasonal epidemic and a spatiotemporal epidemic. IF2 combined with parametric bootstrapping produced superior predictions in all scenarios. S-mapping combined with Dewdrop Regression produced forecasts slightly less-accurate than IF2 and HMC, but demonstrated vastly reduced running times. Hence, S-mapping with or without Dewdrop Regression should be used to glean initial insight into future epidemic behaviour, while IF2 and parametric bootstrapping should be used to refine forecast estimates in time.

Acknowledgements

There are many people I have to thank for their support over the last two years:

My supervisor Dr. Ben Bolker for his mentorship, advice, direction, and especially patience.

My defence committee members Dr. Jonathan Dushoff and Dr. David Earn who have taken the time to read my work and provide valuable input.

The Theobio lab for including me in stimulating discussions, even when they were over my head.

My Mom, Dad, Joel, and Sofía for being there for me through good times and trying ones.

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 [9][28][40]. 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 duration of an epidemic.

Despite the importance of reliable forecasts, obtaining them remains a challenge from both a theoretical and practical standpoint [28]. Mathematical models can capture the essential drivers in disease dynamics, and extend them past the present into the future. However, different epidemics may present with varying dynamics and require different model parameters to be accurately represented [7]. 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 [5].

Securing precise, error-free data in the midst of an outbreak can be difficult if not impossible [34], thus observational uncertainty must be built into mathematical models of disease spread from the beginning. 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, adding another layer of complexity [23]. With these caveats and concerns acknowledged, we can turn to a discussion of technique.

Broadly, there are three primary categories of approaches used in forecasting: phenomenological, pure mechanistic, and data integration.

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)

1 model. ARIMA assumes a linear underlying process and Gaussian error distributions.
2 It uses three parameters representing the degree of autoregression (p), integration
3 (trend removal) (d), and the moving average (q), where the orders of the autoregression
4 and the moving average are determined through the use of an autocorrelation function
5 (ACF) and partial autocorrelation function (PACF), respectively, applied to the the
6 data *a priori* [41].

7 Pure mechanistic approaches simply try to capture the essential drivers in the disease
8 spreading process and use the model alone to generate predictions. For example one
9 could use a model in which individuals are divided into categories based on whether
10 they are susceptible to infection or infected but not yet themselves infectious, infec-
11 tious, or recovered. These are called compartmental models and are heavily used in
12 epidemiological studies. Typically the transition between compartments is governed
13 by a set of ordinary differential equations, such as

14
$$
\begin{aligned}
\frac{dS}{dt} &= -\beta IS \\
\frac{dI}{dt} &= \beta IS - \gamma I \\
\frac{dR}{dt} &= \gamma I,
\end{aligned}
\tag{1.1}
$$

15 where S, I, and R are the number of individuals in each compartment, β is contact
16 rate between susceptible and infected individuals, and γ is a recovery rate. We also
17 let $\beta = R_0 r / N$, where R_0 is the number of secondary cases per infected individual
18 in a wholly susceptible population, and N is the population size. As an outbreak
19 progresses, individuals transition from the susceptible compartment, through the in-
20 fectious compartment, then finish in the removed compartment where they no longer
21 impact the system dynamics. Many extensions of the SIR model exist and are com-
22 monly used, such as the SEIR model in which susceptible individuals pass through an
23 exposed class (or several) where they have been infected but are not yet themselves
24 infectious, and the SIRS model in which individuals become susceptible again after
25 their immunity wanes [7].

26 Combining the phenomenological and mechanistic approaches are data integration
27 schemes. These methods use a model to define the expected underlying dynamics
28 of the system, but integrate data into the model in order to refine estimates of the
29 model parameters and produce more accurate forecasts. Typically the first step in
30 implementing such a technique is fitting the desired model to existing data. There are
31 many ways to do this, most of which fall into two main categories: Sequential Monte
32 Carlo (SMC) methods [3][32][40], and Markov chain Monte Carlo-based (MCMC)
33 methods [2][27]. From there data can either be integrated into the model by refitting
34 the model to the new longer data set, or in an "on-line" fashion in which data points

1 can be directly integrated without the need to refit the entire model. Normally,
2 MCMC-based machinery must refit the entire model whereas SMC-based approaches
3 can sometimes integrate data in an on-line fashion.

4 Another, broader, distinction among techniques can be drawn between those that rely
5 on assumptions of linearity, and those that make no such assumption. As epidemic
6 dynamics are highly non-linear, it can be questionable as to even consider linear ap-
7 proaches to epidemic forecasting at all. In particular, stalwart approaches such as
8 ARIMA and the venerable Kalman filter face a distinct (at least theoretical) disad-
9 vantage in the face of newer SMC-based methods [37][40]. Extensions of the Kalman
10 filter, such as the ensemble adjustment Kalman filter are designed to handle non-
11 linearity, but these methods are very well-studied, and further work showing their
12 viability would likely prove extraneous in the modern academic landscape.

13 Somewhat frustratingly, there exists no "gold standard" in forecasting [9][28][40]. As
14 methodology varies widely in theoretical justification, implementation, and operation,
15 it is difficult to compare the state of the art in forecasting methods from a first-
16 principles perspective. Further, published work making use of any of these methods
17 to forecast uses different prediction accuracy metrics, such as SSE, peak time/dura-
18 tion/intensity, correlation tests, or RMSE, among others [9][29]. Thus it is difficult
19 to select the best tool for the job when faced with a forecasting problem.

20 The primary focus of this work is to compare best-in-class methods for forecasting in
21 several epidemically-focused scenarios. These include the a "standard" one-shot fore-
22 cast outbreak in which the outbreak subsides and does not recur, a seasonal outbreak
23 scenario such as what we see with influenza each year, and a spatiotemporal scenario
24 in which multiple spatial locations are connected and disease is free to spread from
25 one to another.

26 From MCMC-based methods we have selected Hamiltonian MCMC (HMC) [27], a
27 (slightly) less cutting-edge but nonetheless highly effective technique. We are us-
28 ing HMC through an implementation in the R package RStan [8], which at its core
29 uses HMC, but also contains implementations of several other innovative techniques.
30 Interestingly, the original goal of this package was not to implement a statistical
31 programming language similar to Just Another Gibbs Sampler (JAGS) or Bayesian
32 Inference Using Gibbs Sampling (BUGS), but with an HMC backend. In fact the
33 developers' original goal was to implement any method that could fit multilevel hi-
34 erarchical models without halting as they were witnessing with BUGS and JAGS.
35 Only after experimenting with several options and starting to hear about it more and
36 more frequently did they attempt to work with HMC. In the end, the scope of the
37 project grew to include the development and subsequent integration of the No-U-Turn
38 Sampler (NUTS) [18], and an implementation of automatic differentiation machinery
39 [35].

40 For PF-based methods we have selected Iterated Filtering (IF) 2 [21], a very recently

developed approach that uses multiple particle filtering rounds to generate maximum likelihood estimates (MLEs). It functions similarly to its predecessor, the Maximum likelihood via Iterated Filtering (MIF) algorithm [20], but aims to be simpler, faster, and more accurate. Theoretical justification and synthetic testing indicates that IF2 meets these goals, and as such the authors recommend skipping MIF and jumping straight to IF2 if an algorithm of their variety is sought. And so, we are doing just that. We wrote our own IF2 implementation in C++ and integrated it into R using the Rcpp package [12]. The developers of MIF and IF2 have their own R package that implements MIF and IF2, Partially Observed Markov Processes (POMP) [22][24], but it didn't provide some of the diagnostic information we needed, so it was not used here.

Finally, from the phenomenological methods we have selected the sequential locally weighted global linear maps (S-map) [14][19][36][37], combined with Dewdrop Regression [14]. These methods stand on their own as a unique take on the forecasting problem, and bear little resemblance to other methodology. The virtues of these techniques have been long-extolled by their developers, but their efficacy when compared to competing methods has not been well-studied. This work will mark one of the first times this has been done.

This paper will begin with descriptions of HMC and IF2 with examples of simple model fitting in Chapters 2 and 3. Chapter 4 explores parameter fitting of a stochastic SIR model to synthetic data. Chapter 5 will establish the full forecasting frameworks used with IF2 and HMC, and compare them in a simple scenario. All three methods will be used to compare forecasts using a SIRS model in Chapter 6. Chapter 7 will show forecasts using the aforementioned IF2 and HMC frameworks, along with Dewdrop Regression combined with S-mapping. Finally, a summary of these results, and a discussion of parallel computing and future directions will finish the paper in Chapter 8.

1 Chapter 2

2 Hamiltonian MCMC

3 Markov Chain Monte Carlo (MCMC) is a general class of methods designed to sample
4 from the posterior distribution of model parameters [2]. It is an algorithm used when
5 we wish to fit a model M that depends on some parameter (or more typically vector
6 of parameters) θ to observed data D. MCMC works by constructing a Markov chain
7 whose stationary distribution converges to desired posterior distribution. The samples
8 drawn using MCMC are used to numerically approximate the stationary distribution,
9 and in turn the posterior [2].

10 2.1 Markov Chains

11 Figure [2.1] shows a finite state machine with 3 states $S = \{x_1, x_2, x_3\}$.

12 The transition probabilities can be summarized as a matrix as

13
$$T = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0.1 & 0.9 \\ 0.6 & 0.4 & 0 \end{bmatrix}. \tag{2.1}$$

14 The probability vector $\mu(x^{(1)})$ for a state $x^{(1)}$ can be evolved using T by evaluating
15 $\mu(x^{(1)})T$, then again by evaluating $\mu(x^{(1)})T^2$, and so on. If we take the limit as the
16 number of transitions approaches infinity, we find

17
$$\lim_{t \to \infty} \mu(x^{(1)})T^t = (27/122, 50/122, 45/122). \tag{2.2}$$
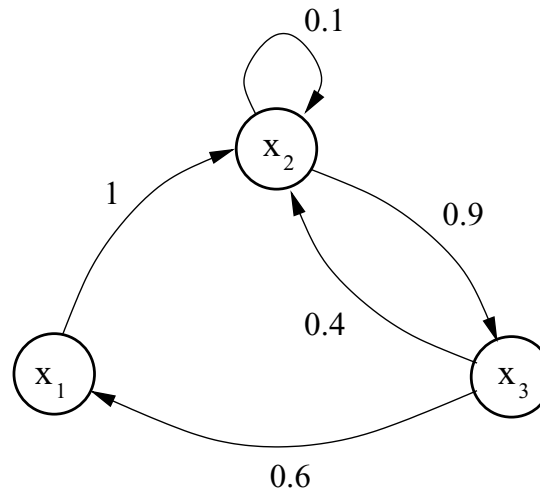
Figure 2.1: A finite state machine. States are shown as graph nodes, and the probability of transitioning from one particular state to another is shown as a weighted graph edge. [2]

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.

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].

2.2   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;

1 this is done by specifying a likelihood function $\mathcal{L}(\theta)$ such that

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

3 In frequentist Maximum Likelihood approaches, $\mathcal{L}(\theta)$ is searched to find a value of θ
4 that maximizes $\mathcal{L}(\theta)$, then this θ is taken to be the most likely true value. Bayesian
5 approaches take this further by aiming to generate a posterior distribution of likelihood
6 values conditioned on prior information about the parameters and the data – to not
7 just maximize the likelihood but to also explore the area around it [2].

2.3 Prior distribution

9 Another significant component of MCMC is the user-specified prior distribution for
10 θ or distributions for the individual components of θ (priors). Priors serve as a way
11 for us to tell the MCMC algorithm what we think consist of good values for the
12 parameters. Note that if very little is known about the parameters, or we are worried
13 about biasing our estimate of the posterior, we can simply use a a wide uniform
14 distribution. We cannot, however, avoid this problem entirely. Bayesian frameworks,
15 such as MCMC, *require* priors to be specified; what the user must decide is how strong
16 to make priors.

17 Exceedingly weak priors can pose problematic in some circumstances. In the case of
18 MCMC, weak priors handicap the algorithm in two ways: convergence of the chain
19 may become exceedingly slow, and more pressure is put on the likelihood function to
20 be as good as possible – it will now be the only thing informing the algorithm of what
21 constitutes a "good" set of parameters, and what should be considered poor. In the
22 majority of cases this does not pose as much as problem as it would appear; if enough
23 samples are drawn, we should still obtain a good posterior estimate. We will only
24 really run into problems if an exceedingly weak prior, such as an unbounded uniform
25 distribution, or another unbounded distribution with a high standard deviation, is
26 specified – in those cases we may obtain poor posterior estimates if the data are weak
27 [2].

2.4 Proposal distribution

29 As part of the MCMC algorithm, when we find a state in the parameter space that
30 is accepted as part of the Markov chain construction process, we need a good way
31 of generating a good next step to try. Unlike basic rejection sampling in which we
32 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 our prior distribution.

The choice of this distribution is theoretically not of the utmost importance, but in practice becomes important so as to not waste computer time [2].

2.5   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. This is known as the Metropolis-Hastings rule.

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

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

Algorithm [1] shows the Metropolis MCMC algorithm.

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

---

Algorithm 1: Metropolis MCMC

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

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

```
    /* Sample                                                  */
```
2      $\theta^* \sim q(\cdot | \theta^{(i-1)})$

3      $u \sim \mathcal{U}(0,1)$

```
    /* Evaluate acceptance ratio                               */
```
4      $r \leftarrow \frac{\mathcal{L}(\theta^*)p(\theta^*)}{\mathcal{L}(\theta)p(\theta)}$

```
    /* Step acceptance criterion                               */
```
5   **if** $u < \min\{1, r\}$ **then**

6         $\theta^{(i)} = \theta^*$

7   **else**

8         $\theta^{(i)} = \theta^{(i-1)}$

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

---

ensures that as much of the parameter space as possible will be explored but more efficiently than using pure brute force [2].

2.6 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].

2.7   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 [25].

2.8   Hamiltonian Monte Carlo

The Metropolis-Hastings algorithm has a primary drawback in that the parameter space may not be explored efficiently in some circumstances – 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 potential decrease in overall computational complexity. This algorithm is coined Hamiltonian MCMC (HMC) [27]. Prior to the advent of HMC, some work was conducted exploring adaptive step-sizing using MCMC-based methods, but found they lack strong theoretical justification, and can lead to some samples being drawn from an incorrect distribution [27].

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 [27]. However, some recent algorithms, such as the No U-Turn sampler implemented in RStan, and adaptively select appropriate values automatically during the sampling process [18].

Algorithm 2: Hamiltonian MCMC

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

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

     /* Resample moments   */
2   **for** $i = 1 : n$ **do**
3 $r(i) \leftarrow \mathcal{N}(0, 1)$

     /* Leapfrog initialization   */
4 $\theta_0 \leftarrow \theta^{(i-1)}$
5 $r_0 \leftarrow r - \nabla U(\theta_0) \cdot \varepsilon / 2$

     /* Leapfrog intermediate steps   */
6   **for** $j = 1 : L - 1$ **do**
7 $\theta_j \leftarrow \theta_{j-1} + M^{-1} r_{j-1} \cdot \varepsilon$
8 $r_j \leftarrow r_{j-1} - \nabla U(\theta_j) \cdot \varepsilon$

     /* Leapfrog last steps   */
9 $\theta^* \leftarrow \theta_{L-1} + M^{-1} r_{L-1} \cdot \varepsilon$
10 $r^* \leftarrow \nabla U(\theta_L) \cdot \varepsilon / 2 - r_{L-1}$

     /* Evaluate acceptance ratio   */
11 $r = \exp\left[H(\theta^{(i-1)}, r) - H(\theta^*, r^*) \right]$

     /* Sample   */
12 $u \sim \mathcal{U}(0, 1)$

     /* Step acceptance criterion   */
13   **if** $u < \min\{1, r\}$ **then**
14 $\theta^{(i)} = \theta^*$
15   **else**
16 $\theta^{(i)} = \theta^{(i-1)}$

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

2.9   RStan 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 outlined in the introduction in Equation [1.1].

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 the gradient in Equation [1.1].

The true parameter values were set to $R_0 = 3.0, r = 0.1, N = 500$. 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. The system was integrated over $[0, 100]$ with infected counts drawn at each integer time step.

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

Figure [2.2] shows the system simulation results.

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.

Throughout this paper, the explicit Euler integration scheme was used to obtain solutions to our ODE-based models. While this scheme is not the most accurate or efficient one available, it as chosen for its ease of implementation in the required languages and transparency with regards to stochastic processes, which have been added into later models. Using a more advanced integrator such as Runge-Kutta makes it harder to properly specify how stochastic process evolution should be handled, and would have required significantly more implementation work to boot. Hence, we have opted for the lo-fi solution we know will function the way we require.

In order to use an Explicit Euler-like stepping method in the later Stan model, the synthetic observation counts were treated as weekly observations in which the counts on the other six days of the week were unobserved.

Figure [2.3] shows the traceplot for the the post-burn-in chain data returned by the RStan fitting. We see that the chains are mixing well and convergence has likely been reached.
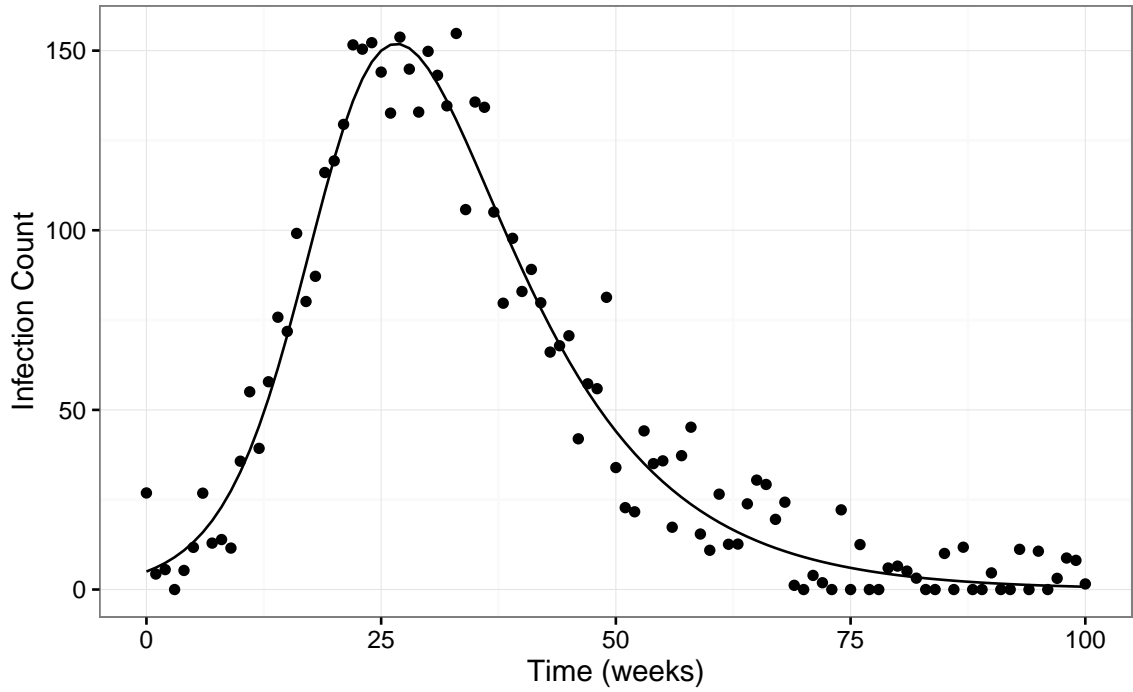
Figure 2.2: True SIR ODE solution infected counts, and with added observation noise.

1 Figure [2.4] shows the chain data including the burn-in samples. We can see why is
2 is wise to discard these samples (note the scale).

3 Figure [2.5] shows the the kernel density estimates for each of the model parameters
4 and the initial number of cases. We see that while the estimates are not perfect, they
5 are more than satisfactory.
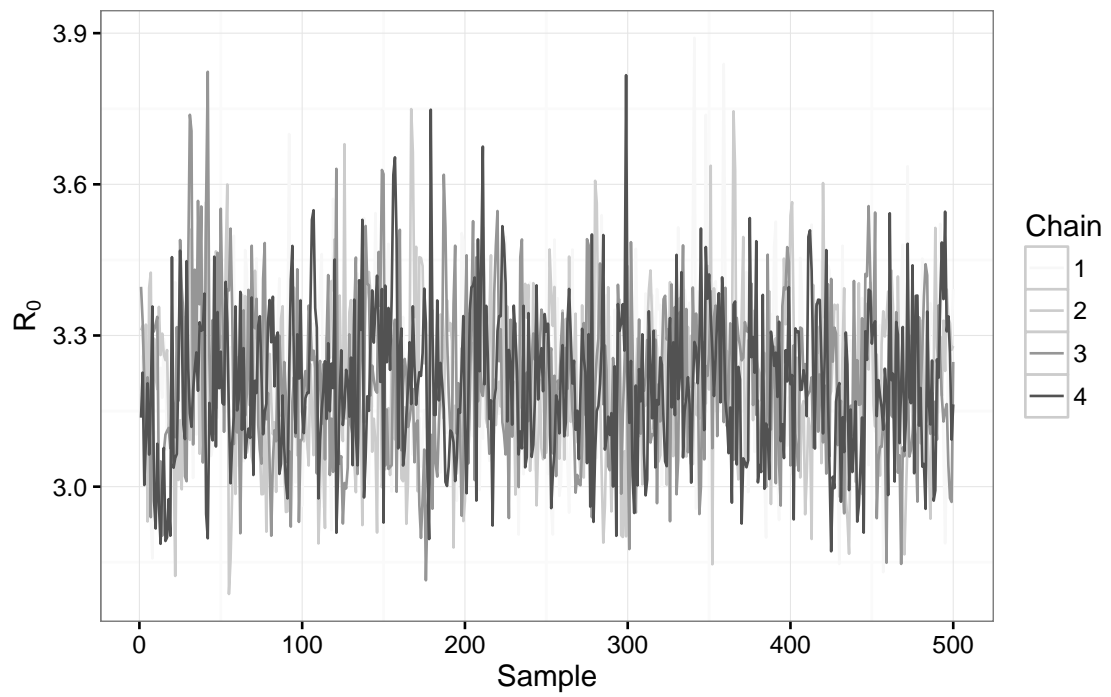
Figure 2.3: Traceplot of samples drawn for parameter R_0, excluding burn-in.
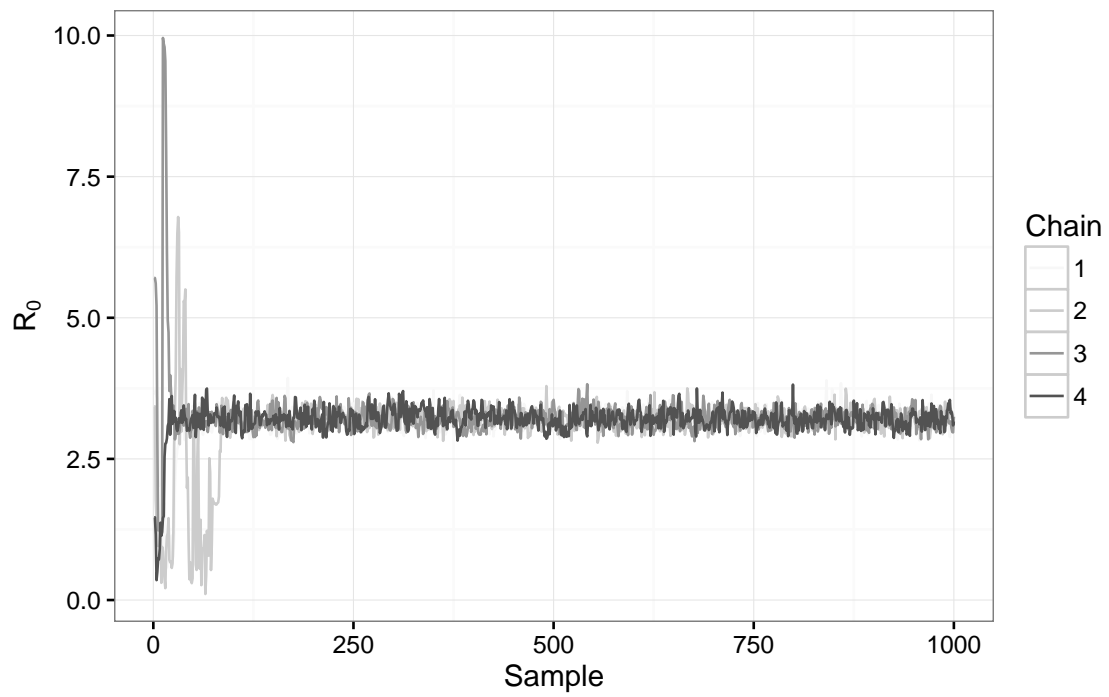
Figure 2.4: Traceplot of samples drawn for parameter R_0, including burn-in.
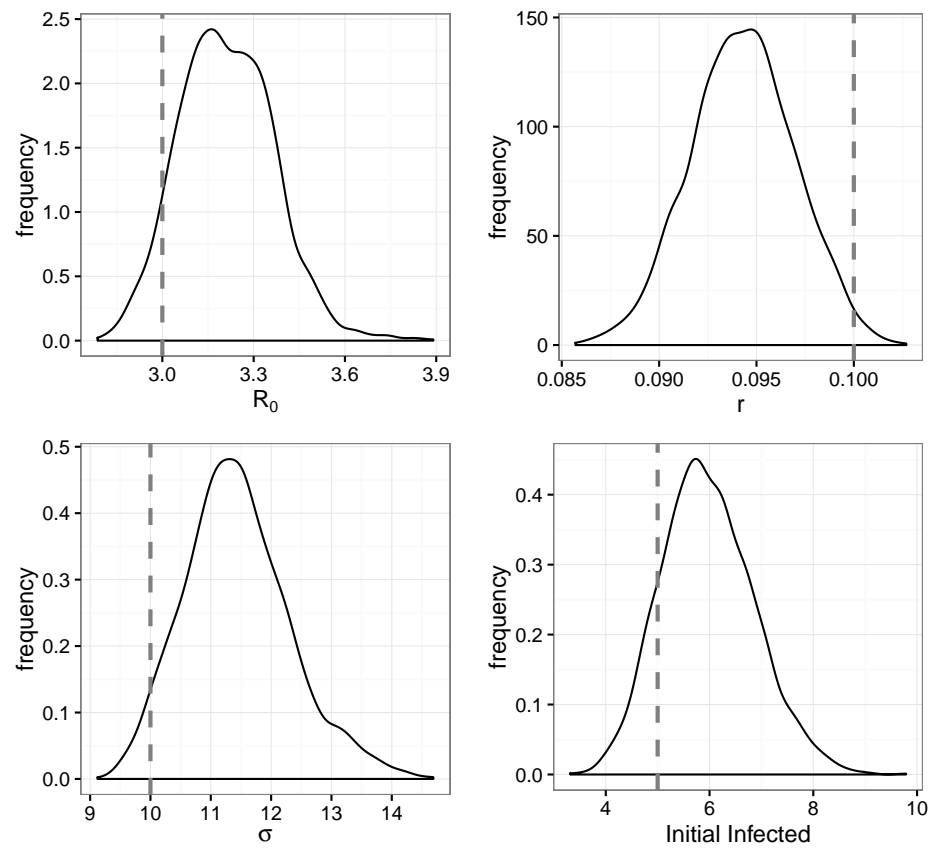
Figure 2.5: Kernel density estimates produced by Stan. Dashed lines show true parameter values.

Chapter 3

Iterated Filtering

Particle filters are similar to MCMC-based methods in that they use likelihoods to evaluate the validity of proposed parameter sets given observed data D, but differ in that they are largely trying to produce point estimates of the parameters instead of samples from the posterior distribution.

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 analogous to Darwinian selection, then this will be a Sequential Importance Resampling (SIR) particle filter [3].

3.1   Formulation

Particle filters, also called Sequential Monte-Carlo (SMC) 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 [3].

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

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

18

1 And the observation process by

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

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

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

3.2 Algorithm

14 Now we will formalize the particle filter.

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

18 The algorithm for a Sequential Importance Resampling particle is shown in Algorithm
19 [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   */

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

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

   /* Evolve   */

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

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

   /* Weight   */

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

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

   /* Normalize   */

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

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

   /* Resample   */

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

/* Samples from approximated posterior distribution   */

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

3.3   Particle Collapse

Often, 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 [6][3].

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

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

This can be used to diagnose not only when collapse has occurred, but can also indicate when it is near [3].

3.4   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 [20][21] 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.

3.5 Iterated Filtering 2 (IF2)

The successor to Iterated Filtering 1 [20], Iterated Filtering 2 [21] is simpler, faster, and demonstrated better convergence toward maximum likelihood. The core concept involves a two-pronged approach. First, a data cloning-like procedure 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.

IF2 is not designed to estimate the full posterior distribution, instead to produce a Maximum Likelihood (ML) point estimate. 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 multivariate 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 through 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   */
1 Initialize particle cohort by sampling $(p^{(1)}, p^{(2)}, ..., p^{(J)})$ from P_0

/* Particle seeding distribution   */
2 $\Theta \leftarrow P_0$

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

    /* Pass perturbation   */
4   **for** $j = 1{:}J$ **do**
5         $p^{(j)} \sim h(\Theta^{(j)}, \sigma_m)$

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

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

            /* Iteration perturbation   */
8             $p^{(j)} \sim h(p^{(j)}, \sigma_m)$

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

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

        /* Normalize   */
11   **for** $j = 1{:}J$ **do**
12           $w^{(j)} \leftarrow w^{(j)} / \sum_1^J w^{(j)}$

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

    /* Collect particles for next pass   */
14   **for** $j = 1 : J$ **do**
15         $\Theta^{(j)} \leftarrow p^{(j)}$

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

3.6 IF2 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.

As in the previous section, the model in Equation [1.1] was use to produce synthetic data. The same parameters and initial conditions were used, namely: parameter values were set to $R_0 = 3.0, r = 0.1, N = 500$, initial conditions were set to 5 infectious individuals, 495 people susceptible to infection, and no one had yet recovered from infection and been removed, and observation error was taken to be $\varepsilon_{obs} \sim \mathcal{N}(0, \sigma)$, where individual values were drawn for each synthetic data point.

Figure [2.2] in the previous section shows the true SIR ODE system solution and data.

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

There are three primary reasons we implemented our own version of IF2 instead of using POMP. First, POMP does not provide final particle state distributions, making it difficult to calibrate the algorithm parameters against the parameters used in RStan (this procedure is described in the next chapter). Second, it is prudent to cross-check the validity of an algorithm using another implementation. Third, this code can then serve as a jumping-off point for further development using Graphics Processing Unit acceleration (outlined in Chapter 8). We must acknowledge the disadvantages as well: POMP has been extensively vetted with real-world usage, and using it would require far less work as we would only need to specify the model. That being said, we believe the advantages outweigh the disadvantages in this case, and so have proceeded to develop our own implementation of IF2.

Figure [3.1] shows the final kernel estimates for four of the key parameters. As with HMC, the distributions are not perfect, but are promising. Unlike with HMC, these distributions are not meant to consist of samples from the true posterior distribution, but rather serve a diagnostic role.
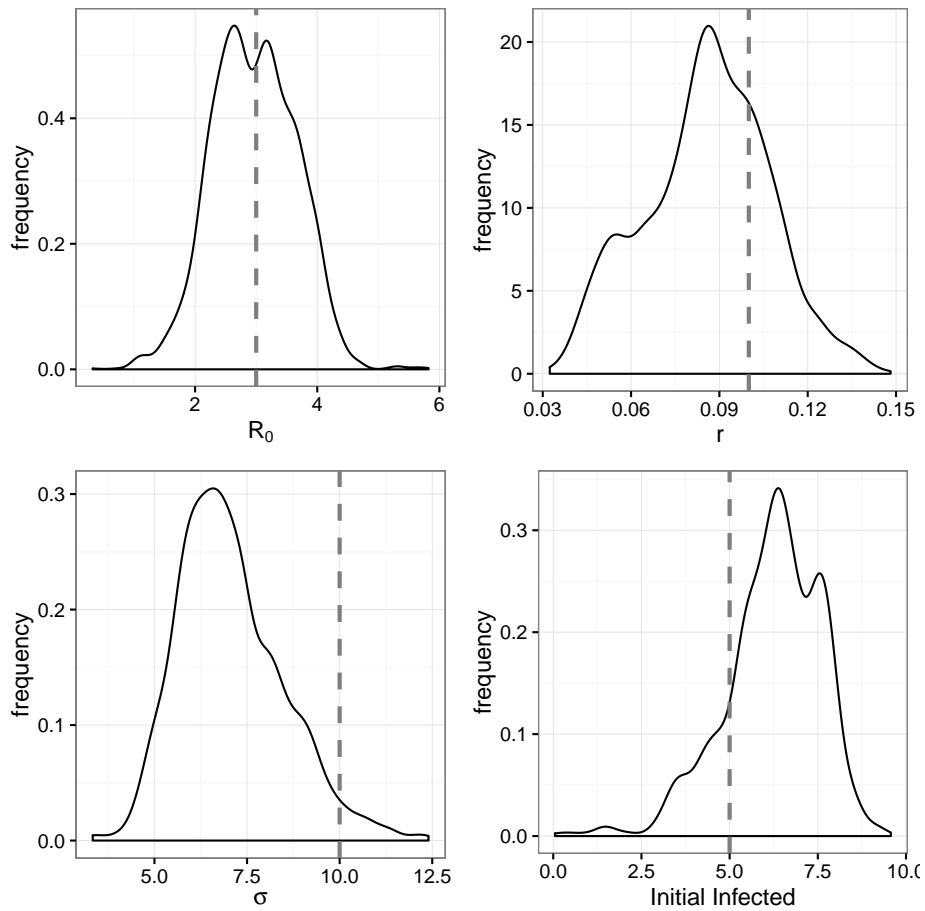
Figure 3.1: Kernel estimates for four essential system parameters. True values are indicated by dashed lines.

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 specified in Equation [1.1] and add process noise by allowing β to follow a geometric random walk given by

$$\beta_{t+1} = \exp\left(\log(\beta_t) + \eta(\log(\bar{\beta}) - \log(\beta_t)) + \epsilon_t\right). \tag{4.1}$$

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).

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

$$X_{t+1} = c + \rho X_t + \epsilon_t, \tag{4.2}$$

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 $\sigma = \sigma_E^2/(1-\rho^2)$.
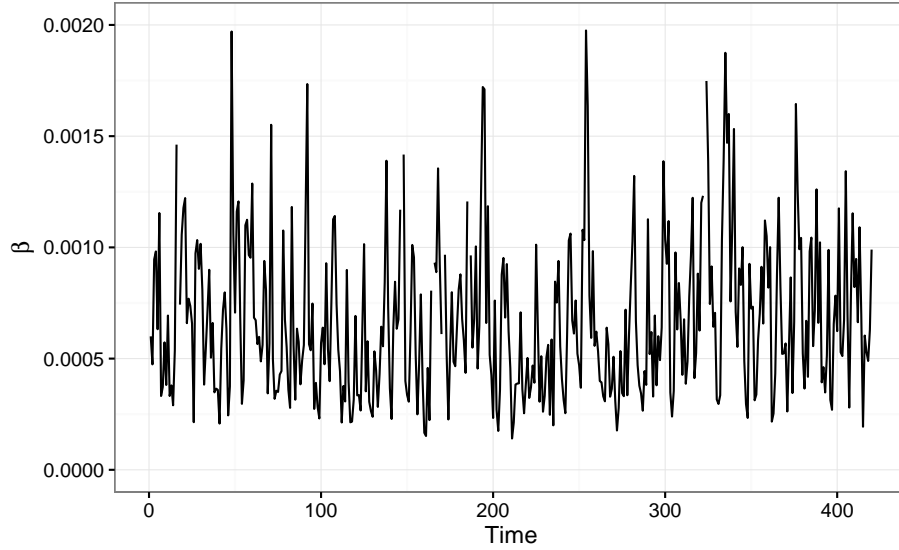
Figure 4.1: Simulated geometric autoregressive process show in Equation [4.1].

1 If we choose $\eta = 0.5$, the resulting log-normal distribution has a mean of 6.80×10^{-4}
2 and standard deviation of 4.46×10^{-4}.

3 Figure [4.1] shows the result of simulating the process in Equation [4.1] with $\eta =$
4 0.5.

5 Figure [4.2] shows the density plot corresponding to the values in Figure [4.1].

6 We see a density plot similar in shape to the desired density, and the geometric random
7 walk displays dependence on previous values. Further the mean of this distribution
8 was calculated to be 6.92×10^{-4} and standard the deviation to be 3.99×10^{-4}, which
9 are very close to the theoretical values.

10 If we take the full stochastic SIR system and evolve it using an Euler stepping scheme
11 with a step size of $h = 1/7$, for 1 step per day, we obtain the plot in Figure [4.3].
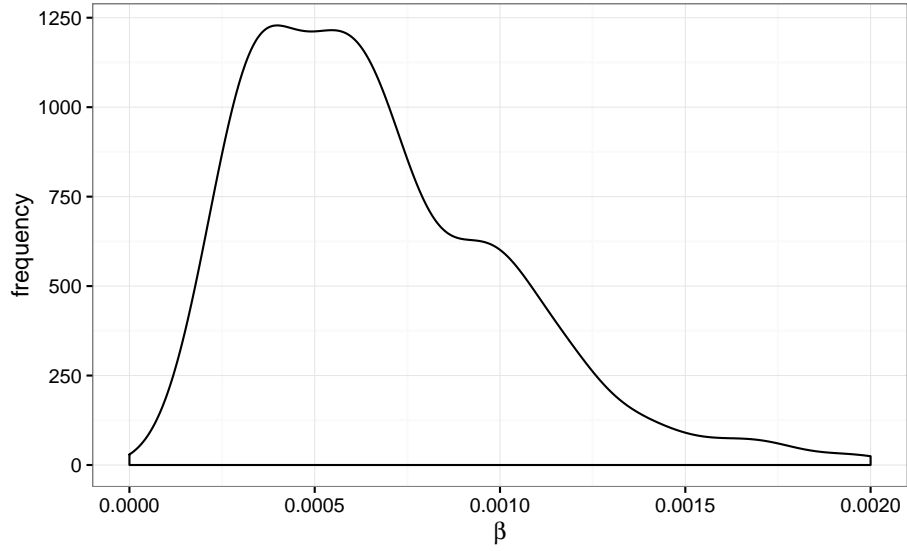
Figure 4.2: Density plot of values shown if Figure[4.1].
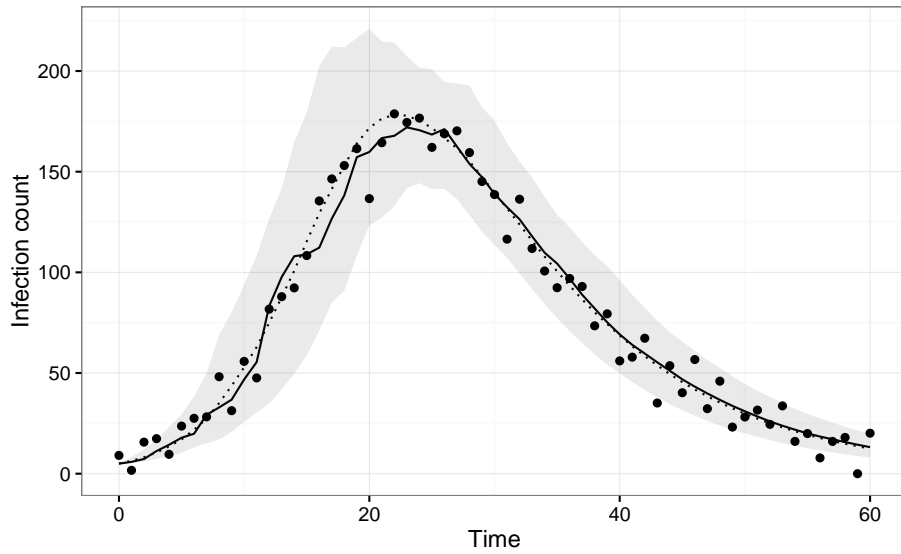
Figure 4.3: Stochastic SIR model simulated using an explicit Euler stepping scheme. The solid line is a single random trajectory, the dots show the data points obtained by adding in observation error defined as $\epsilon_{\text{obvs}} = \mathcal{N}(0, 10)$, and the grey ribbon is centre 95th quantile from 100 random trajectories.

4.2   Calibrating Samples

In order to compare HMC and IF2 we need to set up a fair and theoretically justified way to select the number of samples to draw for the HMC iterations and the number of particles to use for IF2. As we wish to compare, among other things, approximate computational cost using runtimes, we need to determine how many sample draws for each method are required to obtain a certain accuracy. Sample draws are typically not comparable in terms of quality when considering multiple methods. For example, vanilla MCMC draws are computationally cheap compared to those from HMC, but HMC produces draws that more efficiently cover the sampling space. Thus we cannot just set the number of HMC draws equal to the number of particles used in IF2 – we must calibrate both quantities based on a desired target error. We assume that we are working with a problem that has an unknown real solution, so we use the Monte Carlo Standard Error (MCSE) [16].

Suppose we are using a Monte-Carlo based method to obtain a mean 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

$$\hat{\sigma}_n = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(X_i - \hat{\mu})}, \tag{4.3}$$

which is known as the Monte Carlo Standard Error.

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

$$\Theta^* V (\Theta^*)^T \tag{4.4}$$

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

$$\hat{\sigma}_n = \sqrt{\frac{1}{n}\frac{1}{P}\Theta^* V (\Theta^*)^T} \tag{4.5}$$

1 where P is the number of particles.

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

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

11 4.3   IF2 Fitting

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

17 The code used here is a mix of R and C++ implemented using Rcpp. The fitting
18 was undertaken using 2500 particles with 50 IF2 passes and a cooling schedule given
19 by a reduction in particle spread determined by 0.975^p, where p is the pass number
20 starting with 0. This geometric cooling scheme is standard for use with IF2 [22][24],
21 with the cooling rate chosen to neatly scale the perturbation factor from 1 to 0.02
22 (almost 0) over 50 passes.

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

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

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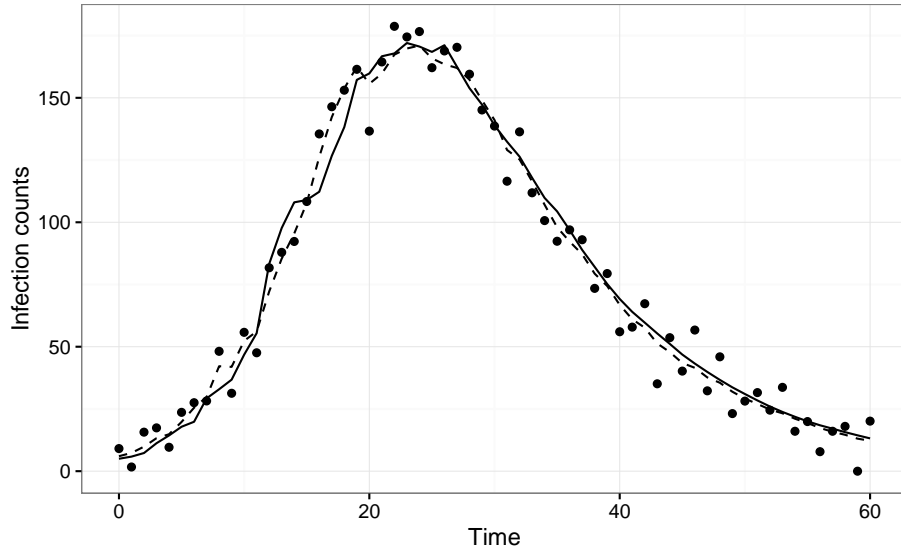
Figure 4.5: True system trajectory (solid line), observed data (dots), and IF2 estimated real state (dashed line).

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. We expect near-convergence in the parameter estimates as IF2 nears the maximum number of passes specified. Unconvincing convergence plots may signal suboptimal algorithm parameters. If sensible algorithm parameters have been chosen, we should see the convergence plots display "flattening" over time.

Figure [4.6] shows evolution of the mean estimates for the six most critical parameters.

Figure [4.7] shows the evolution of the standard deviations of the parameter estimates from the particle swarm as a function of the pass number. We should expect to see asymptotic convergence to zero if the filter is converging.
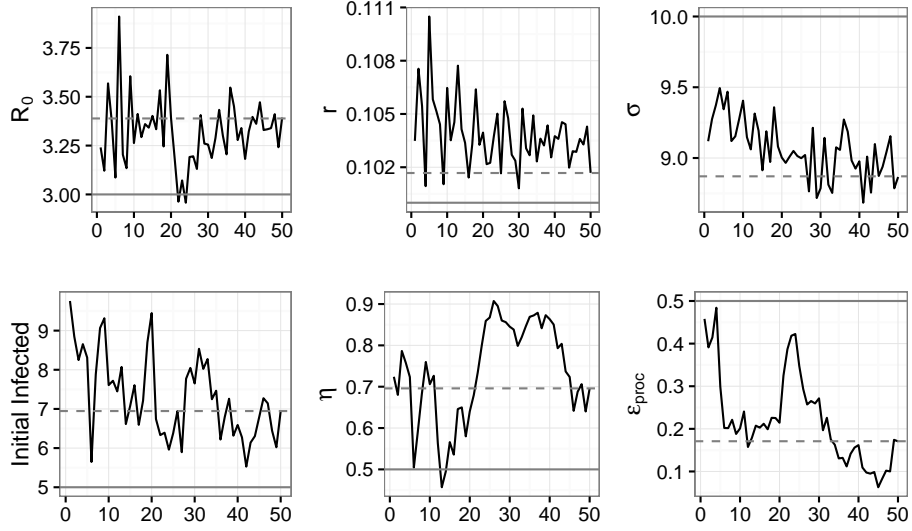
Figure 4.6: The horizontal axis shows the IF2 pass number. The solid black lines show the evolution of the ML estimates, the solid grey lines show the true value, and the dashed grey lines show the mean parameter estimates from the particle swarm after the final pass.
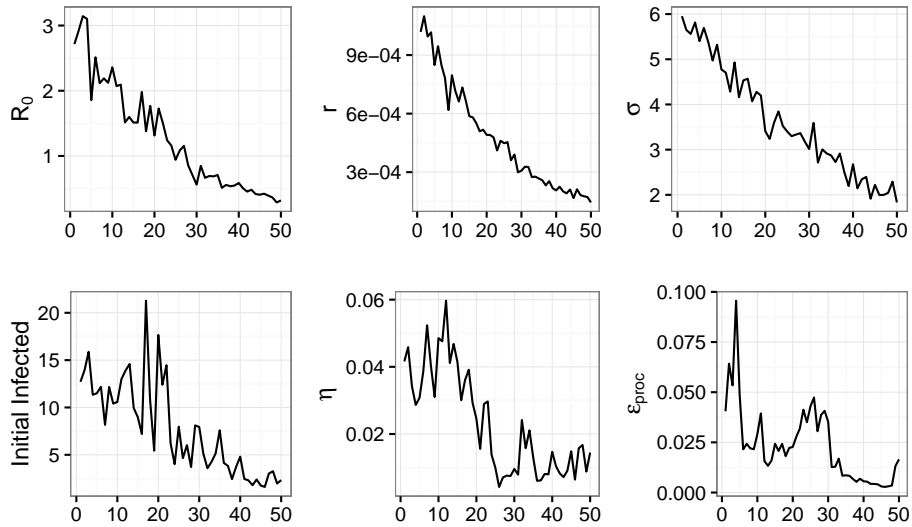
Figure 4.7: The horizontal axis shows the IF2 pass number and the solid black lines show the evolution of the standard deviations of the particle swarm values.

4.5   IF2 Densities

Of diagnostic importance are the densities of the parameter estimates given by the final parameter swarm. If the swarm has collapsed, these densities will be extremely narrow, almost resembling a vertical line. A "healthy" swarm should display relatively smooth kernels of reasonable breadth.

Figure [4.8] shows the parameter sample distributions from the final parameter swarm.

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. This is undesirable as it may indicate instability – the particles may have become "trapped" in a region of the sampling space.
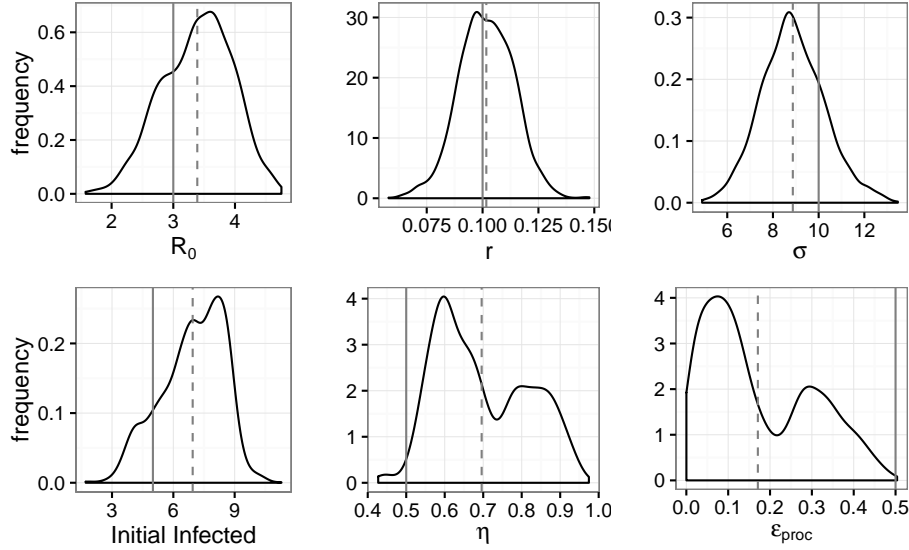
Figure 4.8: The solid grey lines show the true parameter values and the dashed grey lines show the density medians.

4.6   HMC 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   HMC Densities

Figure [4.9] shows the parameter estimation densities from the Stan HMC fitting.

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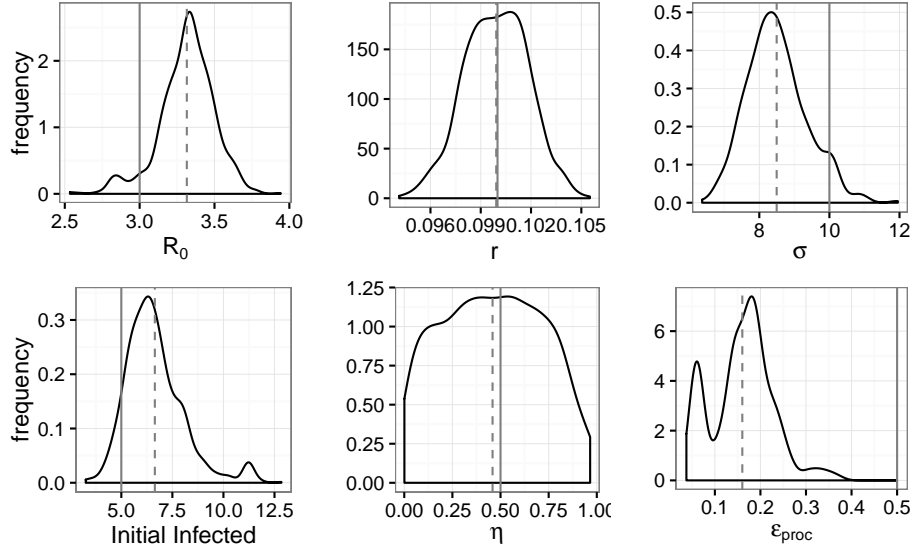
Figure 4.9: As before, the solid grey lines show the true parameter values and the dashed grey lines show the density medians.

4.8   HMC and Bootstrapping

Unlike in some models, our RStan epidemic model does not keep track of state estimates directly, but does keep track of the autoregressive process latent variable draws, which allow us to reconstruct states. This was done to ease implementation as RStan places some restrictions on how interactions between parameters and states can be specified.

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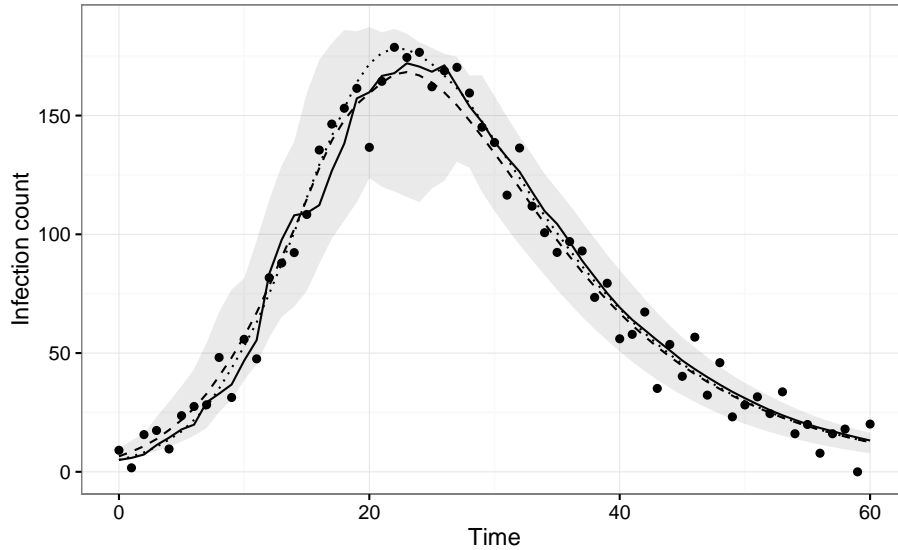
Figure 4.10: Result from 100 HMC bootstrap trajectories. The solid line shows the true states, the dots show the data, the dotted line shows the average system behaviour, the dashed line shows the bootstrap mean, and the grey ribbon shows the centre 95th quantile of the bootstrap trajectories.

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.

Figure [4.11] shows the results of the mean parameter fits from IF2 and HMC for 200 independent data sets generates using the previously described model.

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

Figure [4.12] summarizes the running times for each algorithm.

The average running times were approximately 45.5 seconds and 257.4 seconds for IF2 and HMC respectively, representing a 5.7x speedup for IF2 over HMC. While IF2 may be able to fit the model to data faster than HMC, 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 HMC are anything but, showing a wide spread of potential times.
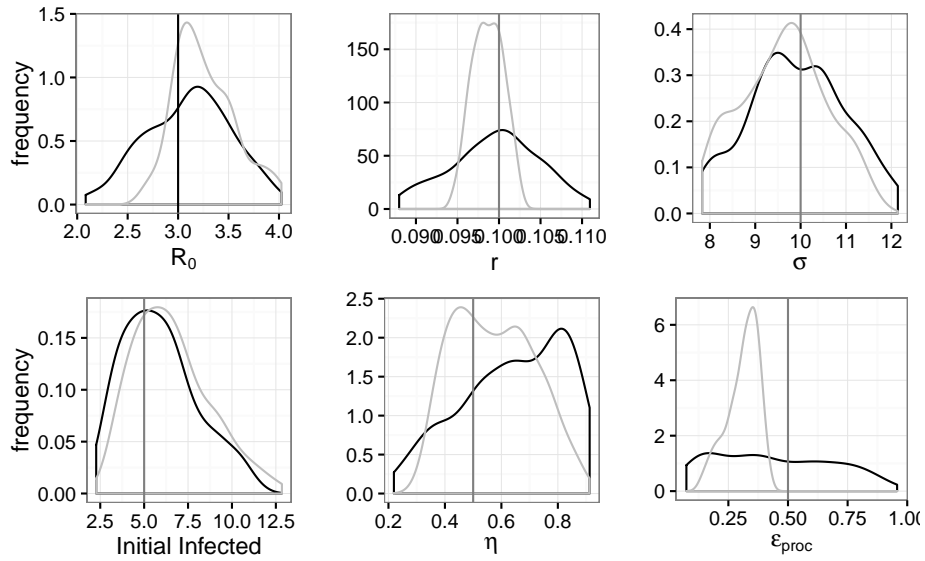
Figure 4.11: IF2 point estimate densities are shown in black and HMC point estimate densities are shown in grey. The vertical lines show the true parameter values.
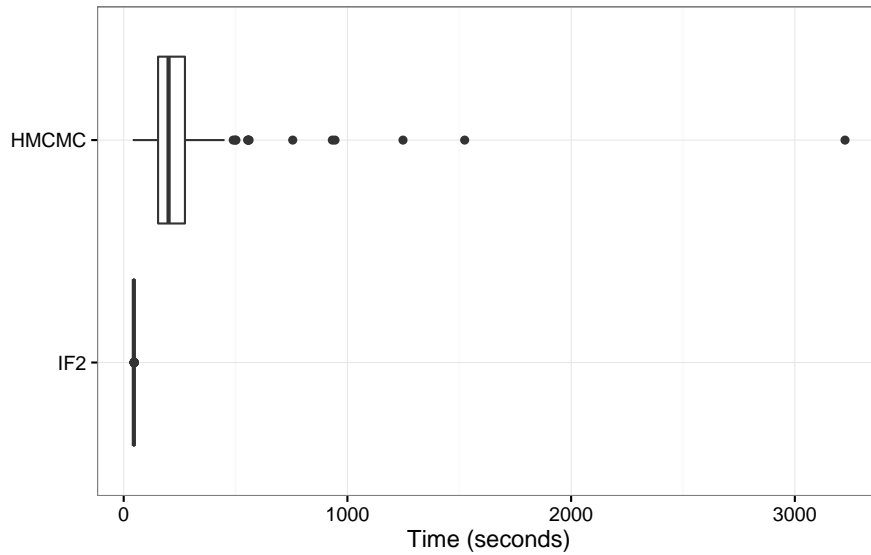
Figure 4.12: Fitting times for IF2 and HMC, in seconds. The centre box in each plot shows the centre 50th percent, with the bold centre line showing the median.

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 HMC can reconstruct out-of-sample forecasts.

Figure [5.1] shows an example of a simulated system with truncated data.

In essence, we want to be able to give either IF2 of HMC only the data points and have it reconstruct the entirety of the true system states.
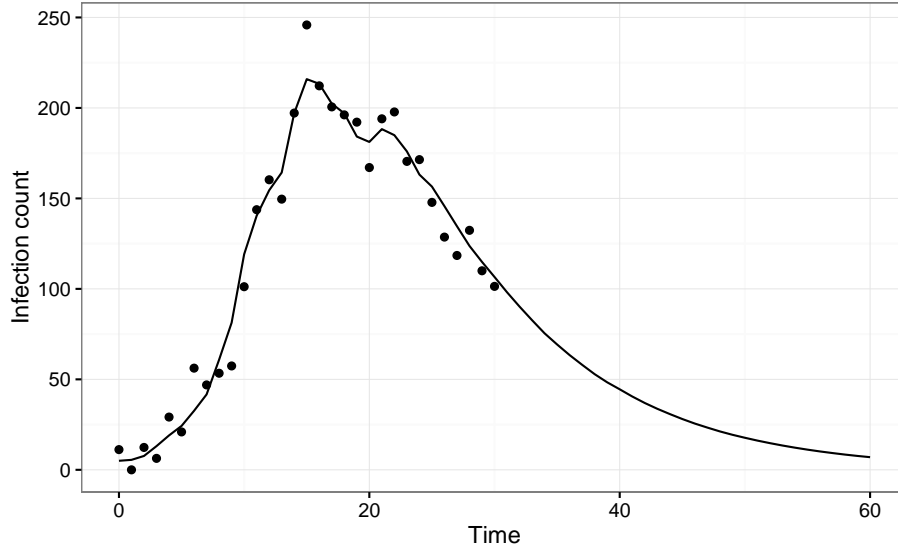
Figure 5.1: Infection count data truncated at $T = 30$. The solid line shows the true underlying system states, and the dots show those states with added observation noise. Parameters used were $R_0 = 3.0$, $r = 0.1$, $\eta = .05$, $\sigma_{proc} = 0.5$, and additive observation noise was drawn from $\mathcal{N}(0, 10)$.

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 ML point estimates for the parameters. 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 the future.

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

Figure 2 shows [5.2] the state estimates for each time point produced by the last IF2 pass.

Recall that IF2 is not trying to generate posterior probability 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 add another layer atop the IF2 machinery, parametric bootstrapping.
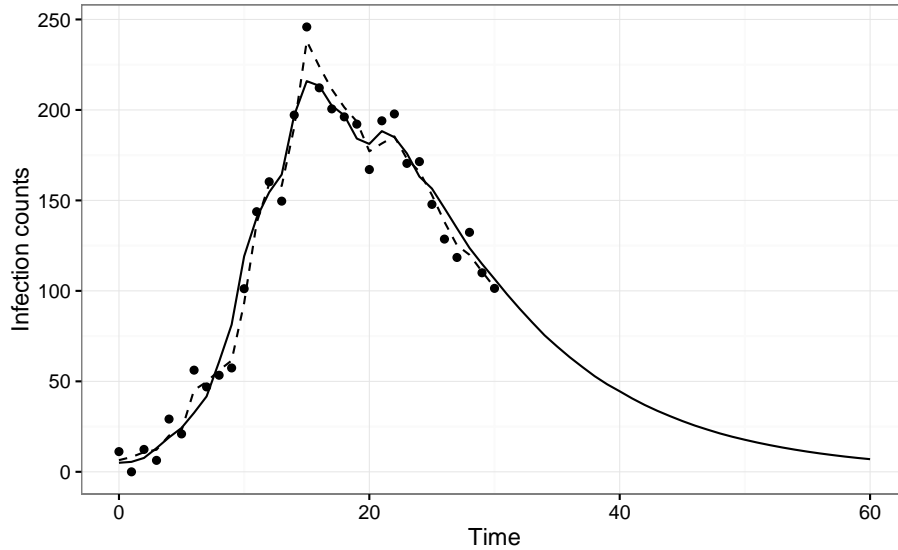
Figure 5.2: Infection count data truncated at $T = 30$ from Figure [5.1]. The dashed line shows IF2's attempt to reconstruct the true underlying state from the observed data points.

5.2.1   Parametric Bootstrapping

The goal of the parametric bootstrap is use an initial density sample θ^* to generate further samples $\theta_1, \theta_2, ..., \theta_M$ from the sampling distribution of θ. 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. The literature suggests the most straightforward way of doing this is to fit the model to obtain θ^*, then use the model's forward simulator to generate new data sets, in essence treating our original estimate θ^* as the "truth" set [[13]].

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 use forward simulations from the last estimated state to produce estimates of the future state.

Figure [5.3] shows a projection of the data from the previous plots in Figures [5.1] and [5.2].

We can define a metric to gauge overall forecast effectiveness by calculating the sum

---

Algorithm 5: Parametric Bootstrap

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

   /* Initial fit   */

1 $\theta^* \leftarrow IF2(D)$

   /* Generate artificial data sets   */

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

3 $\lfloor$ $D_i \leftarrow S(\theta^*)$

   /* Fit to new data sets   */

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

5 $\lfloor$ $\theta_i \leftarrow IF2(D_i)$

Output: Distribution samples $\theta_1, \theta_2, ..., \theta_M$
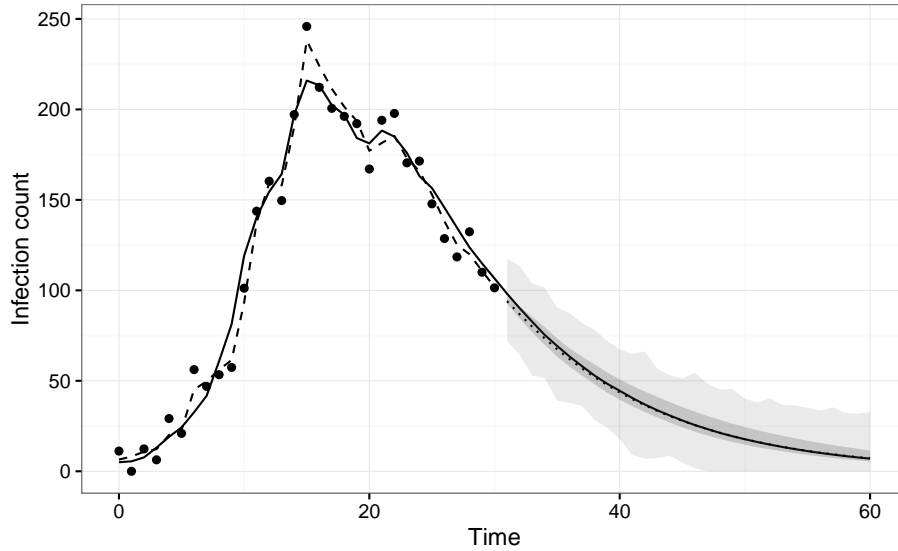
Figure 5.3: Forecast produced by the IF2 / parametric bootstrapping framework. The dotted line shows the mean estimate of the forecasts, the dark grey ribbon shows the 95% confidence interval based on the 0.025 and 0.975 quantiles on the true state estimates, and the lighter grey ribbon shows the same confidence interval on the true state estimates with added observation noise drawn from $\mathcal{N}(0, \sigma)$.

of squared errors of prediction (SSE). For the data in Figure [5.3] the value was approximately $SSE = 50.1$. Normally we would also want to address questions of forecast coverage, but this would require at least a 100-fold increase in computational cost. This is potentially an avenue of future investigation.

5.3   HMC

For HMC we can use a simpler approach to approach forecasting. 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 as described in Chapter 2. Once we've done this we can forward simulate the system from the state estimate into the future.

Figure [5.4] shows the result of the HMC forecasting framework as applied to the data from Figure [5.1].

And as before we can evaluate the SSE of the forecast for the data shown, giving approximately $SSE = 608$.
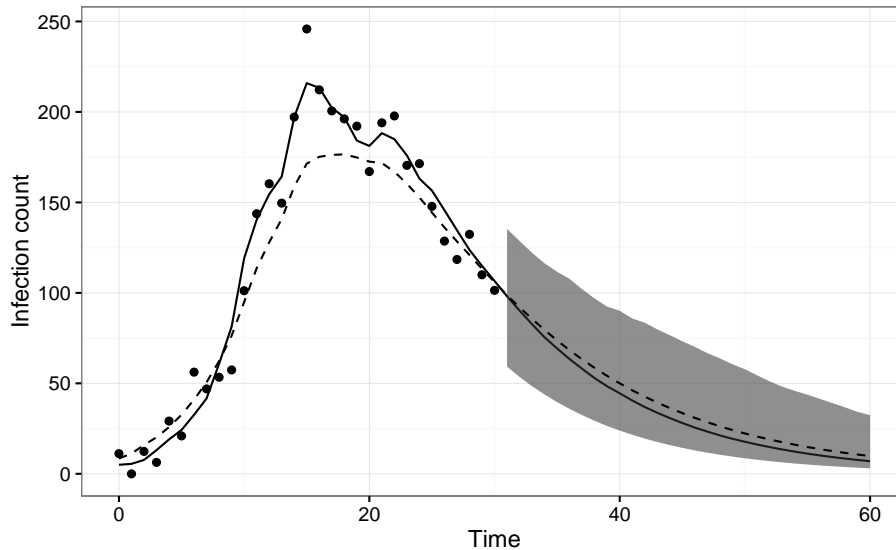
Figure 5.4: Forecast produced by the HMC / bootstrapping framework with $M = 200$ trajectories. The dotted line shows the mean estimate of the forecasts, and the grey ribbon shows the 95% confidence interval on the estimated true states as described in Figure [5.3].

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 HMC 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 HMC framework, about 35.5x as expensive. However the now much longer running time can somewhat alleviated by parallelizing the parametric bootstrapping process; as each of the parametric bootstrap fittings in entirely independent, this can be done without a great deal of additional effort. The code
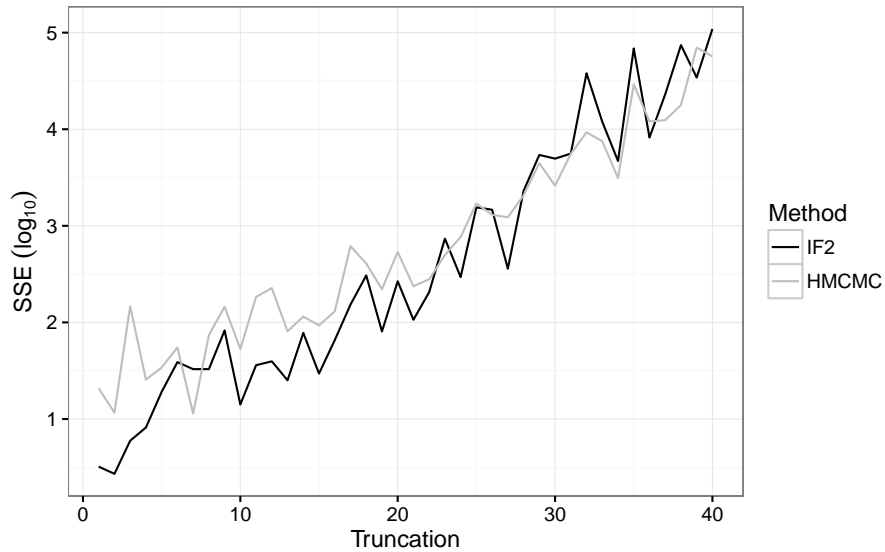
Figure 5.5: Error growth as a function of data truncation amount. Both methods used 200 bootstrap trajectories. Note that the y-axis shows the natural log of the averaged SSE, not the total SSE.

1 used here has this capability, but it was not utilised in the comparison so as to accu-
2 rately represent total computational cost, rather than 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 George Sugihara and collaborators [36][37][19][14] 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 estimation or 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 linear component of the method only comes into play when combining forecast components together to produce a single estimate

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 $|| \cdot ||$ is the Euclidean norm, i ranges over 1 to the length of the library, and j ranges over $[0, E]$. In the above equations and the ones that follow, we set $x_t(0) \equiv 1$ 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 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.

In essence what we are doing is generating a series of forecasts from every vector in the library, weighting those forecasts based on how similar the corresponding library vector is to our predictor vector, obtaining a solution to the system that maps components of a predictor vector to its library vector's forecasted point (the mapping), then applying that mapping to our predictor variable to obtain a forecast.

6.2   S-map Algorithm

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

Algorithm 6: S-map

```
/* Select a starting point                                             */
```
Input   : Time series $x_1, x_2, ..., x_T$, embedding dimension E, distance
             penalization θ, forecast length L, predictor vector $\mathbf{x_t}$

```
/* Construct library {xᵢ}                                              */
```
1 **for** $i = E : T$ **do**
2     $\mathbf{x_i} = (x_i, x_{i-1}, ..., x_{i-E-1})$

```
/* Construct mapping from library vectors to predictions               */
```
3 **for** $i = 1 : (T_E + 1)$ **do**
4   **for** $j = 1 : E$ **do**
5        $A(i, j) = w(||\mathbf{x_i} - \mathbf{x_t}||)\mathbf{x_i}(j)$

6 **for** $i = 1 : (T_E + 1)$ **do**
7     $b(i) = w(||\mathbf{x_i} - \mathbf{x_t}||)y_i$

```
/* Use SVD to solve the mapping system, Ac = b                         */
```
8 $SVD(Ac = b)$

```
/* Compute forecast                                                    */
```
9 $\hat{y}_t = \sum_{j=0}^{E} c_t(j)\mathbf{x_t}(j)$

```
/* 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.

Given, the S-map's data requrements, we need to specify a modified version of the SIR model. As IF2 and HMC in principle should be able operate on any reasonably well-specified model, the easiest way to compare the efficacy of S-mapping to IF2 or HMC 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, with one small addition. 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 rate or waning immunity η through which people become able to be reinfected, and a seasonality factor function $\Gamma(t)$ defined as

$$
\Gamma(t) = \exp\left[2\left(\cos\left(\frac{2\pi}{365}t\right) - 1\right)\right].
\tag{6.5}
$$

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

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

Figure [6.1] shows the SIRS model simulated for the equivalent of 5 years (260 weeks) and adding noise drawn from $\mathcal{N}(0, \sigma)$.
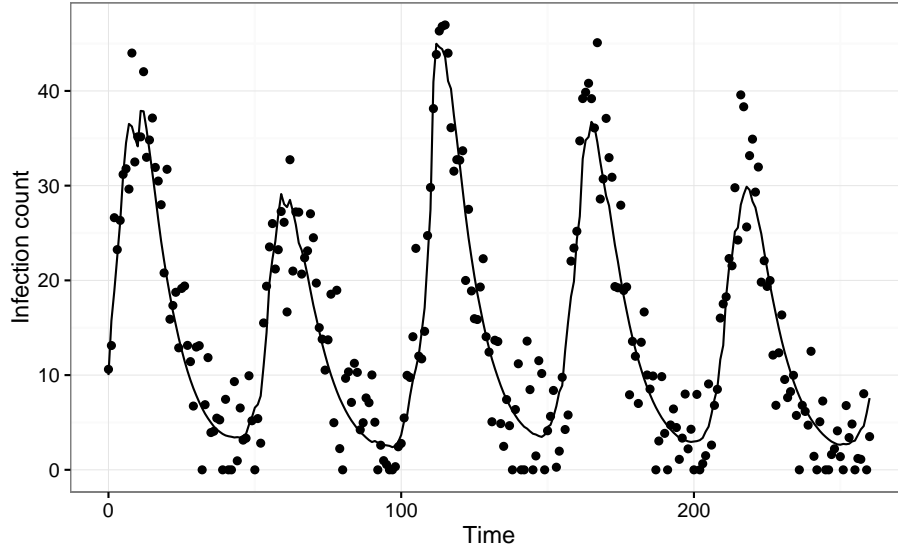
Figure 6.1: Five cycles generated by the SIRS function. The solid line the the true number of cases, dots show case counts with added observation noise. The parameter values were $R_0 = 3.0$, $\gamma = 0.1$, $\eta = 1$, $\sigma = 5$, and 10 initial cases.

1 Figure [6.2] shows how the S-map can reconstruct the next cycle in the time se-
2 ries.
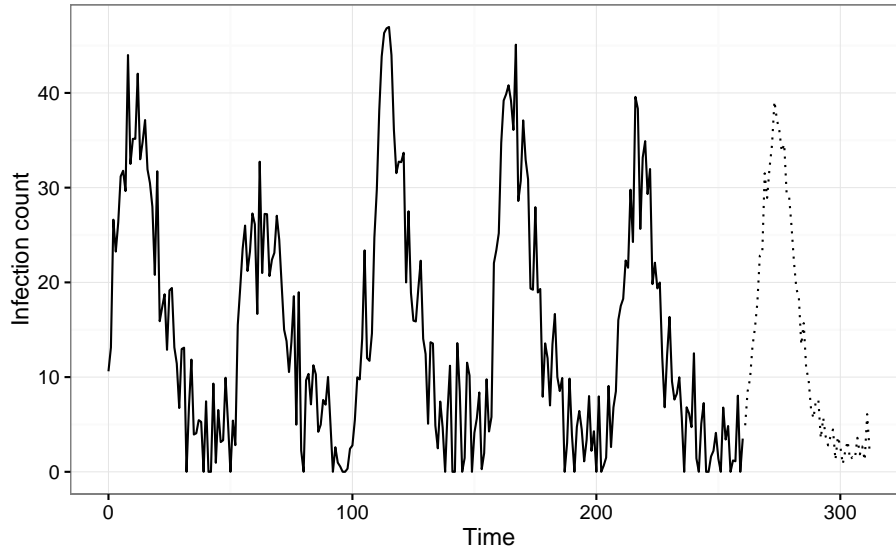
Figure 6.2: S-map applied to the data from the previous figure. The solid line shows the infection counts with observation noise from the previous plot, and the dotted line is the S-map forecast. Parameters chosen were $E = 14$ and $\theta = 3$.

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 [14]. The script to perform this optimisation procedure 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 HMC.

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.

Figure [6.3] shows the results of the simulation.

Interestingly, all methods produce roughly the same result, which is to say the spikes in each outbreak cycle are difficult to accurately predict. IF2 produces better results than either HMC and the S-map for the majority of forecast lengths, with the S-map
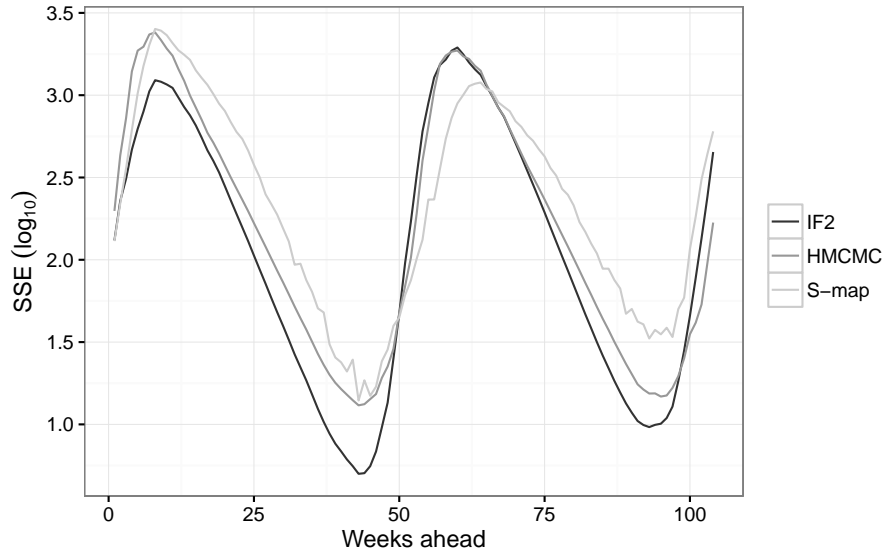
Figure 6.3: Error as a function of forecast length.

1 producing the poorest results with the exception of the second rise in infection rates
2 where it outperforms the other methods.

3 While the S-map may not provide the same fidelity or forecast as IF2 or HMC, it
4 shines when it comes to running time. Figure [6.4] shows the running times over 20
5 simulations.

6 It is clear from Figure [6.4] that the S-map running times are minute compared to the
7 other methods, but to emphasize the degree: The average running time for the S-map
8 is about 0.15 seconds, for IF2 it is about $47,000$, and for HMC it is about $9,200$.
9 This is a speed-up of over 316,000x compared to IF2 and over 61,800x compared to
10 HMC.

Figure 6.4: Runtimes for producing SIRS forecasts. The box shows the middle 50th percent, the bold line is the median, and the dots are outliers. Note that these are not "true" outliers, simply ones outside a threshold based on the interquartile range.

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

$$\frac{dS_i}{dt} = -\left(1 - \phi\frac{N_i}{N_i + 1}\right)\beta_i S_i I_i - \left(\frac{\phi}{N_i + 1}\right)S_i \sum_{j=0}^{N_i}\beta_j I_j$$

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

$$\frac{dR_i}{dt} = \gamma I,$$

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

As before we let β embark on a geometric random walk defined as
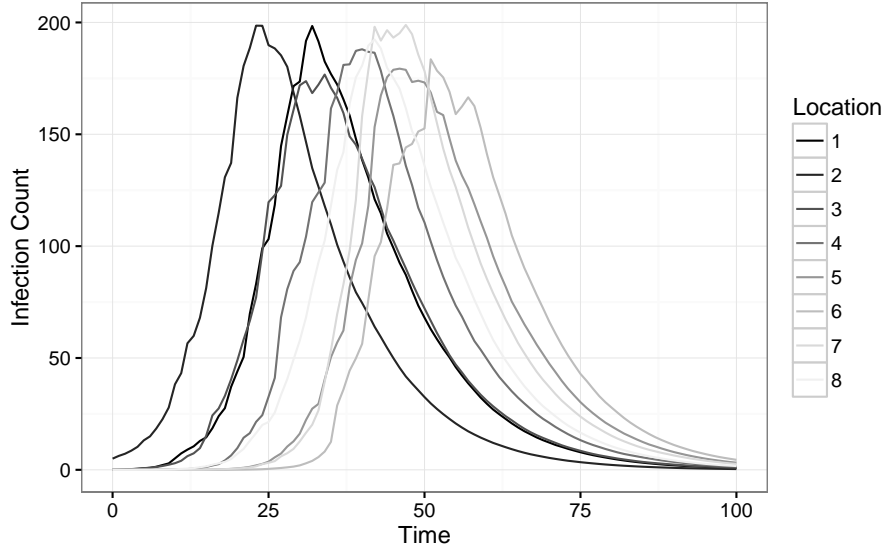
Figure 7.1: Evolution of a spatial epidemic in a ring topology. The outbreak was started with 5 cases in Location 2. Parameters were $R_0 = 3.0$, $\gamma = 0.1$, $\eta = 0.5$, $\sigma_{err} = 0.5$, and $\phi = 0.5$.

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

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 8 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].
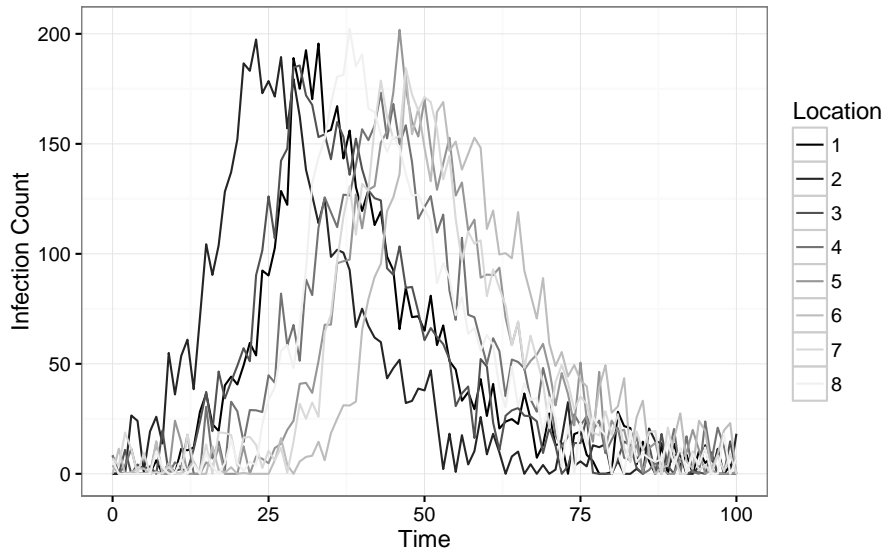
Figure 7.2: Evolution of a spatial epidemic as in Figure [7.1], with added observation noise drawn from $\mathcal{N}(0, 10)$.

7.2 Dewdrop Regression

Dewdrop regression [19] 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 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 HMC, 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 HMC 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 HMC. HMC 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 HMC were about $29,000$ seconds and $38,800$ seconds, respectively. This is a speed-up of just over 116x over IF2 and 156x over HMC.

Considering how well S-mapping performed with regards to forecast error, it shows a significant advantage over HMC in particular – it outperforms it in both forecast error and running times.
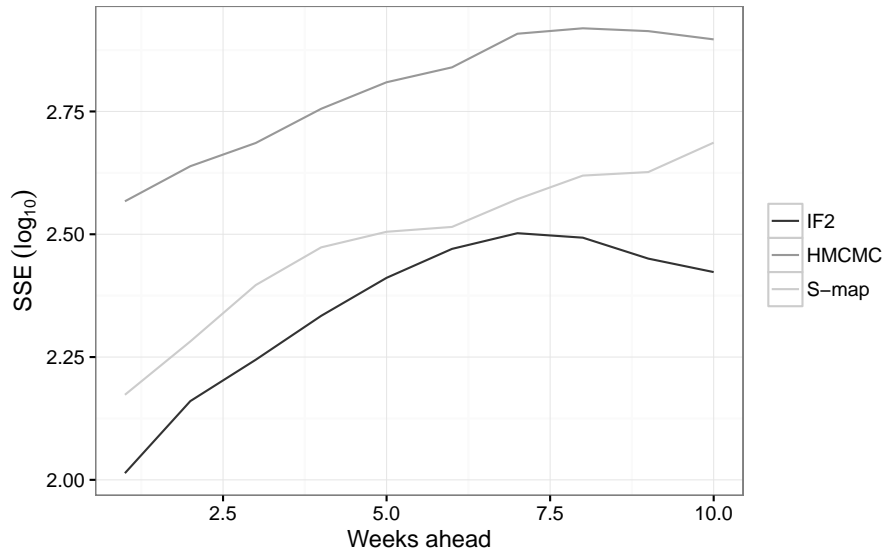
Figure 7.3: Average SSE (log scale) across each location and all trials as a function of the number of weeks ahead in the forecast.
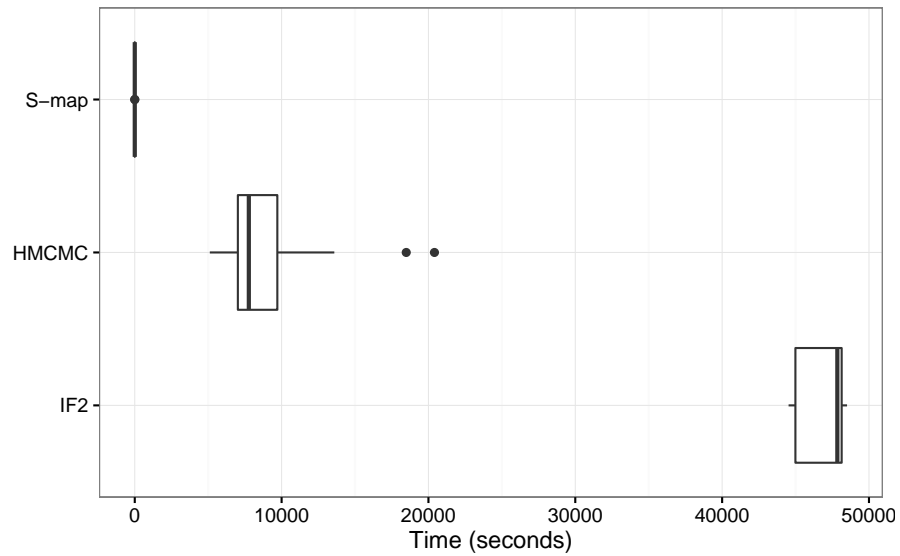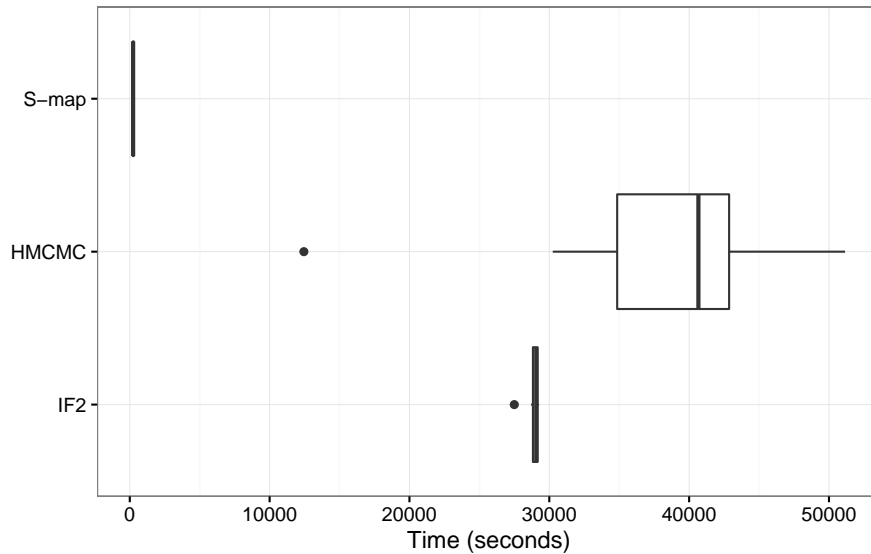

Figure 7.4: Runtimes for producing spatial SIR forecasts. The box shows the middle 50th percent, the bold line is the median, and the dots are outliers.

Chapter 8

Discussion and Future Directions

A summary of the results of forecasts in the previous three chapters follows.

Immediately, we can see that the IF2 / parametric bootstrapping framework produces great results. This framework consistently out-performs both the HMC framework and S-mapping by itself or with Dewdrop Regression. This is not to say that the results produced by the other methods are poor, but rather that the ones produced by IF2 are noticeably better. This is true in every scenario we have explored here, and is particularly pronounced in the SIRS and spatial forecasting set-ups.

A surprise has been how well S-mapping has performed. Given the almost ludicrously shorter running times exhibited by S-mapping, it is almost shocking how well it performs. In the SIRS scenario it produces results only slightly less accurate than the other two methods, and is even the most accurate at predicting the rise to the second outbreak peak. In the spatial scenario it performs almost as well as IF2, and much better than HMC. The critical point here is that S-mapping, with its relative ease of implementation, efficiency, and accuracy would make a great "first-blush" forecasting tool that could be run and give a good prediction well before one would be able to even code the model specification for either of the other two methods. While S-mapping does require an up-front computational cost in "tuning" the two algorithm parameters, it is still negligible when compared to the costs incurred by IF2 and HMC.

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 [39].

Recently, transistor count growth has begin to falter. This is due to several physical factors preventing tighter "packing" of transistors into a single processor. 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 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 one 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, and has become critical in taking full advantage of machines of all scales – from mobile phones which overwhelmingly favour multi-core CPU architectures, to large clusters and supercomputers which 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 HMC 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, which has the added benefit of providing data from which to assess convergence. 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 with HMC, it should be clear that each bootstrap trajectory is completely independent, and thus this component of the forecasting framework can be considered "embarrassingly" parallel or distributed. 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 highly 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. An 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. However, the code and 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 [23] 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, and would be another step towards establishing which tools are best for which jobs.

8.3 Fin

The overarching theme in this paper, from the theoretical considerations to the results to the discussion, is that there still exists no "silver bullet" for forecasting problems. Largely you can decide, as the user, how accurate you need your results to be, how much computer time you have at your disposal, and how fast you need your results, and select the method that best satisfies your needs. If speed is the priority, then you can use S-mapping to get very quick and relatively accurate results. If you require accuracy above all else, you must turn to heavier methods such as IF2, HMC, and parametric bootstrapping in order to produce the cleanest forecast possible. And this only represents three data points in a larger picture. There are a wide variety of methods that are similar but not identical to methods explored here, each with their own positive and negative attributes, their own advantages and disadvantages, and that are ultimately likely to fill out our spectrum of methods more completely. Thus future work should focus on attempting further direct comparison across a wider swath of techniques, and implementing those techniques in a parallel fashion to take advantage of the current and future landscape of high-performance computing.

Bibliography

[1] Mohammad Ali et al. "Time Series Analysis of Cholera in Matlab, Bangladesh, during 1988-2001". In: *Journal of health, population, . . .* 31.1 (2013), pp. 11–19. URL: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3702354/.

[2] Christophe Andrieu et al. "An introduction to MCMC for machine learning". In: *Machine Learning* 50.1-2 (2003), pp. 5–43. ISSN: 08856125. DOI: 10.1023/A:1020281327116. arXiv: 1109.4435v1.

[3] M.S. Arulampalam et al. "A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking". In: *IEEE Transactions on Signal Processing* 50.2 (2002), pp. 174–188. ISSN: 1053587X. DOI: 10.1109/78.978374. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=978374.

[4] Jacob Bock Axelsen et al. "Multiannual forecasting of seasonal influenza dynamics reveals climatic and evolutionary drivers." In: *Proceedings of the National Academy of Sciences of the United States of America* 111.26 (July 2014), pp. 9538–42. ISSN: 1091-6490. DOI: 10.1073/pnas.1321656111. URL: http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=4084473%7B%5C&%7Dtool=pmcentrez%7B%5C&%7Drendertype=abstract.

[5] M Babyak. "What You See May Not Be What You Get: A Brief, Nontechnical Introduction to Overfitting in Regression Type Models". In: *J. Bio. Med.* 66.3 (2004), pp. 411–421. ISSN: 0033-3174. DOI: 10.1097/01.psy.0000127692.23278.a9.

[6] Thomas Bengtsson, Peter Bickel, and Bo Li. "Curse-of-dimensionality revisited: Collapse of the particle filter in very large scale systems". In: *Probability and Statistics* 2 (2008), pp. 316–334. DOI: 10.1214/193940307000000518. arXiv: 0805.3034. URL: http://arxiv.org/abs/0805.3034.

[7] a. Camacho et al. "Explaining rapid reinfections in multiple-wave influenza outbreaks: Tristan da Cunha 1971 epidemic as a case study". In: *Proceedings of the Royal Society B: Biological Sciences* 278 (2011), pp. 3635–3643. ISSN: 0962-8452. DOI: 10.1098/rspb.2011.0300.

[8]   Bob Carpenter et al. "Stan: A Probabilistic Programming Language". In: *Journal of Statistical Software* (2016). URL: `http://www.stat.columbia.edu/%7B~%7Dgelman/research/unpublished/stan-resubmit-JSS1293.pdf`.

[9]   Jean-Paul Chretien et al. "Influenza forecasting in human populations: a scoping review." In: *PloS one* 9.4 (Jan. 2014), e94130. ISSN: 1932-6203. DOI: `10.1371/journal.pone.0094130`. URL: `http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3979760%7B%5C&%7Dtool=pmcentrez%7B%5C&%7Drendertype=abstract`.

[10] Samantha Cook et al. "Assessing Google Flu trends performance in the United States during the 2009 influenza virus A (H1N1) pandemic". In: *PLoS ONE* 6.8 (2011), pp. 1–8. ISSN: 19326203. DOI: `10.1371/journal.pone.0023610`.

[11] Andrea Freyer Dugas et al. "Influenza forecasting with Google Flu Trends." In: *PloS one* 8.2 (Jan. 2013), e56176. ISSN: 1932-6203. DOI: `10.1371/journal.pone.0056176`. URL: `http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3572967%7B%5C&%7Dtool=pmcentrez%7B%5C&%7Drendertype=abstract`.

[12] Dirk Eddelbuettel and Romain Fran. "Rcpp: Seamless R and C ++ Integration". In: *Journal Of Statistical Software* 40.8 (2011), pp. 1–18. ISSN: 15487660. DOI: `10.1007/978-1-4614-6868-4`. arXiv: `arXiv:1011.1669v3`. URL: `http://www.jstatsoft.org/v40/i08/`.

[13] Christian Genest and Bruno Remillard. "Validity of the parametric bootstrap for goodness-of-fit testing in semiparametric models". In: *Annales de l'institut Henri Poincare (B) Probability and Statistics* 44.6 (2008), pp. 1096–1127. ISSN: 02460203. DOI: `10.1214/07-AIHP148`.

[14] Sarah M. Glaser, Hao Ye, and George Sugihara. "A nonlinear, low data requirement model for producing spatially explicit fishery forecasts". In: *Fisheries Oceanography* 23 (2014), pp. 45–53. ISSN: 10546006. DOI: `10.1111/fog.12042`.

[15] Andrea L Graham et al. "Explaining rapid reinfections in multiple-wave influenza outbreaks : Tristan da Cunha 1971 epidemic as a case study". In: *Proc. R. Soc. B* (2016). DOI: `10.1098/rspb.2011.0300`.

[16] Bradley Harding. "Standard errors : A review and evaluation of standard error estimators using Monte Carlo simulations". In: (2014), pp. 107–123.

[17] Florian Hartig et al. "Statistical inference for stochastic simulation models - theory and application". In: *Ecology Letters* 14.8 (2011), pp. 816–827. ISSN: 1461023X. DOI: `10.1111/j.1461-0248.2011.01640.x`.

[18] Matthew D. Hoffman and Andrew Gelman. "The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo". In: *Journal of Machine Learning Research* 15.April (2014), pp. 1593–1623. arXiv: `1111.4246`. URL: `http://mc-stan.org/`.

[19] Chih-hao Hsieh, Christian Anderson, and George Sugihara. "Extending nonlinear analysis to short ecological time series." In: *The American naturalist* 171.1 (2008), pp. 71–80. ISSN: 0003-0147. DOI: 10.1086/524202.

[20] E L Ionides, C Bret, and A. A. King. "Inference for nonlinear dynamical systems." In: *Proceedings of the National Academy of Sciences of the United States of America* 103.49 (Dec. 2006), pp. 18438–43. ISSN: 0027-8424. DOI: 10.1073/pnas.0603181103. URL: http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3020138%7B%5C&%7Dtool=pmcentrez%7B%5C&%7Drendertype=abstract.

[21] Edward L. Ionides et al. "Inference for dynamic and latent variable models via iterated, perturbed Bayes maps". In: *Proceedings of the National Academy of Sciences* 112.3 (2015), pp. 719–724. ISSN: 0027-8424. DOI: 10.1073/pnas.1410597112. URL: http://www.pnas.org/lookup/doi/10.1073/pnas.1410597112.

[22] Aaron A King, Dao Nguyen, and Edward L. Ionides. "Statistical Inference for Partially Observed Markov Processes via the R Package pomp". In: *Journal of Statistical Software* 59.10 (2015). arXiv: arXiv:1509.00503v1.

[23] Aaron A King et al. "Avoidable errors in the modelling of outbreaks of emerging pathogens, with special reference to Ebola". In: *Proceedings of the Royal Society B: Biological Sciences* 282.1806 (2015), pp. 20150347–20150347. ISSN: 0962-8452. DOI: 10.1098/rspb.2015.0347. arXiv: 1412.0968. URL: http://rspb.royalsocietypublishing.org/cgi/doi/10.1098/rspb.2015.0347.

[24] Aaron A King et al. *pomp: Statistical inference for partially observed Markov processes*. 2016. URL: http://kingaa.github.io/pomp.

[25] William A. Link and Mitchell J. Eaton. "On thinning of chains in MCMC". In: *Methods in Ecology and Evolution* 3.1 (2012), pp. 112–115. ISSN: 2041210X. DOI: 10.1111/j.2041-210X.2011.00131.x.

[26] Kanti V. Mardia et al. "The Kriged Kalman filter". In: *Test* 7.2 (1998), pp. 217–282. ISSN: 11330686. DOI: 10.1007/BF02565111.

[27] Radford M Neal. "Handbook of Markov Chain Monte Carlo". In: *Handbook of Markov Chain Monte Carlo* 20116022 (2011), pp. 113–162. DOI: 10.1201/b10905. arXiv: arXiv:1206.1901v1. URL: http://www.crcnetbase.com/doi/book/10.1201/b10905.

[28] Elaine O Nsoesie et al. "A systematic review of studies on forecasting the dynamics of influenza outbreaks." In: *Influenza and other respiratory viruses* 8.3 (May 2014), pp. 309–16. ISSN: 1750-2659. DOI: 10.1111/irv.12226. URL: http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=4181479%7B%5C&%7Dtool=pmcentrez%7B%5C&%7Drendertype=abstract.

[29] Elaine Nsoesie, Madhav Mararthe, and John Brownstein. "Forecasting peaks of seasonal influenza epidemics." In: *PLoS currents* 5 (Jan. 2013), pp. 1–14. ISSN: 2157-3999. DOI: `10.1371/currents.outbreaks.bb1e879a23137022ea79a8c508b030bc`. URL: `http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3712489%7B%5C&%7Dtool=pmcentrez%7B%5C&%7Drendertype=abstract`.

[30] Robert C Reiner et al. "Highly localized sensitivity to climate forcing drives endemic cholera in a megacity." In: *Proceedings of the National Academy of Sciences of the United States of America* 109.6 (Feb. 2012), pp. 2033–6. ISSN: 1091-6490. DOI: `10.1073/pnas.1108438109`. URL: `http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3277579%7B%5C&%7Dtool=pmcentrez%7B%5C&%7Drendertype=abstract`.

[31] Sujit K Sahu. "A Bayesian Kriged-Kalman model for short-term fore- casting of air pollution levels". In: *Appl. Statist* 54 (2005), pp. 223–244. ISSN: 0035-9254. DOI: `10.1111/j.1467-9876.2005.00480.x`.

[32] Jacques Sau et al. "Particle filter-based real-time estimation and prediction of traffic conditions Modeling framework". In: *Information Systems* (1918), pp. 1–8.

[33] Jeffrey Shaman and Alicia Karspeck. "Forecasting seasonal outbreaks of influenza." In: *Proceedings of the National Academy of Sciences of the United States of America* 109.50 (Dec. 2012), pp. 20425–30. ISSN: 1091-6490. DOI: `10.1073/pnas.1208772109`. URL: `http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3528592%7B%5C&%7Dtool=pmcentrez%7B%5C&%7Drendertype=abstract`.

[34] Jeffrey Shaman, Wan Yang, and Sasikiran Kandula. "Inference and Forecast of the Current West African Ebola Outbreak in Guinea, Sierra Leone and Liberia". In: *PLoS Currents* 6 (2014), ecurrents.outbreaks.3408774290b1a0f2dd7cae877c8b8f. ISSN: 2157-3999. DOI: `10.1371/currents.outbreaks.3408774290b1a0f2dd7cae877c8b8ff6`.

[35] Stan Development Team. "Stan Modeling Language Users Guide and Reference Manual". In: 2.9.0 (2015).

[36] G Sugihara and R M May. *Nonlinear forecasting as a way of distinguishing chaos from measurement error in time series.* 1990. DOI: `10.1038/344734a0`.

[37] George Sugihara. "Nonlinear Forecasting for the Classification of Natural Time Series". In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 348 (1994), pp. 477–495. ISSN: 1364-503X. DOI: `10.1098/rsta.1994.0106`.

[38] Carmen L Vidal Rodeiro and Andrew B Lawson. "Online updating of space-time disease surveillance models via particle filters." In: *Statistical methods in medical research* 15.5 (2006), pp. 423–444. ISSN: 0962-2802. DOI: `10.1177/0962280206071640`.

[39] Mitchell Waldrop. "More then Moore". In: *Nature* 530.11. February (2016), p. 145.

[40] Wan Yang, Alicia Karspeck, and Jeffrey Shaman. "Comparison of filtering methods for the modeling and retrospective forecasting of influenza epidemics." In: *PLoS computational biology* 10.4 (Apr. 2014), e1003583. ISSN: 1553-7358. DOI: `10.1371/journal.pcbi.1003583`. URL: `http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3998879%7B%5C&%7Dtool=pmcentrez%7B%5C&%7Drendertype=abstract`.

[41] Xingyu Zhang et al. "Comparative study of four time series methods in forecasting typhoid fever incidence in China." In: *PloS one* 8.5 (Jan. 2013), e63116. ISSN: 1932-6203. DOI: `10.1371/journal.pone.0063116`. URL: `http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3641111%7B%5C&%7Dtool=pmcentrez%7B%5C&%7Drendertype=abstract`.

1 Appendix A

2 Hamiltonian MCMC

3 A.1   Full R code

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

```r
## Dexter Barrows
## dbarrows.github.io
## 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)))

    })

}
```

```
30
31 pars ← c(R0 ← 3.0,     # average number of new infected individuals
       per infectious person
32             r  ← 0.1,    # recovery rate
33             N  ← 500)    # population size
34
35 T ← 100
36 y_ini ← c(S = 495, I = 5, R = 0)
37 times ← seq(0, T, by = 1)
38
39 odeout ← ode(y_ini, times, SIR, pars)
40
41 set.seed(1001)
42 sigma ← 10
43 infec_counts_raw ← odeout[,3] + rnorm(T+1, 0, sigma)
44 infec_counts      ← ifelse(infec_counts_raw < 0, 0, infec_counts_raw)
45
46 g ← qplot(0:T, odeout[,3], geom = "line", xlab = "Time (weeks)", ylab
       = "Infection Count") +
       geom_point(aes(y = infec_counts)) +
47     theme_bw()
48
49
50 print(g)
51 ggsave(g, filename="dataplot.pdf", height=4, width=6.5)
52
53 sPw ← 7
54 datlen ← (T-1)*7 + 1
55
56 data ← matrix(data = -1, nrow = T+1, ncol = sPw)
57 data[,1] ← infec_counts
58 standata ← as.vector(t(data))[1:datlen]
59
60 sir_data ← list( T = datlen,   # simulation time
61                  y = standata, # infection count data
62                  N = 500,      # population size
63                  h = 1/sPw )   # step size per day
64
65 rstan_options(auto_write = TRUE)
66 options(mc.cores = parallel::detectCores())
67 stan_options ← list(   chains = 4,    # number of chains
68                        iter   = 2000, # iterations per chain
69                        warmup = 1000, # warmup interations
70                        thin   = 2)    # thinning number
71 fit ← stan(file    = "d_sirode_euler.stan",
72            data    = sir_data,
73            chains  = stan_options$chains,
74            iter    = stan_options$iter,
75            warmup  = stan_options$warmup,
76            thin    = stan_options$thin )
77
78 exfit ← extract(fit, permuted = TRUE, inc_warmup = FALSE)
```

```
79
80  R0points ← exfit$R0
81  R0kernel ← qplot(R0points, geom = "density", xlab = expression(R[0]),
        ylab = "frequency") +
82          geom_vline(aes(xintercept=R0), linetype="dashed", size=1,
                color="grey50") +
83          theme_bw()
84
85  print(R0kernel)
86  ggsave(R0kernel, filename="kernelR0.pdf", height=3, width=3.25)
87
88  rpoints ← exfit$r
89  rkernel ← qplot(rpoints, geom = "density", xlab = "r", ylab = "
        frequency") +
90          geom_vline(aes(xintercept=r), linetype="dashed", size=1,
                color="grey50") +
91          theme_bw()
92
93  print(rkernel)
94  ggsave(rkernel, filename="kernelr.pdf", height=3, width=3.25)
95
96  sigmapoints ← exfit$sigma
97  sigmakernel ← qplot(sigmapoints, geom = "density", xlab = expression(
        sigma), ylab = "frequency") +
98          geom_vline(aes(xintercept=sigma), linetype="dashed", size=1,
                color="grey50") +
99          theme_bw()
100
101 print(sigmakernel)
102 ggsave(sigmakernel, filename="kernelsigma.pdf", height=3, width=3.25)
103
104 infecpoints ← exfit$y0[,2]
105 infeckernel ← qplot(infecpoints, geom = "density", xlab = "Initial
        Infected", ylab = "frequency") +
106          geom_vline(aes(xintercept=y_ini[['I']]), linetype="dashed",
                size=1, color="grey50") +
107          theme_bw()
108
109 print(infeckernel)
110 ggsave(infeckernel, filename="kernelinfec.pdf", height=3, width=3.25)
111
112 exfit ← extract(fit, permuted = FALSE, inc_warmup = FALSE)
113 plotdata ← melt(exfit[,,"R0"])
114 tracefitR0 ← ggplot() +
115          geom_line(data = plotdata,
116                       aes(x = iterations,
117                       y = value,
118                       color = factor(chains, labels = 1:stan_
                          options$chains))) +
119          labs(x = "Sample", y = expression(R[0]), color = "Chain
                    ") +
```

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

A.2   Full Stan code

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

```
1   ## Dexter Barrows
2   ## dbarrows.github.io
3   ## McMaster University
4   ## 2016
5
6   data {
7
8       int     <lower=1>   T;      // total integration steps
9       real                y[T];   // observed number of cases
10      int     <lower=1>   N;      // population size
11      real                h;      // step size
12
13  }
14
15  parameters {
16
17      real <lower=0, upper=10>   R0;    // R0
18      real <lower=0, upper=10>   r;     // recovery rate
```

72

```
19        real <lower=0, upper=20>    sigma;  // observation error
20        real <lower=0, upper=500>   y0[3];  // initial conditions
21
22  }
23
24  model {
25
26        real S[T];
27        real I[T];
28        real R[T];
29
30        S[1] <- y0[1];
31        I[1] <- y0[2];
32        R[1] <- y0[3];
33
34        y[1] ~ normal(y0[2], sigma);
35
36        for (t in 2:T) {
37
38            S[t] <- S[t-1] + h*( - S[t-1]*I[t-1]*R0*r/N );
39            I[t] <- I[t-1] + h*( S[t-1]*I[t-1]*R0*r/N  - I[t-1]*r );
40            R[t] <- R[t-1] + h*( I[t-1]*r );
41
42            if (y[t] > 0) {
43                y[t] ~ normal( I[t], sigma );
44            }
45
46        }
47
48        y0[1] ~ normal(N - y[1], sigma);
49        y0[2] ~ normal(y[1], sigma);
50
51        R0      ~ lognormal(1,1);
52        r       ~ lognormal(1,1);
53        sigma   ~ lognormal(1,1);
54
55  }
```

1 Appendix B

2 Iterated Filtering

3 B.1   Full R code

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

```
1  ## Dexter Barrows
2  ## dbarrows.github.io
3  ## McMaster University
4  ## 2016
5
6  library(deSolve)
7  library(ggplot2)
8  library(reshape2)
9  library(gridExtra)
10 library(Rcpp)
11
12 SIR ← function(Time, State, Pars) {
13
14     with(as.list(c(State, Pars)), {
15
16         B   ← R0*r/N
17         BSI ← B*S*I
18         rI  ← r*I
19
20         dS = -BSI
21         dI = BSI - rI
22         dR = rI
23
24         return(list(c(dS, dI, dR)))
25
26     })
27
28 }
29
```

74

```
30 T      ← 100
31 N      ← 500
32 sigma   ← 10
33 i_infec ← 5
34
35 ## Generate true trajecory and synthetic data
36 ##
37
38 true_init_cond ← c(S = N - i_infec,
39                       I = i_infec,
40                        R = 0)
41
42 true_pars ← c(R0 = 3.0,
43                 r = 0.1,
44                 N = 500.0)
45
46 odeout ← ode(true_init_cond, 0:T, SIR, true_pars)
47 trueTraj ← odeout[,3]
48
49 set.seed(1001)
50
51 infec_counts_raw ← odeout[,3] + rnorm(T+1, 0, sigma)
52 infec_counts     ← ifelse(infec_counts_raw < 0, 0, infec_counts_raw)
53
54 g ← qplot(0:T, odeout[,3], geom = "line", xlab = "Time (weeks)", ylab
       = "Infection Count") +
55       geom_point(aes(y = infec_counts)) +
56       theme_bw()
57
58 print(g)
59 ggsave(g, filename="dataplot.pdf", height=4, width=6.5)
60
61 ## Rcpp stuff
62 ##
63
64 sourceCpp(paste(getwd(),"d_if2.cpp",sep="/"))
65
66 paramdata ← data.frame(if2(infec_counts, T+1, N))
67 colnames(paramdata) ← c("R0", "r", "sigma", "Sinit", "Iinit", "Rinit"
      )
68
69 ## Parameter density kernels
70 ##
71
72 R0points ← paramdata$R0
73 R0kernel ← qplot(R0points, geom = "density", xlab = expression(R[0]),
       ylab = "frequency") +
74          geom_vline(aes(xintercept=true_pars[["R0"]]), linetype="
              dashed", size=1, color="grey50") +
75          theme_bw()
76
```

```
77 print(R0kernel)
78 ggsave(R0kernel, filename="kernelR0.pdf", height=3, width=3.25)
79
80 rpoints ← paramdata$r
81 rkernel ← qplot(rpoints, geom = "density", xlab = "r", ylab = "
      frequency") +
82          geom_vline(aes(xintercept=true_pars[["r"]]), linetype="dashed
               ", size=1, color="grey50") +
83          theme_bw()
84
85 print(rkernel)
86 ggsave(rkernel, filename="kernelr.pdf", height=3, width=3.25)
87
88 sigmapoints ← paramdata$sigma
89 sigmakernel ← qplot(sigmapoints, geom = "density", xlab = expression(
      sigma), ylab = "frequency") +
90          geom_vline(aes(xintercept=sigma), linetype="dashed", size=1,
               color="grey50") +
91          theme_bw()
92
93 print(sigmakernel)
94 ggsave(sigmakernel, filename="kernelsigma.pdf", height=3, width=3.25)
95
96 infecpoints ← paramdata$Iinit
97 infeckernel ← qplot(infecpoints, geom = "density", xlab = "Initial
      Infected", ylab = "frequency") +
98          geom_vline(aes(xintercept=true_init_cond[['I']]), linetype="
               dashed", size=1, color="grey50") +
99          theme_bw()
100
101 print(infeckernel)
102 ggsave(infeckernel, filename="kernelinfec.pdf", height=3, width=3.25)
103
104 # show grid
105 grid.arrange(R0kernel, rkernel, sigmakernel, infeckernel, ncol = 2,
      nrow = 2)
106
107 pdf("if2kernels.pdf", height = 6.5, width = 6.5)
108 grid.arrange(R0kernel, rkernel, sigmakernel, infeckernel, ncol = 2,
      nrow = 2)
109 dev.off()
```

B.2   Full C++ code

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

```
1 /*  Dexter Barrows
2     dbarrows.github.io
```

```
 1    3      McMaster University
 2    4      2016
 3    5
 4    6      */
 5    7
 6    8  #include <stdio.h>
 7    9  #include <math.h>
 8   10  #include <sys/time.h>
 9   11  #include <time.h>
10   12  #include <stdlib.h>
11   13  #include <string>
12   14  #include <cmath>
13   15  #include <cstdlib>
14   16  #include <fstream>
15   17
16   18  #define Treal   100           // time to simulate over
17   19  #define R0true  3.0           // infectiousness
18   20  #define rtrue   0.1           // recovery rate
19   21  #define Nreal   500.0         // population size
20   22  #define merr    10.0          // expected measurement error
21   23  #define I0      5.0           // Initial infected individuals
22   24
23   25  #include <Rcpp.h>
24   26  using namespace Rcpp;
25   27
26   28  struct Particle {
27   29      double R0;
28   30      double r;
29   31      double sigma;
30   32      double S;
31   33      double I;
32   34      double R;
33   35      double Sinit;
34   36      double Iinit;
35   37      double Rinit;
36   38  };
37   39
38   40  struct ParticleInfo {
39   41      double R0mean;       double R0sd;
40   42      double rmean;        double rsd;
41   43      double sigmamean;    double sigmasd;
42   44      double Sinitmean;    double Sinitsd;
43   45      double Iinitmean;    double Iinitsd;
44   46      double Rinitmean;    double Rinitsd;
45   47  };
46   48
47   49
48   50  int timeval_subtract (double *result, struct timeval *x, struct
49          timeval *y);
50   51  int check_double(double x,double y);
51   52  void exp_euler_SIR(double h, double t0, double tn, int N, Particle *
```

```
        particle);
53 void copyParticle(Particle * dst, Particle * src);
54 void perturbParticles(Particle * particles, int N, int NP, int
       passnum, double coolrate);
55 bool isCollapsed(Particle * particles, int NP);
56 void particleDiagnostics(ParticleInfo * partInfo, Particle *
       particles, int NP);
57 NumericMatrix if2(NumericVector * data, int T, int N);
58 double randu();
59 double randn();
60
61 // [[Rcpp::export]]
62 NumericMatrix if2(NumericVector data, int T, int N) {
63
64     int     NP           = 2500;
65     int     nPasses      = 50;
66     double  coolrate     = 0.975;
67
68     int     i_infec      = I0;
69
70     NumericMatrix paramdata(NP, 6);
71
72     srand(time(NULL));          // Seed PRNG with system time
73
74     double w[NP];               // particle weights
75
76     Particle particles[NP];     // particle estimates for current
           step
77     Particle particles_old[NP]; // intermediate particle states for
           resampling
78
79     printf("Initializing particle states\n");
80
81     // initialize particle parameter states (seeding)
82     for (int n = 0; n < NP; n++) {
83
84         double R0can, rcan, sigmacan, Iinitcan;
85
86         do {
87             R0can = R0true + R0true*randn();
88         } while (R0can < 0);
89         particles[n].R0 = R0can;
90
91         do {
92             rcan = rtrue + rtrue*randn();
93         } while (rcan < 0);
94         particles[n].r = rcan;
95
96         do {
97             sigmacan = merr + merr*randn();
98         } while (sigmacan < 0);
```

```
99              particles[n].sigma = sigmacan;
100
101             do {
102                 Iinitcan = i_infec + i_infec*randn();
103             } while (Iinitcan < 0 || N < Iinitcan);
104             particles[n].Sinit = N - Iinitcan;
105             particles[n].Iinit = Iinitcan;
106             particles[n].Rinit = 0.0;
107
108     }
109
110     // START PASSES THROUGH DATA
111
112     printf("Starting filter\n");
113     printf("---------------\n");
114     printf("Pass\n");
115
116
117     for (int pass = 0; pass < nPasses; pass++) {
118
119         printf("...%d / %d\n", pass, nPasses);
120
121         perturbParticles(particles, N, NP, pass, coolrate);
122
123         // initialize particle system states
124         for (int n = 0; n < NP; n++) {
125
126             particles[n].S = particles[n].Sinit;
127             particles[n].I = particles[n].Iinit;
128             particles[n].R = particles[n].Rinit;
129
130         }
131
132         // between-pass perturbations
133
134         for (int t = 1; t < T; t++) {
135
136             // between-iteration perturbations
137             perturbParticles(particles, N, NP, pass, coolrate);
138
139             // generate individual predictions and weight
140             for (int n = 0; n < NP; n++) {
141
142                 exp_euler_SIR(1.0/10.0, 0.0, 1.0, N, &particles[n]);
143
144                 double merr_par = particles[n].sigma;
145                 double y_diff   = data[t] - particles[n].I;
146
147                 w[n] = 1.0/(merr_par*sqrt(2.0*M_PI)) * exp( - y_diff*
                        y_diff / (2.0*merr_par*merr_par) );
148
```

```
149                     }
150
151                     // cumulative sum
152                     for (int n = 1; n < NP; n++) {
153                         w[n] += w[n-1];
154                     }
155
156                     // save particle states to resample from
157                     for (int n = 0; n < NP; n++){
158                         copyParticle(&particles_old[n], &particles[n]);
159                     }
160
161                     // resampling
162                     for (int n = 0; n < NP; n++) {
163
164                         double w_r = randu() * w[NP-1];
165                         int i = 0;
166                         while (w_r > w[i]) {
167                             i++;
168                         }
169
170                         // i is now the index to copy state from
171                         copyParticle(&particles[n], &particles_old[i]);
172
173                     }
174
175                 }
176
177             }
178
179         ParticleInfo pInfo;
180         particleDiagnostics(&pInfo, particles, NP);
181
182         printf("Parameter results (mean | sd)\n");
183         printf("-----------------------------\n");
184         printf("R0        %f %f\n", pInfo.R0mean, pInfo.R0sd);
185         printf("r         %f %f\n", pInfo.rmean, pInfo.rsd);
186         printf("sigma     %f %f\n", pInfo.sigmamean, pInfo.sigmasd);
187         printf("S_init  %f %f\n", pInfo.Sinitmean, pInfo.Sinitsd);
188         printf("I_init    %f %f\n", pInfo.Iinitmean, pInfo.Iinitsd);
189         printf("R_init    %f %f\n", pInfo.Rinitmean, pInfo.Rinitsd);
190
191         printf("\n");
192
193
194
195         // Get particle results to pass back to R
196
197         for (int n = 0; n < NP; n++) {
198
199             paramdata(n, 0) = particles[n].R0;
```

```
200          paramdata(n, 1) = particles[n].r;
201          paramdata(n, 2) = particles[n].sigma;
202          paramdata(n, 3) = particles[n].Sinit;
203          paramdata(n, 4) = particles[n].Iinit;
204          paramdata(n, 5) = particles[n].Rinit;
205
206      }
207
208      return paramdata;
209
210 }
211
212
213 /*  Use the Explicit Euler integration scheme to integrate SIR model
        forward in time
214      double h    - time step size
215      double t0   - start time
216      double tn   - stop time
217      double * y  - current system state; a three-component vector
            representing [S I R], susceptible-infected-recovered
218
219      */
220 void exp_euler_SIR(double h, double t0, double tn, int N, Particle *
        particle) {
221
222      int num_steps = floor( (tn-t0) / h );
223
224      double S = particle->S;
225      double I = particle->I;
226      double R = particle->R;
227
228      double R0   = particle->R0;
229      double r    = particle->r;
230      double B    = R0 * r / N;
231
232      for(int i = 0; i < num_steps; i++) {
233          // get derivatives
234          double dS = - B*S*I;
235          double dI = B*S*I - r*I;
236          double dR = r*I;
237          // step forward by h
238          S += h*dS;
239          I += h*dI;
240          R += h*dR;
241      }
242
243      particle->S = S;
244      particle->I = I;
245      particle->R = R;
246
247 }
```

81

```
248
249
/*  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) {
```

```
297      dst->R0      = src->R0;
298      dst->r       = src->r;
299      dst->sigma   = src->sigma;
300      dst->S       = src->S;
301      dst->I       = src->I;
302      dst->R       = src->R;
303      dst->Sinit   = src->Sinit;
304      dst->Iinit   = src->Iinit;
305      dst->Rinit   = src->Rinit;
306
307  }


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

314      */
315  bool isCollapsed(Particle * particles, int NP) {
316
317      bool retVal;
318
319      double R0mean = 0, rmean = 0, sigmamean = 0, Sinitmean = 0,
320          Iinitmean = 0, Rinitmean = 0;
320
321      // means
322
323      for (int n = 0; n < NP; n++) {
324
325          R0mean       += particles[n].R0;
326          rmean        += particles[n].r;
327          sigmamean    += particles[n].sigma;
328          Sinitmean    += particles[n].Sinit;
329          Iinitmean    += particles[n].Iinit;
330          Rinitmean    += particles[n].Rinit;
331
332      }
333
334      R0mean       /= NP;
335      rmean        /= NP;
336      sigmamean    /= NP;
337      Sinitmean    /= NP;
338      Iinitmean    /= NP;
339      Rinitmean    /= NP;
340
341      double  R0sd = 0, rsd = 0, sigmasd = 0, Sinitsd = 0, Iinitsd = 0,
342          Rinitsd = 0;
342
343      for (int n = 0; n < NP; n++) {
```

```
344
345          R0sd     += ( particles[n].R0 - R0mean ) * ( particles[n].R0 -
                   R0mean );
346          rsd      += ( particles[n].r - rmean ) * ( particles[n].r -
                  rmean );
347          sigmasd += ( particles[n].sigma - sigmamean ) * ( particles[n
                  ].sigma - sigmamean );
348          Sinitsd += ( particles[n].Sinit - Sinitmean ) * ( particles[n
                  ].Sinit - Sinitmean );
349          Iinitsd += ( particles[n].Iinit - Iinitmean ) * ( particles[n
                  ].Iinit - Iinitmean );
350          Rinitsd += ( particles[n].Rinit - Rinitmean ) * ( particles[n
                  ].Rinit - Rinitmean );
351
352      }
353
354      R0sd        /= NP;
355      rsd         /= NP;
356      sigmasd     /= NP;
357      Sinitsd     /= NP;
358      Iinitsd     /= NP;
359      Rinitsd     /= NP;
360
361      if ( (R0sd + rsd + sigmasd) < 1e-5)
362          retVal = true;
363      else
364          retVal = false;
365
366      return retVal;
367
368 }
369
370 void particleDiagnostics(ParticleInfo * partInfo, Particle *
        particles, int NP) {
371
372      double   R0mean       = 0.0,
373               rmean        = 0.0,
374               sigmamean    = 0.0,
375               Sinitmean    = 0.0,
376               Iinitmean    = 0.0,
377               Rinitmean    = 0.0;
378
379      // means
380
381      for (int n = 0; n < NP; n++) {
382
383          R0mean       += particles[n].R0;
384          rmean        += particles[n].r;
385          sigmamean    += particles[n].sigma;
386          Sinitmean    += particles[n].Sinit;
387          Iinitmean    += particles[n].Iinit;
```

```
388            Rinitmean    += particles[n].Rinit;
389
390        }
391
392        R0mean       /= NP;
393        rmean        /= NP;
394        sigmamean    /= NP;
395        Sinitmean    /= NP;
396        Iinitmean    /= NP;
397        Rinitmean    /= NP;
398
399        // standard deviations
400
401        double  R0sd    = 0.0,
402                rsd     = 0.0,
403                sigmasd = 0.0,
404                Sinitsd = 0.0,
405                Iinitsd = 0.0,
406                Rinitsd = 0.0;
407
408        for (int n = 0; n < NP; n++) {
409
410            R0sd    += ( particles[n].R0 - R0mean ) * ( particles[n].R0 -
                    R0mean );
411            rsd     += ( particles[n].r - rmean ) * ( particles[n].r -
                    rmean );
412            sigmasd += ( particles[n].sigma - sigmamean ) * ( particles[n
                    ].sigma - sigmamean );
413            Sinitsd += ( particles[n].Sinit - Sinitmean ) * ( particles[n
                    ].Sinit - Sinitmean );
414            Iinitsd += ( particles[n].Iinit - Iinitmean ) * ( particles[n
                    ].Iinit - Iinitmean );
415            Rinitsd += ( particles[n].Rinit - Rinitmean ) * ( particles[n
                    ].Rinit - Rinitmean );
416
417        }
418
419        R0sd         /= NP;
420        rsd          /= NP;
421        sigmasd      /= NP;
422        Sinitsd      /= NP;
423        Iinitsd      /= NP;
424        Rinitsd      /= NP;
425
426        partInfo->R0mean    = R0mean;
427        partInfo->R0sd      = R0sd;
428        partInfo->sigmamean = sigmamean;
429        partInfo->sigmasd   = sigmasd;
430        partInfo->rmean     = rmean;
431        partInfo->rsd       = rsd;
432        partInfo->Sinitmean = Sinitmean;
```

```
433        partInfo->Sinitsd   = Sinitsd;
434        partInfo->Iinitmean = Iinitmean;
435        partInfo->Iinitsd   = Iinitsd;
436        partInfo->Rinitmean = Rinitmean;
437        partInfo->Rinitsd   = Rinitsd;
438
439 }
440
441 double randu() {
442
443        return (double) rand() / (double) RAND_MAX;
444
445 }
446
447
448 /*   Return a normally distributed random number with mean 0 and
         standard deviation 1
449      Uses the polar form of the Box-Muller transformation
450      From http://www.design.caltech.edu/erik/Misc/Gaussian.html
451      */
452 double randn() {
453
454        double x1, x2, w, y1;
455
456        do {
457            x1 = 2.0 * randu() - 1.0;
458            x2 = 2.0 * randu() - 1.0;
459            w = x1 * x1 + x2 * x2;
460        } while ( w >= 1.0 );
461
462        w = sqrt( (-2.0 * log( w ) ) / w );
463        y1 = x1 * w;
464
465        return y1;
466
467 }
```

Appendix C

Parameter Fitting

C.1 SIR Forward Simulator

The basic Stochastic SIR model simulation function.

```
1  ## Dexter Barrows
2  ## dbarrows.github.com
3  ## McMaster University
4  ## 2016
5
6  StocSIR ← function(y, pars, T, steps) {
7
8      out ← matrix(NA, nrow = (T+1), ncol = 4)
9
10     R0 ← pars[['R0']]
11     r  ← pars[['r']]
12     N  ← pars[['N']]
13     eta ← pars[['eta']]
14     berr ← pars[['berr']]
15
16     S ← y[['S']]
17     I ← y[['I']]
18     R ← y[['R']]
19
20     B0 ← R0 * r / N
21     B ← B0
22
23     out[1,] ← c(S,I,R,B)
24
25     h ← 1 / steps
26
27     for ( i in 1:(T*steps) ) {
28
```

```
            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
            I ← I + h*dI
            R ← R + h*dR

            if (i %% steps == 0)
                out[i/steps+1,] ← c(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
```

Appendix D

Forecasting Frameworks

D.1 IF2 Parametric Bootstrapping Function

The parametric bootstrapping machinery used to produce forecasts.

```
1  # Dexter Barrows
2  # dbarrows.github.io
3  # McMaster University
4  # 2016
5  #
6  # IF2 parametric bootstrapping function
7
8  library(foreach)
9  library(parallel)
10 library(doParallel)
11 library(Rcpp)
12
13 if2_paraboot ← function(if2data_parent, T, Tlim, steps, N, nTrials,
14     if2file, if2_s_file, stoc_sir_file, NP, nPasses, coolrate) {
15
16   source(stoc_sir_file)
17
18   if (nTrials < 2)
19     ntrials ← 2
20
21   # unpack if2 first fit data
22   # ...parameters
23   paramdata_parent ← data.frame( if2data_parent$paramdata )
24   names(paramdata_parent) ← c("R0", "r", "sigma", "eta", "berr", "
25       Sinit", "Iinit", "Rinit")
26   parmeans_parent ← colMeans(paramdata_parent)
27   names(parmeans_parent) ← c("R0", "r", "sigma", "eta", "berr", "
28       Sinit", "Iinit", "Rinit")
29   # ...states
```

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

```
73
74      ## generate the rest of the trajectory
75      ##
76
77      # pack new parameter estimates
78      pars ← with(  as.list(parmeans),
79                      c(R0 = R0,
80                        r = r,
81                        N = N,
82                        eta = eta,
83                        berr = berr) )
84      init_cond ← c(S = statemeans_parent[['S']],
85                      I = statemeans_parent[['I']],
86                      R = statemeans_parent[['R']])
87
88      # generate remaining trajectory part
89      sdeout_future ← StocSIR(init_cond, pars, T-Tlim, steps)
90      colnames(sdeout_future) ← c('S','I','R','B')
91
92      return ( c( counts = unname(sdeout_future[,'I']),
93                  parmeans,
94                  time = if2time[['user.self']]) )
95
96
97    }
98
99    return(trajectories)
100
101 }
```

D.2 RStan Forward Simulator

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

```
1 ## Dexter Barrows
2 ## dbarrows.github.io
3 ## McMaster University
4 ## 2016
5
6 StocSIRstan ← function(y, pars, T, steps, berrvec, bveclim) {
7
8   out ← matrix(NA, nrow = (T+1), ncol = 4)
9
10   R0 ← pars[['R0']]
11   r ← pars[['r']]
12   N ← pars[['N']]
13   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
  I ← I + h*dI
  R ← R + 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 outlined SIRS function.

```
## Dexter Barrows
## dbarrows.github.io
## McMaster University
## 2016

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) ) {

```

```
30            #Bfac ← 1/2 - cos((2*pi/365)*i)/2
31            Bfac ← exp(2*cos((2*pi/365)*i) - 2)
32
33      B ← exp( log(B) + eta*(log(B0) - log(B)) + rnorm(1, 0, berr) )
34
35      BSI ← Bfac*B*S*I
36      rI ← r*I
37          reR ← re*R
38
39      dS ← -BSI + reR
40      dI ← BSI - rI
41      dR ← rI - reR
42
43      S ← S + h*dS   #newInf
44      I ← I + h*dI   #newInf - h*dR
45      R ← R + h*dR   #h*dR
46
47      if (i %% steps == 0)
48        out[i/steps+1,] ← c(S,I,R,B)
49
50    }
51
52    colnames(out) ← c("S","I","R","B")
53    return(out)
54
55 }
56
57 ### suggested parameters
58 #
59 # T       ← 200
60 # i_infec   ← 10
61 # steps   ← 7
62 # N       ← 500
63 # sigma   ← 5
64 #
65 # pars ← c(R0 = 3.0,   # new infected people per infected person
66 #          r = 0.1,   # recovery rate
67 #          N = 500,    # population size
68 #          eta = 0.5,  # geometric random walk
69 #          berr = 0.5, # Beta geometric walk noise
70 #          re = 1)   # resuceptibility rate
```

E.2   SIRS HMC R Function Code

R code to simulate the outlined SIRS function with HMC state reconstruction.

```
1 ## Dexter Barrows
2 ## dbarrows.github.io
```

```
 1    3 ## McMaster University
 2    4 ## 2016
 3    5
 4    6 StocSIRSstan ← function(y, pars, T, steps, berrvec, bveclim) {
 5    7
 6    8   out ← matrix(NA, nrow = (T+1), ncol = 4)
 7    9
 8   10   R0 ← pars[['R0']]
 9   11   r ← pars[['r']]
10   12   N ← pars[['N']]
11   13   eta ← pars[['eta']]
12   14   berr ← pars[['berr']]
13   15   re ← pars[['re']]
14   16
15   17   S ← y[['S']]
16   18   I ← y[['I']]
17   19   R ← y[['R']]
18   20
19   21   B0 ← R0 * r / N
20   22   B ← B0
21   23
22   24   out[1,] ← c(S,I,R,B)
23   25
24   26   h ← 1 / steps
25   27
26   28   for ( i in 1:(T*steps) ) {
27   29
28   30     Bfac ← exp(2*cos((2*pi/365)*i) - 2)
29   31
30   32       if (i <= bveclim) {
31   33         B ← exp( log(B) + eta*(log(B0) - log(B)) + berrvec[i])
32   34       } else {
33   35           B ← exp( log(B) + eta*(log(B0) - log(B)) + rnorm(1, 0, berr
34            ))
35   36       }
36   37
37   38     BSI ← Bfac*B*S*I
38   39     rI ← r*I
39   40     reR ← re*R
40   41
41   42     dS ← -BSI + reR
42   43     dI ← BSI - rI
43   44     dR ← rI -reR
44   45
45   46     S ← S + h*dS   #newInf
46   47     I ← I + h*dI   #newInf - h*dR
47   48     R ← R + h*dR   #h*dR
48   49
49   50     if (i %% steps == 0)
50   51       out[i/steps+1,] ← c(S,I,R,B)
51   52
```

```
53    }
54
55    return(out)
56
57 }
58
59 ### suggested parameters
60 #
61 # T       ← 200
62 # i_infec  ← 5
63 # steps   ← 7
64 # N       ← 500
65 # sigma   ← 5
66 #
67 # pars ← c(R0 = 3.0,  # new infected people per infected person
68 #            r = 0.1,   # recovery rate
69 #            gam = 2,   # new infected shock intensity
70 #          N = 500,    # population size
71 #          eta = 0.5,  # geometric random walk
72 #          berr = 0.5, # Beta geometric walk noise
73 #            re = 2)   # resuceptibility rate
```

E.3   SMAP Code

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

```
1 ## Dexter Barrows
2 ## dbarrows.github.io
3 ## McMaster University
4 ## 2016
5
6 library(pracma) # needed for tiling function
7
8 smap ← function(data, E, theta, stepsAhead) {
9
10     # construct library
11     tseries ← as.vector(data)
12     liblen  ← length(tseries) - E + 1 - stepsAhead
13     lib     ← matrix(NA, liblen, E)
14
15     for (i in 1:E) {
16         lib[,i] ← tseries[(E-i+1):(liblen+E-i)]
17     }
18
19     # predict from the last index
20     tslen ← length(tseries)
21     predictee ← rev(t(as.matrix(tseries[(tslen-E+1):tslen])))
22     predictions ← numeric(stepsAhead)
```

```
    # 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

    }

    return(predictions)

}
```

E.4   SMAP Parameter Optimization Code

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

```
## Dexter Barrows
## dbarrows.github.io
## McMaster University
## 2016
```

```
 1   5
 2   6  library(deSolve)
 3   7  library(ggplot2)
 4   8  library(RColorBrewer)
 5   9  library(pracma)     ## for tiling function
 6  10
 7  11  set.seed(1010)
 8  12
 9  13  ## external files
10  14  ##
11  15  stoc_sirs_file  ← paste(getwd(), "../sir-functions", "StocSIRS.r",
12         sep = "/")
13  16  smap_file       ← paste(getwd(), "smap.r", sep = "/")
14  17  source(stoc_sirs_file)
15  18  source(smap_file)
16  19
17  20  ## parameters
18  21  ##
19  22  T      ← 6*52
20  23  Tlim  ← T - 52
21  24  i_infec ← 10
22  25  steps  ← 7
23  26  N      ← 500
24  27  sigma  ← 5
25  28
26  29  true_pars ← c(  R0 = 3.0,   # new infected people per infected person
27  30                   r = 0.1,   # recovery rate
28  31               N = 500,      # population size
29  32               eta = 0.5,    # geometric random walk
30  33               berr = 0.5,  # Beta geometric walk noise
31  34                re = 1)    # resuceptibility rate
32  35
33  36  true_init_cond ← c(S = N - i_infec,
34  37                         I = i_infec,
35  38                         R = 0)
36  39
37  40  ## trial parameter values to check
38  41  ##
39  42  Elist ← 1:20
40  43  thetalist ← 10*exp(-(seq(0,9.5,0.5)))
41  44  nTrials ← 100
42  45
43  46  ssemat ← matrix(NA, 20, 20)
44  47
45  48  for (i in 1:length(Elist)) {
46  49    for (j in 1:length(thetalist)) {
47  50
48  51        ssemean ← 0
49  52
50  53        for (k in 1:nTrials) {
51  54
```

```
55          E ← Elist[i]
56          theta ← thetalist[j]
57
58          ## get true trajectory
59          ##
60          sdeout ← StocSIRS(true_init_cond, true_pars, T, steps)
61
62          ## perturb to get data
63          ##
64          infec_counts_raw ← sdeout[1:(Tlim+1),'I'] + rnorm(Tlim+1,0,
                sigma)
65          infec_counts   ← ifelse(infec_counts_raw < 0, 0, infec_counts_
                raw)
66
67          predictions ← smap(infec_counts, E, theta, 52)
68
69          err ← sdeout[(Tlim+2):dim(sdeout)[1],'I'] - predictions
70          sse ← sum(err^2)
71
72          ssemean ← ssemean + (sse / nTrials)
73
74      }
75
76      ssemat[i,j] ← ssemean
77
78
79  }
80 }
81
82 quartz()
83 image(-ssemat)
84 quartz()
85 filled.contour(-ssemat)
86
87 mininds ← which(ssemat==min(ssemat),arr.ind=TRUE)
88
89 Emin ← Elist[mininds[,'row']]
90 thetamin ← thetalist[mininds[,'col']]
91
92 print(Emin)
93 print(thetamin)
```

E.5   RStan SIRS Code

This code implements a periodic SIRS model in Rstan.

```
1 ## Dexter Barrows
2 ## dbarrows.github.io
```

```
 1   3 ## McMaster University
 2   4 ## 2016
 3   5
 4   6 data {
 5   7
 6   8     int     <lower=1>   T;      // total integration steps
 7   9     real                y[T];   // observed number of cases
 8  10     int     <lower=1>   N;      // population size
 9  11     real                h;      // step size
10  12
11  13 }
12  14
13  15 parameters {
14  16
15  17     real <lower=0, upper=10>      R0;      // R0
16  18     real <lower=0, upper=10>      r;       // recovery rate
17  19     real <lower=0, upper=10>      re;      // resusceptibility rate
18  20     real <lower=0, upper=20>      sigma;   // observation error
19  21     real <lower=0, upper=30>      Iinit;   // initial infected
20  22     real <lower=0, upper=1>       eta;     // geometric walk
21            attraction strength
22  23     real <lower=0, upper=1>       berr;    // beta walk noise
23  24     real <lower=-1.5, upper=1.5>  Bnoise[T];  // Beta vector
24  25
25  26 }
26  27
27  28 model {
28  29
29  30     real S[T];
30  31     real I[T];
31  32     real R[T];
32  33     real B[T];
33  34     real B0;
34  35
35  36     real pi;
36  37     real Bfac;
37  38
38  39     pi ← 3.1415926535;
39  40
40  41     B0 ← R0 * r / N;
41  42
42  43     B[1] ← B0;
43  44
44  45     S[1] ← N - Iinit;
45  46     I[1] ← Iinit;
46  47     R[1] ← 0.0;
47  48
48  49     for (t in 2:T) {
49  50
50  51         Bnoise[t] ~ normal(0,berr);
51  52         Bfac ← exp(2*cos((2*pi/365)*t) - 2);
```

```
53          B[t] ← exp( log(B0) + eta * ( log(B[t-1]) - log(B0) ) +
54              Bnoise[t] );

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

58
59          if (y[t] > 0) {
60              y[t] ~ normal( I[t], sigma );
61          }

62
63      }

64
65      R0       ~ lognormal(1,1);
66      r        ~ lognormal(1,1);
67      sigma    ~ lognormal(1,1);
68      re       ~ lognormal(1,1);
69      Iinit    ~ normal(y[1], sigma);

70
71  }
```

E.6   IF2 SIRS Code

This code implements a periodic SIRS model using IF2 in C++.

```
1  /*   Dexter Barrows
2       dbarrows.github.io
3       McMaster University
4       2016
5
6       */
7
8  #include <stdio.h>
9  #include <math.h>
10 #include <sys/time.h>
11 #include <time.h>
12 #include <stdlib.h>
13 #include <string>
14 #include <cmath>
15 #include <cstdlib>
16 #include <fstream>
17
18 #define Treal       100         // time to simulate over
19 #define R0true      3.0         // infectiousness
20 #define rtrue       0.1         // recovery rate
21 #define retrue      0.05        // resusceptibility rate
22 #define Nreal       500.0       // population size
23 #define etatrue     0.5         // real drift attraction strength
```

```
24 #define berrtrue     0.5              // real beta drift noise
25 #define merr          5.0              // expected measurement error
26 #define I0            5.0              // Initial infected individuals
27
28 #define PSC           0.5              // scale factor for more sensitive
       parameters
29
30 #include <Rcpp.h>
31 using namespace Rcpp;
32
33 struct State {
34     double S;
35     double I;
36     double R;
37 };
38
39 struct Particle {
40     double R0;
41     double r;
42     double re;
43     double sigma;
44     double eta;
45     double berr;
46     double B;
47     double S;
48     double I;
49     double R;
50     double Sinit;
51     double Iinit;
52     double Rinit;
53 };
54
55 struct ParticleInfo {
56     double R0mean;      double R0sd;
57     double rmean;       double rsd;
58     double remean;      double resd;
59     double sigmamean;   double sigmasd;
60     double etamean;     double etasd;
61     double berrmean;    double berrsd;
62     double Sinitmean;   double Sinitsd;
63     double Iinitmean;   double Iinitsd;
64     double Rinitmean;   double Rinitsd;
65 };
66
67
68 int timeval_subtract (double *result, struct timeval *x, struct
       timeval *y);
69 int check_double(double x,double y);
70 void exp_euler_SIRS(double h, double t0, double tn, int N, Particle *
       particle);
71 void copyParticle(Particle * dst, Particle * src);
```

```
 1   72 void perturbParticles(Particle * particles, int N, int NP, int
 2         passnum, double coolrate);
 3   73 void particleDiagnostics(ParticleInfo * partInfo, Particle *
 4         particles, int NP);
 5   74 void getStateMeans(State * state, Particle* particles, int NP);
 6   75 NumericMatrix if2(NumericVector * data, int T, int N);
 7   76 double randu();
 8   77 double randn();
 9   78
10   79 // [[Rcpp::export]]
11   80 Rcpp::List if2_sirs(NumericVector data, int T, int N, int NP, int
12         nPasses, double coolrate) {
13   81
14   82     int npar = 9;
15   83
16   84     NumericMatrix paramdata(NP, npar);
17   85     NumericMatrix means(nPasses, npar);
18   86     NumericMatrix sds(nPasses, npar);
19   87     NumericMatrix statemeans(T, 3);
20   88     NumericMatrix statedata(NP, 4);
21   89
22   90     srand(time(NULL));         // Seed PRNG with system time
23   91
24   92     double w[NP];              // particle weights
25   93
26   94     Particle particles[NP];     // particle estimates for current
27            step
28   95     Particle particles_old[NP]; // intermediate particle states for
29            resampling
30   96
31   97     printf("Initializing particle states\n");
32   98
33   99     // initialize particle parameter states (seeding)
34  100     for (int n = 0; n < NP; n++) {
35  101
36  102         double R0can, rcan, recan, sigmacan, Iinitcan, etacan,
37             berrcan;
38  103
39  104         do {
40  105             R0can = R0true + R0true*randn();
41  106         } while (R0can < 0);
42  107         particles[n].R0 = R0can;
43  108
44  109         do {
45  110             rcan = rtrue + rtrue*randn();
46  111         } while (rcan < 0);
47  112         particles[n].r = rcan;
48  113
49  114         do {
50  115             recan = retrue + retrue*randn();
51  116         } while (recan < 0);
```

```
117            particles[n].re = recan;
118
119            particles[n].B = (double) R0can * rcan / N;
120
121            do {
122                sigmacan = merr + merr*randn();
123            } while (sigmacan < 0);
124            particles[n].sigma = sigmacan;
125
126            do {
127                etacan = etatrue + PSC*etatrue*randn();
128            } while (etacan < 0 || etacan > 1);
129            particles[n].eta = etacan;
130
131            do {
132                berrcan = berrtrue + PSC*berrtrue*randn();
133            } while (berrcan < 0);
134            particles[n].berr = berrcan;
135
136            do {
137                Iinitcan = I0 + I0*randn();
138            } while (Iinitcan < 0 || N < Iinitcan);
139            particles[n].Sinit = N - Iinitcan;
140            particles[n].Iinit = Iinitcan;
141            particles[n].Rinit = 0.0;
142
143        }
144
145        // START PASSES THROUGH DATA
146
147        printf("Starting filter\n");
148        printf("----------------\n");
149        printf("Pass\n");
150
151
152        for (int pass = 0; pass < nPasses; pass++) {
153
154            printf("...%d / %d\n", pass, nPasses);
155
156            // reset particle system evolution states
157            for (int n = 0; n < NP; n++) {
158
159                particles[n].S = particles[n].Sinit;
160                particles[n].I = particles[n].Iinit;
161                particles[n].R = particles[n].Rinit;
162                particles[n].B = (double) particles[n].R0 * particles[n].
                       r / N;
163
164            }
165
166            if (pass == (nPasses-1)) {
```

```
1  167                    State sMeans;
2  168                    getStateMeans(&sMeans, particles, NP);
3  169                    statemeans(0,0) = sMeans.S;
4  170                    statemeans(0,1) = sMeans.I;
5  171                    statemeans(0,2) = sMeans.R;
6  172               }
7  173
8  174           for (int t = 1; t < T; t++) {
9  175
10 176               // generate individual predictions and weight
11 177               for (int n = 0; n < NP; n++) {
12 178
13 179                    exp_euler_SIRS(1.0/7.0, (double) t-1, (double) t, N,
14                            &particles[n]);
15 180
16 181                    double merr_par = particles[n].sigma;
17 182                    double y_diff   = data[t] - particles[n].I;
18 183
19 184                    w[n] = 1.0/(merr_par*sqrt(2.0*M_PI)) * exp( - y_diff*
20                         y_diff / (2.0*merr_par*merr_par) );
21 185
22 186               }
23 187
24 188               // cumulative sum
25 189               for (int n = 1; n < NP; n++) {
26 190                    w[n] += w[n-1];
27 191               }
28 192
29 193               // save particle states to resample from
30 194               for (int n = 0; n < NP; n++){
31 195                    copyParticle(&particles_old[n], &particles[n]);
32 196               }
33 197
34 198               // resampling
35 199               for (int n = 0; n < NP; n++) {
36 200
37 201                    double w_r = randu() * w[NP-1];
38 202                    int i = 0;
39 203                    while (w_r > w[i]) {
40 204                        i++;
41 205                    }
42 206
43 207                    // i is now the index to copy state from
44 208                    copyParticle(&particles[n], &particles_old[i]);
45 209
46 210               }
47 211
48 212               // between-iteration perturbations, not after last time
49                        step
50 213               if (t < (T-1))
51 214                    perturbParticles(particles, N, NP, pass, coolrate);
```

```
215
216                    if (pass == (nPasses-1)) {
217                        State sMeans;
218                        getStateMeans(&sMeans, particles, NP);
219                        statemeans(t,0) = sMeans.S;
220                        statemeans(t,1) = sMeans.I;
221                        statemeans(t,2) = sMeans.R;
222                    }
223
224            }
225
226            ParticleInfo pInfo;
227            particleDiagnostics(&pInfo, particles, NP);
228
229            means(pass, 0) = pInfo.R0mean;
230            means(pass, 1) = pInfo.rmean;
231            means(pass, 2) = pInfo.remean;
232            means(pass, 3) = pInfo.sigmamean;
233            means(pass, 4) = pInfo.etamean;
234            means(pass, 5) = pInfo.berrmean;
235            means(pass, 6) = pInfo.Sinitmean;
236            means(pass, 7) = pInfo.Iinitmean;
237            means(pass, 8) = pInfo.Rinitmean;
238
239            sds(pass, 0) = pInfo.R0sd;
240            sds(pass, 1) = pInfo.rsd;
241            sds(pass, 2) = pInfo.resd;
242            sds(pass, 3) = pInfo.sigmasd;
243            sds(pass, 4) = pInfo.etasd;
244            sds(pass, 5) = pInfo.berrsd;
245            sds(pass, 6) = pInfo.Sinitsd;
246            sds(pass, 7) = pInfo.Iinitsd;
247            sds(pass, 8) = pInfo.Rinitsd;
248
249            // between-pass perturbations, not after last pass
250            if (pass < (nPasses + 1))
251                perturbParticles(particles, N, NP, pass, coolrate);
252
253        }
254
255    ParticleInfo pInfo;
256    particleDiagnostics(&pInfo, particles, NP);
257
258    printf("Parameter results (mean | sd)\n");
259    printf("---------------------------\n");
260    printf("R0        %f %f\n", pInfo.R0mean, pInfo.R0sd);
261    printf("r         %f %f\n", pInfo.rmean, pInfo.rsd);
262    printf("re        %f %f\n", pInfo.remean, pInfo.resd);
263    printf("sigma     %f %f\n", pInfo.sigmamean, pInfo.sigmasd);
264    printf("eta       %f %f\n", pInfo.etamean, pInfo.etasd);
265    printf("berr      %f %f\n", pInfo.berrmean, pInfo.berrsd);
```

```
266        printf("S_init   %f %f\n", pInfo.Sinitmean, pInfo.Sinitsd);
267        printf("I_init    %f %f\n", pInfo.Iinitmean, pInfo.Iinitsd);
268        printf("R_init    %f %f\n", pInfo.Rinitmean, pInfo.Rinitsd);
269
270        printf("\n");
271
272        // Get particle results to pass back to R
273
274        for (int n = 0; n < NP; n++) {
275
276            paramdata(n, 0) = particles[n].R0;
277            paramdata(n, 1) = particles[n].r;
278            paramdata(n, 2) = particles[n].re;
279            paramdata(n, 3) = particles[n].sigma;
280            paramdata(n, 4) = particles[n].eta;
281            paramdata(n, 5) = particles[n].berr;
282            paramdata(n, 6) = particles[n].Sinit;
283            paramdata(n, 7) = particles[n].Iinit;
284            paramdata(n, 8) = particles[n].Rinit;
285
286        }
287
288        for (int n = 0; n < NP; n++) {
289
290            statedata(n, 0) = particles[n].S;
291            statedata(n, 1) = particles[n].I;
292            statedata(n, 2) = particles[n].R;
293            statedata(n, 3) = particles[n].B;
294
295        }
296
297        return Rcpp::List::create(  Rcpp::Named("paramdata") = paramdata,
298                                    Rcpp::Named("means") = means,
299                                    Rcpp::Named("statemeans") =
                                            statemeans,
300                                    Rcpp::Named("statedata") = statedata,
301                                    Rcpp::Named("sds") = sds);
302
303 }
304
305
306 /*  Use the Explicit Euler integration scheme to integrate SIR model
        forward in time
307     double h    - time step size
308     double t0   - start time
309     double tn   - stop time
310     double * y  - current system state; a three-component vector
            representing [S I R], susceptible-infected-recovered
311
312     */
313 void exp_euler_SIRS(double h, double t0, double tn, int N, Particle *
```

```
 1        particle) {
 2 314
 3 315    int num_steps = floor( (tn-t0) / h );
 4 316
 5 317    double S = particle->S;
 6 318    double I = particle->I;
 7 319    double R = particle->R;
 8 320
 9 321    double R0   = particle->R0;
10 322    double r    = particle->r;
11 323    double re   = particle->re;
12 324    double B0   = R0 * r / N;
13 325    double eta  = particle->eta;
14 326    double berr = particle->berr;
15 327
16 328    double B = particle->B;
17 329
18 330    for(int i = 0; i < num_steps; i++) {
19 331
20 332        //double Bfac = 0.5 - 0.95*cos( (2.0*M_PI/365)*(t0*num_steps+
21              i) )/2.0;
22 333        double Bfac = exp(2*cos((2*M_PI/365)*(t0*num_steps+i)) - 2);
23 334        B = exp( log(B) + eta*(log(B0) - log(B)) + berr*randn() );
24 335
25 336        double BSI = Bfac*B*S*I;
26 337        double rI  = r*I;
27 338        double reR = re*R;
28 339
29 340        // get derivatives
30 341        double dS = - BSI + reR;
31 342        double dI = BSI - rI;
32 343        double dR = rI - reR;
33 344
34 345        // step forward by h
35 346        S += h*dS;
36 347        I += h*dI;
37 348        R += h*dR;
38 349
39 350    }
40 351
41 352    particle->S = S;
42 353    particle->I = I;
43 354    particle->R = R;
44 355    particle->B = B;
45 356
46 357 }
47 358
48 359
49 360 /*  Particle pertubation function to be run between iterations and
50         passes
51 361
```

```
 1  362        */
 2  363  void perturbParticles(Particle * particles, int N, int NP, int
 3            passnum, double coolrate) {
 4  364
 5  365        //double coolcoef = exp( - (double) passnum / coolrate );
 6  366        double coolcoef = pow(coolrate, passnum);
 7  367
 8  368
 9  369        double spreadR0     = coolcoef * R0true / 10.0;
10  370        double spreadr      = coolcoef * rtrue / 10.0;
11  371        double spreadre     = coolcoef * retrue / 10.0;
12  372        double spreadsigma  = coolcoef * merr / 10.0;
13  373        double spreadIinit  = coolcoef * I0 / 10.0;
14  374        double spreadeta    = coolcoef * etatrue / 10.0;
15  375        double spreadberr   = coolcoef * berrtrue / 10.0;
16  376
17  377
18  378        double R0can, rcan, recan, sigmacan, Iinitcan, etacan, berrcan;
19  379
20  380        for (int n = 0; n < NP; n++) {
21  381
22  382            do {
23  383                R0can = particles[n].R0 + spreadR0*randn();
24  384            } while (R0can < 0);
25  385            particles[n].R0 = R0can;
26  386
27  387            do {
28  388                rcan = particles[n].r + spreadr*randn();
29  389            } while (rcan < 0);
30  390            particles[n].r = rcan;
31  391
32  392            do {
33  393                recan = particles[n].re + spreadre*randn();
34  394            } while (recan < 0);
35  395            particles[n].re = recan;
36  396
37  397            do {
38  398                sigmacan = particles[n].sigma + spreadsigma*randn();
39  399            } while (sigmacan < 0);
40  400            particles[n].sigma = sigmacan;
41  401
42  402            do {
43  403                etacan = particles[n].eta + PSC*spreadeta*randn();
44  404            } while (etacan < 0 || etacan > 1);
45  405            particles[n].eta = etacan;
46  406
47  407            do {
48  408                berrcan = particles[n].berr + PSC*spreadberr*randn();
49  409            } while (berrcan < 0);
50  410            particles[n].berr = berrcan;
51  411
```

```
412            do {
413                Iinitcan = particles[n].Iinit + spreadIinit*randn();
414            } while (Iinitcan < 0 || Iinitcan > 500);
415            particles[n].Iinit = Iinitcan;
416            particles[n].Sinit = N - Iinitcan;
417
418        }
419
420 }
421
422
423 /*   Convinience function for particle resampling process
424
425      */
426 void copyParticle(Particle * dst, Particle * src) {
427
428     dst->R0     = src->R0;
429     dst->r      = src->r;
430     dst->re     = src->re;
431     dst->sigma  = src->sigma;
432     dst->eta    = src->eta;
433     dst->berr   = src->berr;
434     dst->B      = src->B;
435     dst->S      = src->S;
436     dst->I      = src->I;
437     dst->R      = src->R;
438     dst->Sinit  = src->Sinit;
439     dst->Iinit  = src->Iinit;
440     dst->Rinit  = src->Rinit;
441
442 }
443
444 void particleDiagnostics(ParticleInfo * partInfo, Particle *
445        particles, int NP) {
446     double  R0mean      = 0.0,
447             rmean       = 0.0,
448             remean      = 0.0,
449             sigmamean   = 0.0,
450             etamean     = 0.0,
451             berrmean    = 0.0,
452             Sinitmean   = 0.0,
453             Iinitmean   = 0.0,
454             Rinitmean   = 0.0;
455
456     // means
457
458     for (int n = 0; n < NP; n++) {
459
460         R0mean      += particles[n].R0;
461         rmean       += particles[n].r;
```

```
462            remean      += particles[n].re;
463            etamean     += particles[n].eta,
464            berrmean    += particles[n].berr,
465            sigmamean   += particles[n].sigma;
466            Sinitmean   += particles[n].Sinit;
467            Iinitmean   += particles[n].Iinit;
468            Rinitmean   += particles[n].Rinit;
469
470        }
471
472        R0mean      /= NP;
473        rmean       /= NP;
474        remean      /= NP;
475        sigmamean   /= NP;
476        etamean     /= NP;
477        berrmean    /= NP;
478        Sinitmean   /= NP;
479        Iinitmean   /= NP;
480        Rinitmean   /= NP;
481
482        // standard deviations
483
484        double  R0sd    = 0.0,
485                rsd     = 0.0,
486                resd    = 0.0,
487                sigmasd = 0.0,
488                etasd   = 0.0,
489                berrsd  = 0.0,
490                Sinitsd = 0.0,
491                Iinitsd = 0.0,
492                Rinitsd = 0.0;
493
494        for (int n = 0; n < NP; n++) {
495
496            R0sd    += ( particles[n].R0 - R0mean ) * ( particles[n].R0 -
                    R0mean );
497            rsd     += ( particles[n].r - rmean ) * ( particles[n].r -
                    rmean );
498            resd    += ( particles[n].re - rmean ) * ( particles[n].re -
                    rmean );
499            sigmasd += ( particles[n].sigma - sigmamean ) * ( particles[n
                    ].sigma - sigmamean );
500            etasd   += ( particles[n].eta - etamean ) * ( particles[n.
                    eta - etamean );
501            berrsd  += ( particles[n].berr - berrmean ) * ( particles[n.
                    berr - berrmean );
502            Sinitsd += ( particles[n].Sinit - Sinitmean ) * ( particles[n
                    ].Sinit - Sinitmean );
503            Iinitsd += ( particles[n].Iinit - Iinitmean ) * ( particles[n
                    ].Iinit - Iinitmean );
504            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;
    }

```

```
555        state->S = (double) Smean / NP;
556        state->I = (double) Imean / NP;
557        state->R = (double) Rmean / NP;
558
559    }
560
561
562    /*   Return a normally distributed random number with mean 0 and
           standard deviation 1
563        Uses the polar form of the Box-Muller transformation
564        From http://www.design.caltech.edu/erik/Misc/Gaussian.html
565        */
566    double randn() {
567
568        double x1, x2, w, y1;
569
570        do {
571            x1 = 2.0 * randu() - 1.0;
572            x2 = 2.0 * randu() - 1.0;
573            w = x1 * x1 + x2 * x2;
574        } while ( w >= 1.0 );
575
576        w = sqrt( (-2.0 * log( w ) ) / w );
577        y1 = x1 * w;
578
579        return y1;
580
581    }
```

Appendix F

Spatial Epidemics

F.1 Spatial SIR R Function Code

R code to simulate the outlined Spatial SIR function.

```r
## Dexter Barrows
## dbarrows.github.io
## McMaster University
## 2016

## 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
```

114

```
27       BSI ← numeric(nloc)
28       rI ← numeric(nloc)
29
30       ## extract parameters
31       R0 ← pars[['R0']]
32       r ← pars[['r']]
33       N ← pars[['N']]
34       eta ← pars[['eta']]
35       berr ← pars[['berr']]
36       phi ← pars[['phi']]
37
38       B0 ← rep(R0*r/N, nloc)
39
40       ## state vectors
41       S ← ymat[,'S']
42       I ← ymat[,'I']
43       R ← ymat[,'R']
44       B ← B0
45
46       ## assign starting to output matrix
47       out[,,1] ← cbind(ymat, B0)
48
49       h ← 1 / steps
50
51       for ( i in 1:(T*steps) ) {
52
53           B ← exp( log(B) + eta*(log(B0) - log(B)) + rnorm(nloc, 0,
                 berr) )
54
55           for (loc in 1:nloc) {
56             n ← neinum[loc]
57             sphi ← 1 - phi*(n/(n+1))
58             ophi ← phi/(n+1)
59             nBIsum ← B[neibmat[loc,1:n]] %*% I[neibmat[loc,1:n]]
60             BSI[loc] ← S[loc]*( sphi*B[loc]*I[loc] + ophi*nBIsum )
61           }
62
63           rI ← r*I
64
65           dS ← -BSI
66           dI ← BSI - rI
67           dR ← rI
68
69           S ← S + h*dS
70           I ← I + h*dI
71           R ← R + h*dR
72
73           if (i %% steps == 0) {
74               out[,,i/steps+1] ← cbind(S,I,R,B)
75           }
76
```

```
77       }
78
79    return(out)
80
81 }
82
83 ### Suggested parameters
84 #
85 # T        ← 60
86 # i_infec  ← 5
87 # steps    ← 7
88 # N        ← 500
89 # sigma    ← 10
90 #
91 # pars ← c(R0 = 3.0,      # new infected people per infected person
92 #             r = 0.1,       # recovery rate
93 #             N = 500,       # population size
94 #             eta = 0.5,    # geometric random walk
95 #             berr = 0.5,   # Beta geometric walk noise
96 #       phi = 0.5)    # degree of connectivity
```

F.2 Spatial SIR HMC R Function Code

R code to simulate the outlined Spatial SIR function with HMC state reconstruction.

```
1 ## Dexter Barrows
2 ## dbarrows.github.io
3 ## McMaster University
4 ## 2016
5
6 ## ymat:  Contains the initial conditions where:
7 #        - rows are locations
8 #        - columns are S, I, R
9 ## pars:  Contains the parameters: global values for R0, r, N, eta,
      berr
10 ## T:      The stop time. Since 0 in included, there should be T+1
      time steps in the simulation
11 ## neinum:  Number of neighbors for each location, in order
12 ## neibmat: Contains lists of neighbors for each location
13 #      - rows are parent locations (nodes)
14 #        - columns are locations each parent is attached to (edges)
15 StocSSIRstan ← function(ymat, pars, T, steps, neinum, neibmat,
      berrmat, bmatlim) {
16
17    ## number of locations
18       nloc ← dim(ymat)[1]
19
```

```
20        ## storage
21        ## dims are locations, (S,I,R,B), times
22        # output array
23        out ← array(NA, c(nloc, 4, T+1), dimnames = list(NULL, c("S","I",
              "R","B"), NULL))
24        # temp storage
25        BSI ← numeric(nloc)
26        rI ← numeric(nloc)
27
28        ## extract parameters
29        R0 ← pars[['R0']]
30        r ← pars[['r']]
31        N ← pars[['N']]
32        eta ← pars[['eta']]
33        berr ← pars[['berr']]
34        phi ← pars[['phi']]
35
36        B0 ← rep(R0*r/N, nloc)
37
38        ## state vectors
39        S ← ymat[,'S']
40        I ← ymat[,'I']
41        R ← ymat[,'R']
42        B ← B0
43
44        ## assign starting to output matrix
45        out[,,1] ← cbind(ymat, B0)
46
47        h ← 1 / steps
48
49        for ( i in 1:(T*steps) ) {
50
51          if (i <= bmatlim) {
52            B ← exp( log(B) + eta*(log(B0) - log(B)) + berrmat[,i])
53          } else {
54              B ← exp( log(B) + eta*(log(B0) - log(B)) + rnorm(nloc, 0,
                  berr) )
55          }
56
57
58          for (loc in 1:nloc) {
59            n ← neinum[loc]
60            sphi ← 1 - phi*(n/(n+1))
61            ophi ← phi/(n+1)
62            nBIsum ← B[neibmat[loc,1:n]] %*% I[neibmat[loc,1:n]]
63            BSI[loc] ← S[loc]*( sphi*B[loc]*I[loc] + ophi*nBIsum )
64          }
65
66          rI ← r*I
67
68              dS ← -BSI
```

```
69            dI ← BSI - rI
70            dR ← rI
71
72            S ← S + h*dS
73            I ← I + h*dI
74            R ← R + h*dR
75
76            if (i %% steps == 0)
77                out[,,i/steps+1] ← cbind(S,I,R,B)
78
79        }
80
81     return(out)
82
83 }
84
85 ### Suggested parameters
86 #
87 # T       ← 60
88 # i_infec ← 5
89 # steps   ← 7
90 # N       ← 500
91 # sigma   ← 10
92 #
93 # pars ← c(R0 = 3.0,     # new infected people per infected person
94 #          r = 0.1,      # recovery rate
95 #          N = 500,      # population size
96 #          eta = 0.5,    # geometric random walk
97 #          berr = 0.5,   # Beta geometric walk noise
98 #          phi = 0.5 )   # interconnectivity degree
```

F.3   RStan Spatial SIR Code

This code implements a Spatial SIR model in Rstan.

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

```
13        int       <lower=0>   neinum[nloc];          // number of neighbors
              each location has
14        int                   neibmat[nloc, nloc]; // neighbor list for
              each location
15
16 }
17
18 parameters {
19
20        real <lower=0, upper=10>        R0;      // R0
21        real <lower=0, upper=10>        r;       // recovery rate
22        real <lower=0, upper=20>        sigma;  // observation error
23        real <lower=0, upper=30>        Iinit[nloc];    // initial
              infected for each location
24        real <lower=0, upper=1>         eta;    // geometric walk
              attraction strength
25        real <lower=0, upper=1>         berr;   // beta walk noise
26        real <lower=-1.5, upper=1.5>    Bnoise[nloc,T];   // Beta vector
27        real <lower=0, upper=1>         phi;    // interconnectivity
              strength
28
29 }
30
31 model {
32
33        real S[nloc, T];
34        real I[nloc, T];
35        real R[nloc, T];
36        real B[nloc, T];
37        real B0;
38
39        real BSI[nloc, T];
40        real rI[nloc, T];
41        int  n;
42        real sphi;
43        real ophi;
44        real nBIsum;
45
46        B0 ← R0 * r / N;
47
48        for (loc in 1:nloc) {
49            S[loc, 1] ← N - Iinit[loc];
50            I[loc, 1] ← Iinit[loc];
51            R[loc, 1] ← 0.0;
52            B[loc, 1] ← B0;
53        }
54
55        for (t in 2:T) {
56            for (loc in 1:nloc) {
57
58                Bnoise[loc, t] ˜ normal(0,berr);
```

```
59              B[loc, t] ← exp( log(B[loc, t-1]) + eta * ( log(B0) - log
                    (B[loc, t-1]) ) + Bnoise[loc, t] );
60
61              n ← neinum[loc];
62              sphi ← 1.0 - phi*( n/(n+1.0) );
63              ophi ← phi/(n+1.0);
64
65              nBIsum ← 0.0;
66              for (j in 1:n)
67                  nBIsum ← nBIsum + B[neibmat[loc, j], t-1] * I[neibmat
                        [loc, j], t-1];
68
69              BSI[loc, t] ← S[loc, t-1]*( sphi*B[loc, t-1]*I[loc, t-1]
                    + ophi*nBIsum );
70              rI[loc, t]  ← r*I[loc, t-1];
71
72              S[loc, t] ← S[loc, t-1] + h*( - BSI[loc, t] );
73              I[loc, t] ← I[loc, t-1] + h*( BSI[loc, t] - rI[loc, t] );
74              R[loc, t] ← R[loc, t-1] + h*( rI[loc, t] );
75
76              if (y[loc, t] > 0) {
77                  y[loc, t] ˜ normal( I[loc, t], sigma );
78              }
79
80          }
81      }
82
83      R0      ˜ lognormal(1,1);
84      r       ˜ lognormal(1,1);
85      sigma   ˜ lognormal(1,1);
86      for (loc in 1:nloc) {
87          Iinit[loc] ˜ normal(y[loc, 1], sigma);
88      }
89
90 }
```

F.4 IF2 Spatial SIR Code

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

```
1 /*  Dexter Barrows
2      dbarrows.github.io
3      McMaster University
4      2016
5
6      */
7
8 #include <stdio.h>
```

```
 1    9  #include <math.h>
 2   10  #include <sys/time.h>
 3   11  #include <time.h>
 4   12  #include <stdlib.h>
 5   13  #include <string>
 6   14  #include <cmath>
 7   15  #include <cstdlib>
 8   16  #include <fstream>
 9   17
10   18  #define Treal        100          // time to simulate over
11   19  #define R0true       3.0          // infectiousness
12   20  #define rtrue        0.1          // recovery rate
13   21  #define Nreal        500.0        // population size
14   22  #define etatrue      0.5          // real drift attraction strength
15   23  #define berrtrue     0.5          // real beta drift noise
16   24  #define phitrue      0.5          // real connectivity strength
17   25  #define merr         10.0         // expected measurement error
18   26  #define I0           5.0          // Initial infected individuals
19   27
20   28  #define PSC          0.5          // perturbation scale factor for more
21            sensitive parameters
22   29
23   30  #include <Rcpp.h>
24   31  using namespace Rcpp;
25   32
26   33  struct Particle {
27   34      double R0;
28   35      double r;
29   36      double sigma;
30   37      double eta;
31   38      double berr;
32   39      double phi;
33   40      double * S;
34   41      double * I;
35   42      double * R;
36   43      double * B;
37   44      double * Iinit;
38   45  };
39   46
40   47
41   48  int timeval_subtract (double *result, struct timeval *x, struct
42          timeval *y);
43   49  int check_double(double x,double y);
44   50  void initializeParticles(Particle ** particles, int NP, int nloc, int
45          N);
46   51  void exp_euler_SSIR(double h, double t0, double tn, int N, Particle *
47          particle,
48   52                         NumericVector neinum, NumericMatrix neibmat, int
49                                 nloc) ;
50   53  void copyParticle(Particle * dst, Particle * src, int nloc);
51   54  void perturbParticles(Particle * particles, int N, int NP, int nloc,
```

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

```
 97
 98          if (pass == (nPasses -1)) {
 99              double means[nloc];
100              for (int loc = 0; loc < nloc; loc++) {
101                  means[loc] = 0.0;
102                  for (int n = 0; n < NP; n++) {
103                      means[loc] += particles[n].I[loc] / NP;
104                  }
105                  infecmeans(loc, 0) = means[loc];
106              }
107          }
108
109          for (int t = 1; t < T; t++) {
110
111              // generate individual predictions and weight
112              for (int n = 0; n < NP; n++) {
113
114                  exp_euler_SSIR(1.0/7.0, 0.0, 1.0, N, &particles[n],
115                      neinum, neibmat, nloc);
116                  double merr_par = particles[n].sigma;
117
118                  w[n] = 1.0;
119                  for (int loc = 0; loc < nloc; loc++) {
120                      double y_diff   = data(loc, t) - particles[n].I[
                             loc];
121                      w[n] *= 1.0/(merr_par*sqrt(2.0*M_PI)) * exp( -
                             y_diff*y_diff / (2.0*merr_par*merr_par) );
122                  }
123
124              }
125
126              // cumulative sum
127              for (int n = 1; n < NP; n++) {
128                  w[n] += w[n-1];
129              }
130
131              // save particle states to resample from
132              for (int n = 0; n < NP; n++){
133                  copyParticle(&particles_old[n], &particles[n], nloc);
134              }
135
136              // resampling
137              for (int n = 0; n < NP; n++) {
138
139                  double w_r = randu() * w[NP -1];
140                  int i = 0;
141                  while (w_r > w[i]) {
142                      i++;
143                  }
144
```

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

147
148                    }

149
150                    // between-iteration perturbations, not after last time
                        step
151                    if (t < (T-1))
152                        perturbParticles(particles, N, NP, nloc, pass,
153                            coolrate);

154                    if (pass == (nPasses-1)) {
155                        double means[nloc];
156                        for (int loc = 0; loc < nloc; loc++) {
157                            means[loc] = 0.0;
158                            for (int n = 0; n < NP; n++) {
159                                means[loc] += particles[n].I[loc] / NP;
160                            }
161                            infecmeans(loc, t) = means[loc];
162                        }
163                    }

164
165                }

166
167            // between-pass perturbations, not after last pass
168            if (pass < (nPasses + 1))
169                perturbParticles(particles, N, NP, nloc, pass, coolrate);

170
171        }

172
173        // pack parameter data (minus initial conditions)
174        for (int n = 0; n < NP; n++) {
175            paramdata(n, 0) = particles[n].R0;
176            paramdata(n, 1) = particles[n].r;
177            paramdata(n, 2) = particles[n].sigma;
178            paramdata(n, 3) = particles[n].eta;
179            paramdata(n, 4) = particles[n].berr;
180            paramdata(n, 5) = particles[n].phi;
181        }

182
183        // Pack initial condition data
184        for (int n = 0; n < NP; n++) {
185            for (int loc = 0; loc < nloc; loc++) {
186                initInfec(loc, n) = particles[n].Iinit[loc];
187            }
188        }

189
190        // Pack final state means data
191        double Smeans[nloc], Imeans[nloc], Rmeans[nloc], Bmeans[nloc];
192        for (int loc = 0; loc < nloc; loc++) {
193            Smeans[loc] = 0.0;
```

```
194            Imeans[loc] = 0.0;
195            Rmeans[loc] = 0.0;
196            Bmeans[loc] = 0.0;
197            for (int n = 0; n < NP; n++) {
198                    Smeans[loc] += particles[n].S[loc] / NP;
199                    Imeans[loc] += particles[n].I[loc] / NP;
200                    Rmeans[loc] += particles[n].R[loc] / NP;
201                    Bmeans[loc] += particles[n].B[loc] / NP;
202            }
203            finalstate(loc, 0) = Smeans[loc];
204            finalstate(loc, 1) = Imeans[loc];
205            finalstate(loc, 2) = Rmeans[loc];
206            finalstate(loc, 3) = Bmeans[loc];
207        }
208
209
210        return Rcpp::List::create(  Rcpp::Named("paramdata") = paramdata,
211                                   Rcpp::Named("initInfec") = initInfec,
212                                   Rcpp::Named("infecmeans") =
                                            infecmeans,
213                                   Rcpp::Named("finalstate") =
                                            finalstate);
214
215
216
217 }
218
219
220 /*  Use the Explicit Euler integration scheme to integrate SIR model
        forward in time
221     double h    - time step size
222     double t0   - start time
223     double tn   - stop time
224     double * y  - current system state; a three-component vector
            representing [S I R], susceptible-infected-recovered
225
226     */
227 void exp_euler_SSIR(double h, double t0, double tn, int N, Particle *
        particle,
228                     NumericVector neinum, NumericMatrix neibmat, int
                            nloc) {
229
230     int num_steps = floor( (tn-t0) / h );
231
232     double * S = particle->S;
233     double * I = particle->I;
234     double * R = particle->R;
235     double * B = particle->B;
236
237     // create last state vectors
238     double S_last[nloc];
```

```
  1 239        double I_last[nloc];
  2 240        double R_last[nloc];
  3 241        double B_last[nloc];
  4 242
  5 243        double R0   = particle->R0;
  6 244        double r    = particle->r;
  7 245        double B0   = R0 * r / N;
  8 246        double eta  = particle->eta;
  9 247        double berr = particle->berr;
 10 248        double phi  = particle->phi;
 11 249
 12 250        for(int t = 0; t < num_steps; t++) {
 13 251
 14 252            for (int loc = 0; loc < nloc; loc++) {
 15 253                S_last[loc] = S[loc];
 16 254                I_last[loc] = I[loc];
 17 255                R_last[loc] = R[loc];
 18 256                B_last[loc] = B[loc];
 19 257            }
 20 258
 21 259            for (int loc = 0; loc < nloc; loc++) {
 22 260
 23 261                B[loc] = exp( log(B_last[loc]) + eta*(log(B0) - log(
 24                         B_last[loc])) + berr*randn() );
 25 262
 26 263                int n = neinum[loc];
 27 264                double sphi = 1.0 - phi*( (double) n/(n+1.0) );
 28 265                double ophi = phi/(n+1.0);
 29 266
 30 267                double nBIsum = 0.0;
 31 268                for (int j = 0; j < n; j++)
 32 269                    nBIsum += B_last[(int) neibmat(loc, j) - 1] * I_last
 33                         [(int) neibmat(loc, j) - 1];
 34 270
 35 271                double BSI = S_last[loc]*( sphi*B_last[loc]*I_last[loc] +
 36                         ophi*nBIsum );
 37 272                double rI  = r*I_last[loc];
 38 273
 39 274                // get derivatives
 40 275                double dS = - BSI;
 41 276                double dI = BSI - rI;
 42 277                double dR = rI;
 43 278
 44 279                // step forward by h
 45 280                S[loc] += h*dS;
 46 281                I[loc] += h*dI;
 47 282                R[loc] += h*dR;
 48 283
 49 284            }
 50 285
 51 286        }
```

```
  1  287
  2  288  }
  3  289
  4  290  /*   Initializes particles
  5  291       */
  6  292  void initializeParticles(Particle ** particles, int NP, int nloc, int
  7       N) {
  8  293
  9  294      // allocate space for doubles
 10  295      *particles = (Particle*) malloc (NP*sizeof(Particle));
 11  296
 12  297      // allocate space for arays inside particles
 13  298      for (int n = 0; n < NP; n++) {
 14  299          (*particles)[n].S = (double*) malloc(nloc*sizeof(double));
 15  300          (*particles)[n].I = (double*) malloc(nloc*sizeof(double));
 16  301          (*particles)[n].R = (double*) malloc(nloc*sizeof(double));
 17  302          (*particles)[n].B = (double*) malloc(nloc*sizeof(double));
 18  303          (*particles)[n].Iinit = (double*) malloc(nloc*sizeof(double))
 19          ;
 20  304      }
 21  305
 22  306      // initialize all all parameters
 23  307      for (int n = 0; n < NP; n++) {
 24  308
 25  309          double R0can, rcan, sigmacan, Iinitcan, etacan, berrcan,
 26              phican;
 27  310
 28  311          do {
 29  312              R0can = R0true + R0true*randn();
 30  313          } while (R0can < 0);
 31  314          (*particles)[n].R0 = R0can;
 32  315
 33  316          do {
 34  317              rcan = rtrue + rtrue*randn();
 35  318          } while (rcan < 0);
 36  319          (*particles)[n].r = rcan;
 37  320
 38  321          for (int loc = 0; loc < nloc; loc++)
 39  322              (*particles)[n].B[loc] = (double) R0can * rcan / N;
 40  323
 41  324          do {
 42  325              sigmacan = merr + merr*randn();
 43  326          } while (sigmacan < 0);
 44  327          (*particles)[n].sigma = sigmacan;
 45  328
 46  329          do {
 47  330              etacan = etatrue + PSC*etatrue*randn();
 48  331          } while (etacan < 0 || etacan > 1);
 49  332          (*particles)[n].eta = etacan;
 50  333
 51  334          do {
```

```
335              berrcan = berrtrue + PSC*berrtrue*randn();
336          } while (berrcan < 0);
337          (*particles)[n].berr = berrcan;
338
339          do {
340              phican = phitrue + PSC*phitrue*randn();
341          } while (phican <= 0 || phican >= 1);
342          (*particles)[n].phi = phican;
343
344          for (int loc = 0; loc < nloc; loc++) {
345              do {
346                  Iinitcan = I0 + I0*randn();
347              } while (Iinitcan < 0 || N < Iinitcan);
348              (*particles)[n].Iinit[loc] = Iinitcan;
349          }
350
351      }
352
353 }
354
355 /*  Particle pertubation function to be run between iterations and
        passes
356
357      */
358 void perturbParticles(Particle * particles, int N, int NP, int nloc,
        int passnum, double coolrate) {
359
360      //double coolcoef = exp( - (double) passnum / coolrate );
361      double coolcoef = pow(coolrate, passnum);
362
363      double spreadR0     = coolcoef * R0true / 10.0;
364      double spreadr      = coolcoef * rtrue / 10.0;
365      double spreadsigma  = coolcoef * merr / 10.0;
366      double spreadIinit  = coolcoef * I0 / 10.0;
367      double spreadeta    = coolcoef * etatrue / 10.0;
368      double spreadberr   = coolcoef * berrtrue / 10.0;
369      double spreadphi    = coolcoef * phitrue / 10.0;
370
371      double R0can, rcan, sigmacan, Iinitcan, etacan, berrcan, phican;
372
373      for (int n = 0; n < NP; n++) {
374
375          do {
376              R0can = particles[n].R0 + spreadR0*randn();
377          } while (R0can < 0);
378          particles[n].R0 = R0can;
379
380          do {
381              rcan = particles[n].r + spreadr*randn();
382          } while (rcan < 0);
383          particles[n].r = rcan;
```

```
384
385            do {
386                sigmacan = particles[n].sigma + spreadsigma*randn();
387            } while (sigmacan < 0);
388            particles[n].sigma = sigmacan;
389
390            do {
391                etacan = particles[n].eta + PSC*spreadeta*randn();
392            } while (etacan < 0 || etacan > 1);
393            particles[n].eta = etacan;
394
395            do {
396                berrcan = particles[n].berr + PSC*spreadberr*randn();
397            } while (berrcan < 0);
398            particles[n].berr = berrcan;
399
400            do {
401                phican = particles[n].phi + PSC*spreadphi*randn();
402            } while (phican <= 0 || phican >= 1);
403            particles[n].phi = phican;
404
405            for (int loc = 0; loc < nloc; loc++) {
406                do {
407                    Iinitcan = particles[n].Iinit[loc] + spreadIinit*
                          randn();
408                } while (Iinitcan < 0 || Iinitcan > 500);
409                particles[n].Iinit[loc] = Iinitcan;
410            }
411        }
412
413 }
414
415 /*  Convinience function for particle resampling process
416     */
417 void copyParticle(Particle * dst, Particle * src, int nloc) {
418
419     dst->R0      = src->R0;
420     dst->r       = src->r;
421     dst->sigma   = src->sigma;
422     dst->eta     = src->eta;
423     dst->berr    = src->berr;
424     dst->phi     = src->phi;
425
426     for (int n = 0; n < nloc; n++) {
427         dst->S[n]       = src->S[n];
428         dst->I[n]       = src->I[n];
429         dst->R[n]       = src->R[n];
430         dst->B[n]       = src->B[n];
431         dst->Iinit[n]   = src->Iinit[n];
432     }
433
```

```
434  }
435
436
437  double randu() {
438
439      return (double) rand() / (double) RAND_MAX;
440
441  }
442
443
444  /*  Return a normally distributed random number with mean 0 and
445      standard deviation 1
446      Uses the polar form of the Box-Muller transformation
447      From http://www.design.caltech.edu/erik/Misc/Gaussian.html
448      */
449  double randn() {
450
451      double x1, x2, w, y1;
452
453      do {
454          x1 = 2.0 * randu() - 1.0;
455          x2 = 2.0 * randu() - 1.0;
456          w = x1 * x1 + x2 * x2;
457      } while ( w >= 1.0 );
458
459      w = sqrt( (-2.0 * log( w ) ) / w );
460      y1 = x1 * w;
461
462      return y1;
463
      }
```

F.5   CUDA IF2 Spatial Fitting Code

Below is the nascent CUDA code that will be expanded upon in future work. At
present it only implements the core IF2 fitting algorithm and does not implement
parametric bootstrapping nor produce forecasts.

```
1   /*  Dexter Barrows
2       dbarrows.github.io
3       McMaster University
4       2016
5
6       */
7
8   #include <cuda.h>
9   #include <iostream>
10  #include <fstream>
```

```
11  #include <curand.h>
12  #include <curand_kernel.h>
13  #include <string>
14  #include <sstream>
15  #include <cmath>
16
17  #include "timer.h"
18  #include "rand.h"
19  #include "readdata.h"
20
21  #define NP          (2*2500)    // number of particles
22  #define N           500.0       // population size
23  #define R0true      3.0         // infectiousness
24  #define rtrue       0.1         // recovery rate
25  #define etatrue     0.5         // real drift attraction strength
26  #define berrtrue    0.5         // real beta drift noise
27  #define phitrue     0.5         // real connectivity strength
28  #define merr        10.0        // expected measurement error
29  #define I0          5.0         // Initial infected individuals
30  #define PSC         0.5         // sensitive parameter perturbation
        scaling
31  #define NLOC        10
32
33  #define PI      3.141592654f
34
35  // Wrapper for CUDA calls, from CUDA API
36  // Modified to also print the error code and string
37  # define CUDA_CALL(x) do { if ((x) != cudaSuccess ) {
          \
38      std::cout << " Error at " << __FILE__ << ":" << __LINE__ << std::
            endl;      \
39      std::cout << " Error was " << x << " " << cudaGetErrorString(x)
            << std::endl;   \
40      return EXIT_FAILURE ;}} while (0)
                \
41
42  typedef struct {
43      float R0;
44      float r;
45      float sigma;
46      float eta;
47      float berr;
48      float phi;
49      float S[NLOC];
50      float I[NLOC];
51      float R[NLOC];
52      float B[NLOC];
53      float Iinit[NLOC];
54      curandState randState;  // PRNG state
55  } Particle;
56
```

```
57   __host__ std::string getHRmemsize (size_t memsize);
58   __host__ std::string getHRtime (float runtime);
59
60   __device__ void exp_euler_SSIR(float h, float t0, float tn, Particle
         * particle, int * neinum, int * neibmat, int nloc);
61   __device__ void copyParticle(Particle * dst, Particle * src, int nloc
         );
62
63
64   /*   Initialize all PRNG states, get starting state vector using
         initial distribution
65      */
66   __global__ void initializeParticles (Particle * particles, int nloc)
         {
67
68       int id  = blockIdx.x*blockDim.x + threadIdx.x;   // global thread
             ID
69
70       if (id < NP) {
71
72           // initialize PRNG state
73           curandState state;
74           curand_init(id, 0, 0, &state);
75
76           float R0can, rcan, sigmacan, Iinitcan, etacan, berrcan,
                 phican;
77
78           do {
79               R0can = R0true + R0true*curand_normal(&state);
80           } while (R0can < 0);
81           particles[id].R0 = R0can;
82
83           do {
84               rcan = rtrue + rtrue*curand_normal(&state);
85           } while (rcan < 0);
86           particles[id].r = rcan;
87
88           for (int loc = 0; loc < nloc; loc++)
89               particles[id].B[loc] = (float) R0can * rcan / N;
90
91           do {
92               sigmacan = merr + merr*curand_normal(&state);
93           } while (sigmacan < 0);
94           particles[id].sigma = sigmacan;
95
96           do {
97               etacan = etatrue + PSC*etatrue*curand_normal(&state);
98           } while (etacan < 0 || etacan > 1);
99           particles[id].eta = etacan;
100
101           do {
```

```
102            berrcan = berrtrue + PSC*berrtrue*curand_normal(&state);
103        } while (berrcan < 0);
104        particles[id].berr = berrcan;
105
106        do {
107            phican = phitrue + PSC*phitrue*curand_normal(&state);
108        } while (phican <= 0 || phican >= 1);
109        particles[id].phi = phican;
110
111        for (int loc = 0; loc < nloc; loc++) {
112            do {
113                Iinitcan = I0 + I0*curand_normal(&state);
114            } while (Iinitcan < 0 || N < Iinitcan);
115            particles[id].Iinit[loc] = Iinitcan;
116        }
117
118        particles[id].randState = state;
119
120    }
121
122 }
123
124 __global__ void resetStates (Particle * particles, int nloc) {
125
126    int id  = blockIdx.x*blockDim.x + threadIdx.x;  // global thread
            ID
127
128    if (id < NP) {
129
130        for (int loc = 0; loc < nloc; loc++) {
131            particles[id].S[loc] = N - particles[id].Iinit[loc];
132            particles[id].I[loc] = particles[id].Iinit[loc];
133            particles[id].R[loc] = 0.0;
134        }
135
136    }
137
138 }
139
140 __global__ void clobberParams (Particle * particles, int nloc) {
141
142    int id  = blockIdx.x*blockDim.x + threadIdx.x;  // global thread
            ID
143
144    if (id < NP) {
145
146        particles[id].R0 = R0true;
147        particles[id].r = rtrue;
148        particles[id].sigma = merr;
149        particles[id].eta = etatrue;
150        particles[id].berr = berrtrue;
```

```
151          particles[id].phi = phitrue;
152
153          for (int loc = 0; loc < nloc; loc++) {
154              particles[id].Iinit[loc] = I0;
155          }
156
157      }
158
159 }
160
161
162 /*  Project particles forward , perturb , and save weight based on data
163      int t - time step number (1,...,T)
164      */
165 __global__ void project (Particle * particles, int * neinum, int *
166      neibmat, int nloc) {
167
168      int id = blockIdx.x*blockDim.x + threadIdx.x;   // global id
169
170      if (id < NP) {
171          // project forward
172          exp_euler_SSIR(1.0/7.0, 0.0, 1.0, &particles[id], neinum,
173              neibmat, nloc);
174      }
175
176 }
177
178 __global__ void weight(float * data, Particle * particles, double * w
179      , int t, int T, int nloc) {
180
181      int id = blockIdx.x*blockDim.x + threadIdx.x;   // global id
182
183      if (id < NP) {
184
185          float merr_par = particles[id].sigma;
186
187          // Get weight and save
188          double w_local = 1.0;
189          for (int loc = 0; loc < nloc; loc++) {
190              float y_diff = data[loc*T + t] - particles[id].I[loc];
191              w_local *= 1.0/(merr_par*sqrt(2.0*PI)) * exp( - y_diff*
192                  y_diff / (2.0*merr_par*merr_par) );
193          }
194
195          w[id] = w_local;
196
197      }
198
199 }
200
201 __global__ void stashParticles (Particle * particles, Particle *
```

```
1          particles_old, int nloc) {
2   198
3   199     int id = blockIdx.x*blockDim.x + threadIdx.x;   // global id
4   200
5   201     if (id < NP) {
6   202         // COPY PARTICLE
7   203         copyParticle(&particles_old[id], &particles[id], nloc);
8   204     }
9   205
10  206 }
11  207
12  208
13  209 /*  The 0th thread will perform cumulative sum on the weights.
14  210     There may be a faster way to do this, will investigate.
15  211     */
16  212 __global__ void cumsumWeights (double * w) {
17  213
18  214     int id  = blockIdx.x*blockDim.x + threadIdx.x;  // global thread
19            ID
20  215
21  216     // compute cumulative weights
22  217     if (id == 0) {
23  218         for (int i = 1; i < NP; i++)
24  219             w[i] += w[i-1];
25  220     }
26  221
27  222 }
28  223
29  224
30  225 /*  Resample from all particle states within cell
31  226     */
32  227 __global__ void resample (Particle * particles, Particle *
33        particles_old, double * w, int nloc) {
34  228
35  229     int id  = blockIdx.x*blockDim.x + threadIdx.x;
36  230
37  231     if (id < NP) {
38  232
39  233         // resampling proportional to weights
40  234         double w_r = curand_uniform(&particles[id].randState) * w[NP
41              -1];
42  235         int i = 0;
43  236         while (w_r > w[i]) {
44  237             i++;
45  238         }
46  239
47  240         // i is now the index of the particle to copy from
48  241         copyParticle(&particles[id], &particles_old[i], nloc);
49  242
50  243     }
51  244
```

```
245  }
246
247  // launch this with probably just nloc threads... block structure/
         size probably not important
248  __global__ void reduceStates (Particle * particles, float *
         countmeans, int t, int T, int nloc) {
249
250      int id  = blockIdx.x*blockDim.x + threadIdx.x;
251
252      if (id < nloc) {
253
254          int loc = id;
255
256          double countmean_local = 0.0;
257          for (int n = 0; n < NP; n++) {
258              countmean_local += particles[n].I[loc] / NP;
259          }
260
261          countmeans[loc*T + t] = (float) countmean_local;
262
263      }
264
265  }
266
267  __global__ void perturbParticles(Particle * particles, int nloc, int
         passnum, double coolrate) {
268
269      //double coolcoef = exp( - (double) passnum / coolrate );
270      double coolcoef = pow(coolrate, passnum);
271
272      double spreadR0     = coolcoef * R0true / 10.0;
273      double spreadr      = coolcoef * rtrue / 10.0;
274      double spreadsigma  = coolcoef * merr / 10.0;
275      double spreadIinit  = coolcoef * I0 / 10.0;
276      double spreadeta    = coolcoef * etatrue / 10.0;
277      double spreadberr   = coolcoef * berrtrue / 10.0;
278      double spreadphi    = coolcoef * phitrue / 10.0;
279
280      double R0can, rcan, sigmacan, Iinitcan, etacan, berrcan, phican;
281
282      int id  = blockIdx.x*blockDim.x + threadIdx.x;
283
284      if (id < NP) {
285
286          do {
287              R0can = particles[id].R0 + spreadR0*curand_normal(&
                     particles[id].randState);
288          } while (R0can < 0);
289          particles[id].R0 = R0can;
290
291          do {
```

```
292                    rcan = particles[id].r + spreadr*curand_normal(&particles
                            [id].randState);
293             } while (rcan < 0);
294             particles[id].r = rcan;
295
296             do {
297                 sigmacan = particles[id].sigma + spreadsigma*
                        curand_normal(&particles[id].randState);
298             } while (sigmacan < 0);
299             particles[id].sigma = sigmacan;
300
301             do {
302                 etacan = particles[id].eta + PSC*spreadeta*curand_normal
                        (&particles[id].randState);
303             } while (etacan < 0 || etacan > 1);
304             particles[id].eta = etacan;
305
306             do {
307                 berrcan = particles[id].berr + PSC*spreadberr*
                        curand_normal(&particles[id].randState);
308             } while (berrcan < 0);
309             particles[id].berr = berrcan;
310
311             do {
312                 phican = particles[id].phi + PSC*spreadphi*curand_normal
                        (&particles[id].randState);
313             } while (phican <= 0 || phican >= 1);
314             particles[id].phi = phican;
315
316             for (int loc = 0; loc < nloc; loc++) {
317                 do {
318                     Iinitcan = particles[id].Iinit[loc] + spreadIinit*
                            curand_normal(&particles[id].randState);
319                 } while (Iinitcan < 0 || Iinitcan > 500);
320                 particles[id].Iinit[loc] = Iinitcan;
321             }
322
323         }
324
325 }
326
327
328 int main (int argc, char *argv[]) {
329
330
331     int T, nloc;
332
333     double restime;
334     struct timeval tdr0, tdr1, tdrMaster;
335
336     // Parse arguments ********************************************
```

137

```
337
338     if (argc < 4) {
339         std::cout << "Not enough arguments" << std::endl;
340         return 0;
341     }
342
343     std::string arg1(argv[1]);  // infection counts
344     std::string arg2(argv[2]);  // neighbour counts
345     std::string arg3(argv[3]);  // neighbour indices
346     std::string arg4(argv[4]);  // outfile: params + runtime
347
348     std::cout << "Arguments:" << std::endl;
349     std::cout << "Infection data:    " << arg1 << std::endl;
350     std::cout << "Neighbour counts:  " << arg2 << std::endl;
351     std::cout << "Neighbour indices: " << arg3 << std::endl;
352     std::cout << "Outfile            " << arg4 << std::endl;
353
354     // ****************************************************************
355
356
357     // Read count data ***********************************************
358
359     std::cout << "Getting count data" << std::endl;
360     float * data = getDataFloat(arg1, &T, &nloc);
361     size_t datasize = nloc*T*sizeof(float);
362
363     // ****************************************************************
364
365     // Read neinum matrix data ***************************************
366
367     std::cout << "Getting neighbour count data" << std::endl;
368     int * neinum = getDataInt(arg2, NULL, NULL);
369     size_t neinumsize = nloc * sizeof(int);
370
371     // ****************************************************************
372
373     // Read neibmat matrix data **************************************
374
375     std::cout << "Getting neighbour count data" << std::endl;
376     int * neibmat = getDataInt(arg3, NULL, NULL);
377     size_t neibmatsize = nloc * nloc * sizeof(int);
378
379     // ****************************************************************
380
381     //
        ****************************************************************************

382
383     // start timing
384     gettimeofday (&tdr0, NULL);
385
```

138

```
386      // CUDA data **********************************************
387
388      std::cout << "Allocating device storage" << std::endl;
389
390      float       * d_data;               // device copy of data
391      Particle    * particles;            // particles
392      Particle    * particles_old;        // intermediate particle states
393      double      * w;                    // weights
394      int         * d_neinum;             // device copy of adjacency
             matrix
395      int         * d_neibmat;            // device copy of neighbour
             counts matrix
396      float       * countmeans;           // host copy of reduced infection
              count means from last pass
397      float       * d_countmeans;         // device copy of reduced
             infection count means from last pass
398
399      CUDA_CALL( cudaMalloc( (void**) &d_data          , datasize )
                 );
400      CUDA_CALL( cudaMalloc( (void**) &particles       , NP*sizeof(
             Particle))  );
401      CUDA_CALL( cudaMalloc( (void**) &particles_old   , NP*sizeof(
             Particle))  );
402      CUDA_CALL( cudaMalloc( (void**) &w               , NP*sizeof(
             double))    );
403      CUDA_CALL( cudaMalloc( (void**) &d_neinum        , neinumsize)
                 );
404      CUDA_CALL( cudaMalloc( (void**) &d_neibmat       , neibmatsize)
                 );
405      CUDA_CALL( cudaMalloc( (void**) &d_countmeans    , nloc*T*sizeof(
             float)) );
406
407
408      gettimeofday (&tdr1, NULL);
409      timeval_subtract (&restime, &tdr1, &tdr0);
410
411      std::cout << "\t" << getHRtime(restime) << std::endl;
412
413      size_t avail, total;
414      cudaMemGetInfo( &avail, &total );
415      size_t used = total - avail;
416
417      std::cout << "\t[" << getHRmemsize(used) << "] used of [" <<
             getHRmemsize(total) << "]" <<std::endl;
418
419      std::cout << "Copying data to device" << std::endl;
420
421      gettimeofday (&tdr0, NULL);
422
423      CUDA_CALL( cudaMemcpy(d_data    , data      , datasize       ,
             cudaMemcpyHostToDevice)   );
```

```
424      CUDA_CALL( cudaMemcpy(d_neinum  , neinum    , neinumsize    ,
             cudaMemcpyHostToDevice)   );
425      CUDA_CALL( cudaMemcpy(d_neibmat , neibmat   , neibmatsize   ,
             cudaMemcpyHostToDevice)   );
426
427      gettimeofday (&tdr1, NULL);
428      timeval_subtract (&restime, &tdr1, &tdr0);
429
430      std::cout << "\t" << getHRtime(restime) << std::endl;
431
432      // ***************************************************************
433
434
435
436      // Initialize particles *****************************************
437
438      std::cout << "Initializing particles" << std::endl;
439
440      //gettimeofday (&tdr0, NULL);
441
442      int nThreads    = 32;
443      int nBlocks     = ceil( (float) NP / nThreads);
444
445      initializeParticles <<< nBlocks, nThreads >>> (particles, nloc);
446      CUDA_CALL( cudaGetLastError() );
447      CUDA_CALL( cudaDeviceSynchronize() );
448
449      initializeParticles <<< nBlocks, nThreads >>> (particles_old,
             nloc);
450      CUDA_CALL( cudaGetLastError() );
451      CUDA_CALL( cudaDeviceSynchronize() );
452
453      //gettimeofday (&tdr1, NULL);
454      //timeval_subtract (&restime, &tdr1, &tdr0);
455      //std::cout << "\t" << getHRtime(restime) << std::endl;
456
457      cudaMemGetInfo( &avail, &total );
458      used = total - avail;
459      std::cout << "\t[" << getHRmemsize(used) << "] used of [" <<
             getHRmemsize(total) << "]" <<std::endl;
460
461      // ***************************************************************
462
463      // Starting filtering *******************************************
464
465      for (int pass = 0; pass < 50; pass++) {
466
467          nThreads    = 32;
468          nBlocks     = ceil( (float) NP / nThreads);
469
470          resetStates <<< nBlocks, nThreads >>> (particles, nloc);
```

```
471            CUDA_CALL( cudaGetLastError() );
472            CUDA_CALL( cudaDeviceSynchronize() );
473
474            nThreads = 1;
475            nBlocks  = 10;
476
477            if (pass == 49) {
478                reduceStates <<< nBlocks, nThreads >>> (particles,
479                    d_countmeans, 0, T, nloc);
480                CUDA_CALL( cudaGetLastError() );
480                CUDA_CALL( cudaDeviceSynchronize() );
481            }
482
483            int Tlim = T;
484
485            for (int t = 1; t < Tlim; t++) {
486
487                // Projection
                       **************************************************
488
489                nThreads    = 32;
490                nBlocks     = ceil( (float) NP / nThreads);
491
492                project <<< nBlocks, nThreads >>> (particles, d_neinum,
493                    d_neibmat, nloc);
493                CUDA_CALL( cudaGetLastError() );
494                CUDA_CALL( cudaDeviceSynchronize() );
495
496                // Weighting
                       **************************************************
497
498                nThreads    = 32;
499                nBlocks     = ceil( (float) NP / nThreads);
500
501                weight <<< nBlocks, nThreads >>>(d_data, particles, w, t,
                           T, nloc);
502                CUDA_CALL( cudaGetLastError() );
503                CUDA_CALL( cudaDeviceSynchronize() );
504
505                // Cumulative sum
                       *********************************************
506
507                nThreads    = 1;
508                nBlocks     = 1;
509
510                cumsumWeights <<< nBlocks, nThreads >>> (w);
511                CUDA_CALL( cudaGetLastError() );
512                CUDA_CALL( cudaDeviceSynchronize() );
513
514                // Save particles for resampling from
                       *************************
```

```
515
516            nThreads    = 32;
517            nBlocks     = ceil( (float) NP / nThreads);
518
519            stashParticles <<< nBlocks, nThreads >>> (particles,
                   particles_old, nloc);
520            CUDA_CALL( cudaGetLastError() );
521            CUDA_CALL( cudaDeviceSynchronize() );
522
523
524            // Resampling
                   **************************************************
525
526            nThreads    = 32;
527            nBlocks     = ceil( (float) NP/ nThreads);
528
529            resample <<< nBlocks, nThreads >>> (particles,
                   particles_old, w, nloc);
530            CUDA_CALL( cudaGetLastError() );
531            CUDA_CALL( cudaDeviceSynchronize() );
532
533            // Reduction
                   **************************************************
534
535            if (pass == 49) {
536
537                 nThreads = 1;
538                 nBlocks  = 10;
539
540                 reduceStates <<< nBlocks, nThreads >>> (particles,
                       d_countmeans, t, T, nloc);
541                CUDA_CALL( cudaGetLastError() );
542                CUDA_CALL( cudaDeviceSynchronize() );
543
544            }
545
546            // Perturb particles
                   *******************************************
547
548            nThreads    = 32;
549            nBlocks     = ceil( (float) NP/ nThreads);
550
551            perturbParticles <<< nBlocks, nThreads >>> (particles,
                   nloc, pass, 0.975);
552            CUDA_CALL( cudaGetLastError() );
553            CUDA_CALL( cudaDeviceSynchronize() );
554
555
556        } // end time
557
558    } // end pass
```

```
1   559
2   560        std::cout.precision(10);
3   561
4   562        countmeans = (float*) malloc (nloc*T*sizeof(float));
5   563        cudaMemcpy(countmeans, d_countmeans, nloc*T*sizeof(float),
6                  cudaMemcpyDeviceToHost);
7   564
8   565        // stop master timer and print
9   566
10  567        gettimeofday (&tdrMaster, NULL);
11  568        timeval_subtract(&restime, &tdrMaster, &tdr0);
12  569        std::cout << "Time: " << getHRtime(restime) << std::endl;
13  570        std::cout << "Rawtime: " << restime << std::endl;
14  571
15  572        // Write results out
16  573
17  574        std::string filename = arg4;
18  575
19  576        std::cout << "Writing results to file '" << filename << "' ..."
20                  << std::endl;
21  577
22  578        std::ofstream outfile;
23  579        outfile.open(filename.c_str());
24  580
25  581        for(int loc = 0; loc < nloc; loc++) {
26  582            for (int t = 0; t < T; t++) {
27  583                outfile << countmeans[loc*T + t] << " ";
28  584            }
29  585            outfile << std::endl;
30  586        }
31  587
32  588        outfile.close();
33  589
34  590        cudaFree(d_data);
35  591        cudaFree(particles);
36  592        cudaFree(particles_old);
37  593        cudaFree(w);
38  594        cudaFree(d_neinum);
39  595        cudaFree(d_neibmat);
40  596        cudaFree(d_countmeans);
41  597
42  598        exit (EXIT_SUCCESS);
43  599
44  600   }
45  601
46  602
47  603   /*  Use the Explicit Euler integration scheme to integrate SIR model
48             forward in time
49  604        float h    - time step size
50  605        float t0   - start time
51  606        float tn   - stop time
```

```
607        float * y   - current system state; a three-component vector
               representing [S I R], susceptible-infected-recovered
608        */
609 __device__ void exp_euler_SSIR(float h, float t0, float tn, Particle
        * particle, int * neinum, int * neibmat, int nloc) {
610
611        int num_steps = floor( (tn-t0) / h );
612
613        float * S = particle->S;
614        float * I = particle->I;
615        float * R = particle->R;
616        float * B = particle->B;
617
618        // create last state vectors
619        float * S_last = (float*) malloc (nloc*sizeof(float));
620        float * I_last = (float*) malloc (nloc*sizeof(float));
621        float * R_last = (float*) malloc (nloc*sizeof(float));
622        float * B_last = (float*) malloc (nloc*sizeof(float));
623
624        float R0    = particle->R0;
625        float r     = particle->r;
626        float B0    = R0 * r / N;
627        float eta   = particle->eta;
628        float berr  = particle->berr;
629        float phi   = particle->phi;
630
631        for(int t = 0; t < num_steps; t++) {
632
633            for (int loc = 0; loc < nloc; loc++) {
634                S_last[loc] = S[loc];
635                I_last[loc] = I[loc];
636                R_last[loc] = R[loc];
637                B_last[loc] = B[loc];
638            }
639
640            for (int loc = 0; loc < nloc; loc++) {
641
642                B[loc] = exp( log(B_last[loc]) + eta*(log(B0) - log(
                    B_last[loc])) + berr*curand_normal(&(particle->
                    randState)) );
643
644                int n = neinum[loc];
645                float sphi = 1.0 - phi*( (float) n/(n+1.0) );
646                float ophi = phi/(n+1.0);
647
648                float nBIsum = 0.0;
649                for (int j = 0; j < n; j++)
650                    nBIsum += B_last[neibmat[nloc*loc + j]-1] * I_last[
                        neibmat[nloc*loc + j]-1];
651
652                float BSI = S_last[loc]*( sphi*B_last[loc]*I_last[loc] +
```

144

```
                        ophi*nBIsum );
653             float rI  = r*I_last[loc];
654
655             // get derivatives
656             float dS = - BSI;
657             float dI = BSI - rI;
658             float dR = rI;
659
660             // step forward by h
661             S[loc] += h*dS;
662             I[loc] += h*dI;
663             R[loc] += h*dR;
664
665         }
666
667     }
668
669     free(S_last);
670     free(I_last);
671     free(R_last);
672     free(B_last);
673
674 }
675
676 /*  Convinience function for particle resampling process
677     */
678 __device__ void copyParticle(Particle * dst, Particle * src, int nloc
     ) {
679
680     dst->R0     = src->R0;
681     dst->r      = src->r;
682     dst->sigma  = src->sigma;
683     dst->eta    = src->eta;
684     dst->berr   = src->berr;
685     dst->phi    = src->phi;
686
687     for (int n = 0; n < nloc; n++) {
688         dst->S[n]       = src->S[n];
689         dst->I[n]       = src->I[n];
690         dst->R[n]       = src->R[n];
691         dst->B[n]       = src->B[n];
692         dst->Iinit[n]   = src->Iinit[n];
693     }
694
695 }
696
697 /*  Convert memory size in bytes to human-readable format
698     */
699 std::string getHRmemsize (size_t memsize) {
700
701     std::stringstream ss;
```

```
702     std::string valstring;
703
704     int kb = 1024;
705     int mb = kb*1024;
706     int gb = mb*1024;
707
708     if (memsize <= kb)
709         ss << memsize << " B";
710     else if (memsize > kb && memsize <= mb)
711         ss << (float) memsize/ kb << " KB";
712     else if (memsize > mb && memsize <= gb)
713         ss << (float) memsize/ mb << " MB";
714     else
715         ss << (float) memsize/ gb << " GB";
716
717     valstring = ss.str();
718
719     return valstring;
720
721 }
722
723
724 /*  Convert time in seconds to human readable format
725     */
726 std::string getHRtime (float runtime) {
727
728     std::stringstream ss;
729     std::string valstring;
730
731     int mt = 60;
732     int ht = mt*60;
733     int dt = ht*24;
734
735     if (runtime <= mt)
736         ss << runtime << " s";
737     else if (runtime > mt && runtime <= ht)
738         ss << runtime/mt << " m";
739     else if (runtime > ht && runtime <= dt)
740         ss << runtime/dt << " h";
741     else
742         ss << runtime/ht << " d";
743
744     valstring = ss.str();
745
746     return valstring;
747
748 }
```

Figure [F.1] shows the running times for parameter fitting as compared to IF2 and HMC.
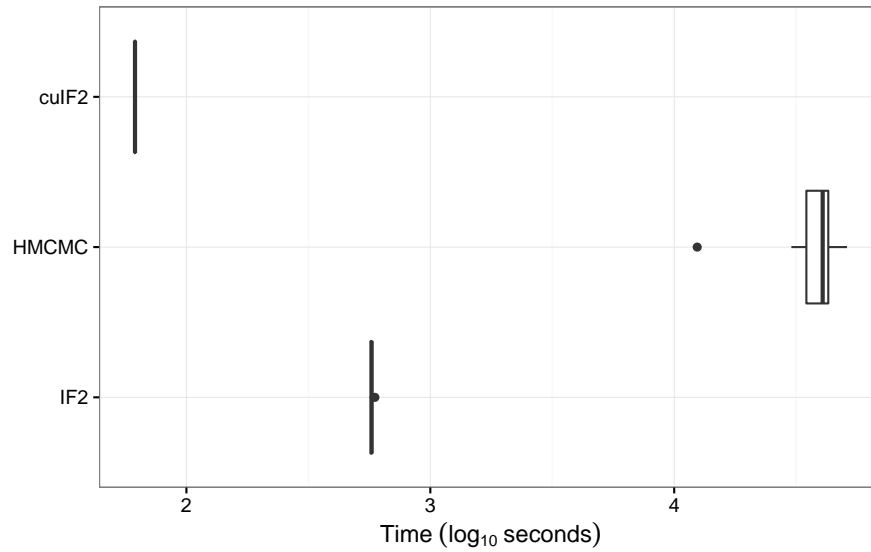
Figure F.1: Running times for fitting the spatial SIR model to data.

1 The means from the data in Figure [F.1] are about 61.5 seconds for `cuIF2`, 574 seconds
2 for IF2, and $38,800$ seconds for HMC. For `cuIF2` This is a speedup of over 9.33x against
3 IF2 and over 617x against HMC.