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)
(Mathematics)

McMaster University
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, 148

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][30][42]. 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 [30]. 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 [36], 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 [25]. 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)

model. ARIMA assumes a linear underlying process and Gaussian error distributions. It uses three parameters representing the degree of autoregression (p), integration (trend removal) (d), and the moving average (q), where the orders of the autoregression and the moving average are determined through the use of an autocorrelation function (ACF) and partial autocorrelation function (PACF), respectively, applied to the data *a priori* [43].

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

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

where S, I, and R are the number of individuals in each compartment, β is contact rate between susceptible and infected individuals, and γ is a recovery rate. We also let $\beta = \mathcal{R}_0\gamma/N$, where $\mathcal{R}_0$ is the number of secondary cases per infected individual in a wholly susceptible population, and N is the population size. As an outbreak progresses, individuals transition from the susceptible compartment, through the infectious compartment, then finish in the removed compartment where they no longer impact the system dynamics. Many extensions of the SIR model exist and are commonly used, such as the SEIR model in which susceptible individuals pass through an exposed class (or several) where they have been infected but are not yet themselves infectious, and the SIRS model in which individuals become susceptible again after their immunity wanes [7][12].

Combining the phenomenological and mechanistic approaches are data integration schemes. These methods use a model to define the expected underlying dynamics of the system, but integrate data into the model in order to refine estimates of the model parameters and produce more accurate forecasts. Typically the first step in implementing such a technique is fitting the desired model to existing data. There are many ways to do this, most of which fall into two main categories: Sequential Monte Carlo (SMC) methods [3][34][42], and Markov chain Monte Carlo-based (MCMC) methods [2][29]. From there data can either be integrated into the model by refitting the model to the new longer data set, or in an "on-line" fashion in which data points

can be directly integrated without the need to refit the entire model. Normally, MCMC-based machinery must refit the entire model whereas SMC-based approaches can sometimes integrate data in an on-line fashion, thus "on-line" methods are most appropriate when data must be integrated in "real-time".

Another, broader, distinction among techniques can be drawn between those that rely on assumptions of linearity, and those that make no such assumption. As epidemic dynamics are highly non-linear, it can be questionable as to even consider linear approaches to epidemic forecasting at all. In particular, stalwart approaches such as ARIMA and the venerable Kalman filter face a distinct (at least theoretical) disadvantage in the face of newer SMC-based methods [39][42]. Extensions of the Kalman filter, such as the ensemble adjustment Kalman filter are designed to handle non-linearity, but these methods are very well-studied, and further work showing their viability would likely prove extraneous in the modern academic landscape.

Somewhat frustratingly, there exists no "gold standard" in forecasting [9][30][42]. As methodology varies widely in theoretical justification, implementation, and operation, it is difficult to compare the state of the art in forecasting methods from a first-principles perspective. Further, published work making use of any of these methods to forecast uses different prediction accuracy metrics, such as the sum of squared errors of prediction (SSE), peak time/duration/intensity, correlation tests, or root-mean-square error of prediction (RMSE), among others [9][31]. Thus it is difficult to select the best tool for the job when faced with a forecasting problem.

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

From MCMC-based methods we have selected Hamiltonian MCMC (HMC) [29], a (slightly) less cutting-edge but nonetheless highly effective technique. We are using HMC through an implementation in the R package RStan [8], which at its core uses HMC, but also contains implementations of several other innovative techniques. Interestingly, the original goal of this package was not to implement a statistical programming language similar to Just Another Gibbs Sampler (JAGS) or Bayesian Inference Using Gibbs Sampling (BUGS), but with an HMC backend. In fact the developers' original goal was to implement any method that could fit multilevel hierarchical models without halting as they were witnessing with BUGS and JAGS. Only after experimenting with several options and starting to hear about it more and more frequently did they attempt to work with HMC. In the end, the scope of the project grew to include the development and subsequent integration of the No-U-Turn Sampler (NUTS) [20], and an implementation of automatic differentiation machinery

[37].

For PF-based methods we have selected Iterated Filtering (IF) 2 [23], 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 [22], 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 [13]. The developers of MIF and IF2 have their own R package that implements MIF and IF2, Partially Observed Markov Processes (POMP) [24][26], 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) [15][21][38][39], combined with Dewdrop Regression [15]. 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 thesis 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 thesis in Chapter 8.

Chapter 2

Hamiltonian MCMC

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

2.1 Markov Chains

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

The transition probabilities can be summarized as a matrix as

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

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

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

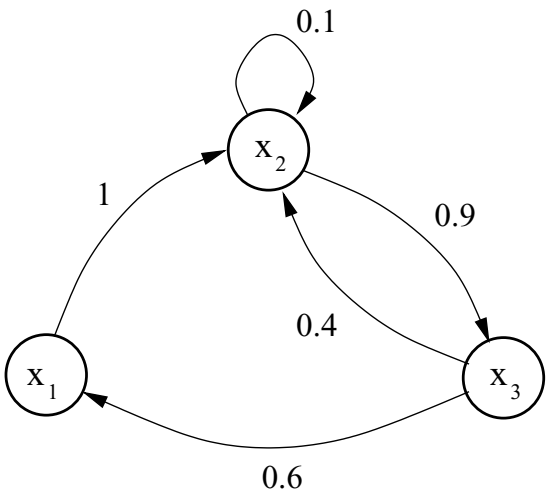

Figure 2.1: A finite state machine. States are shown as graph nodes, and the probability of transitioning from one particular state to another is shown as a weighted graph edge. [2]

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;

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

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

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

2.3 Prior distribution

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

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

2.4 Proposal distribution

As part of the MCMC algorithm, when we find a state in the parameter space that is accepted as part of the Markov chain construction process, we need a good way of generating a good next step to try. Unlike basic rejection sampling in which we would just randomly sample from our prior distribution, MCMC attempts to optimise

our choices by choosing a step that is close enough to the last accepted step so as to stand a decent chance of also being accepted, but far enough away that it doesn't get "trapped" in a particular region of the parameter space.

This is done through the use of a proposal or candidate distribution. This will usually be a distribution centred around our last accepted step and with a dispersion potential narrower than that of 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 θ^(1)
for i = 2 : N do
    /* Sample                                                       */
    θ* ~ q(·|θ^(i−1))
    u ~ U(0, 1)
    /* Evaluate acceptance ratio                                    */
    r ← L(θ*)p(θ*) / L(θ)p(θ)
    /* Step acceptance criterion                                    */
    if u < min{1, r} then
        θ^(i) = θ*
    else
        θ^(i) = θ^(i−1)
/* Samples from approximated posterior distribution                 */
Output: Chain of samples (θ^(1), θ^(2), ..., θ^(N))
```

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 requires less room to store [27].

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) [29]. 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 [29]. HMC has in fact existed for nearly the same amount of time as MCMC – both methods having been developed to model molecular dynamics, with MCMC taking a probabilistic approach and HMC taking a more deterministic one – but had not received much attention outside its native discipline until recently.

In the HMC formulation, the parameter estimates are treated as a physical particle exploring a sloped likelihood surface. From physics, we will borrow the ideas of potential and kinetic energy. Here potential energy, or gravity, 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 as it will not preserve the volume of the system. 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 $\epsilon/2$ 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. This form of the acceptance ratio comes from the definition of the Hamiltonian as an energy function. If we define the distribution of the total potential energy in the system (known as the canonical distribution) as a function of the Hamiltonian as

$$P(\theta, r) = \frac{1}{Z}\exp(-H(\theta, r)) \tag{2.11}$$

where Z is a normalizing constant, then taking the ratio of the total potential energy of the proposed step $P(\theta^*, r^*)$ to the total potential energy in the last accepted step $P(\theta, r)$, we obtain Equation (2.10).

Algorithm 2: Hamiltonian MCMC

```
/* Select a starting point */
Input : Initialize θ^(1)
for i = 2 : N do
    /* Resample moments */
    for i = 1 : n do
        r(i) ← N(0, 1)
    /* Leapfrog initialization */
    θ_0 ← θ^(i−1)
    r_0 ← r − ∇U(θ_0) · ε/2
    /* Leapfrog intermediate steps */
    for j = 1 : L − 1 do
        θ_j ← θ_{j−1} + M^{−1} r_{j−1} · ε
        r_j ← r_{j−1} − ∇U(θ_j) · ε
    /* Leapfrog last steps */
    θ* ← θ_{L−1} + M^{−1} r_{L−1} · ε
    r* ← ∇U(θ_L) · ε/2 − r_{L−1}
    /* Evaluate acceptance ratio */
    r = exp[H(θ^(i−1), r) − H(θ*, r*)]
    /* Sample */
    u ∼ U(0, 1)
    /* Step acceptance criterion */
    if u < min{1, r} then
        θ^(i) = θ*
    else
        θ^(i) = θ^(i−1)
/* Samples from approximated posterior distribution */
Output: Chain of samples (θ^(1), θ^(2), ..., θ^(N))
```

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 utilising trial fitting with candidate values of ϵ and L [29]. However, some recent algorithms, such as the No U-Turn sampler implemented in RStan, and adaptively select appropriate values automatically during the sampling process [20].

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 $\mathcal{R}_0 = 3.0$, $\gamma = 0.1$ recoveries/week, $N = 500$ individuals. 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 $[0, 100]$ weeks 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

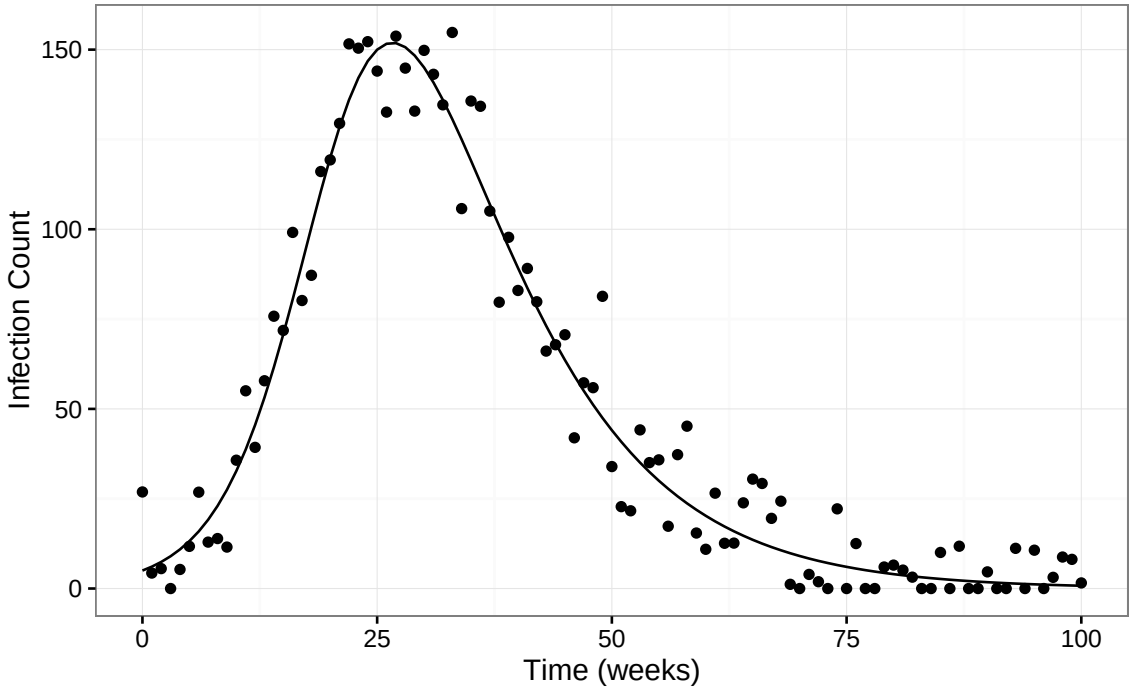

Figure 2.2: True SIR ODE solution infected counts, and with added observation noise.

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, with a step size of one per day, 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 [2.4] shows the chain data including the burn-in samples. We can see why it is wise to discard these samples (note the scale).

Figure [2.5] shows the the kernel density estimates for each of the model parameters and the initial number of cases. We see that while the estimates are not perfect, they are more than satisfactory.

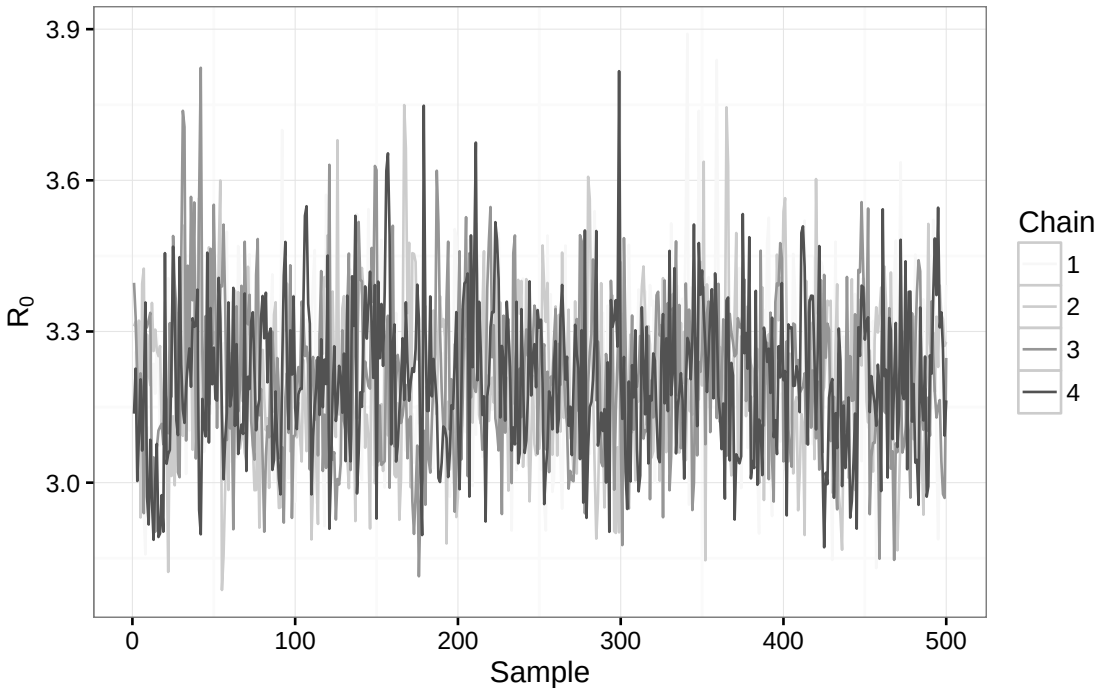

Figure 2.3: Traceplot of samples drawn for parameter $\mathcal{R}_0$, excluding burn-in.

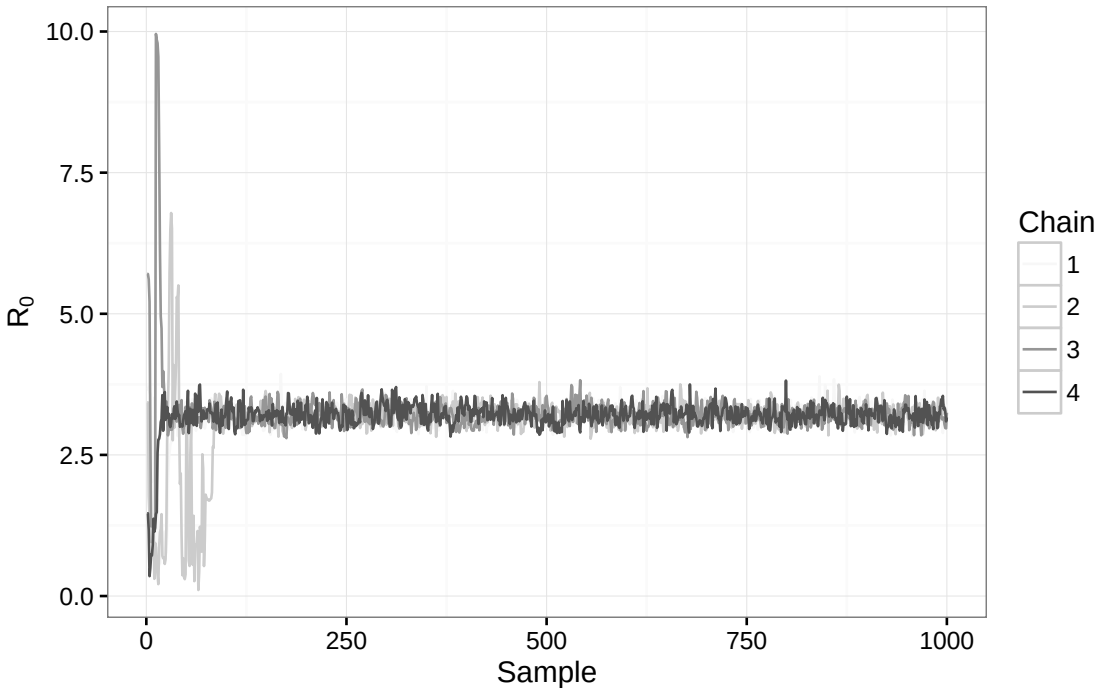

Figure 2.4: Traceplot of samples drawn for parameter $\mathcal{R}_0$, including burn-in.

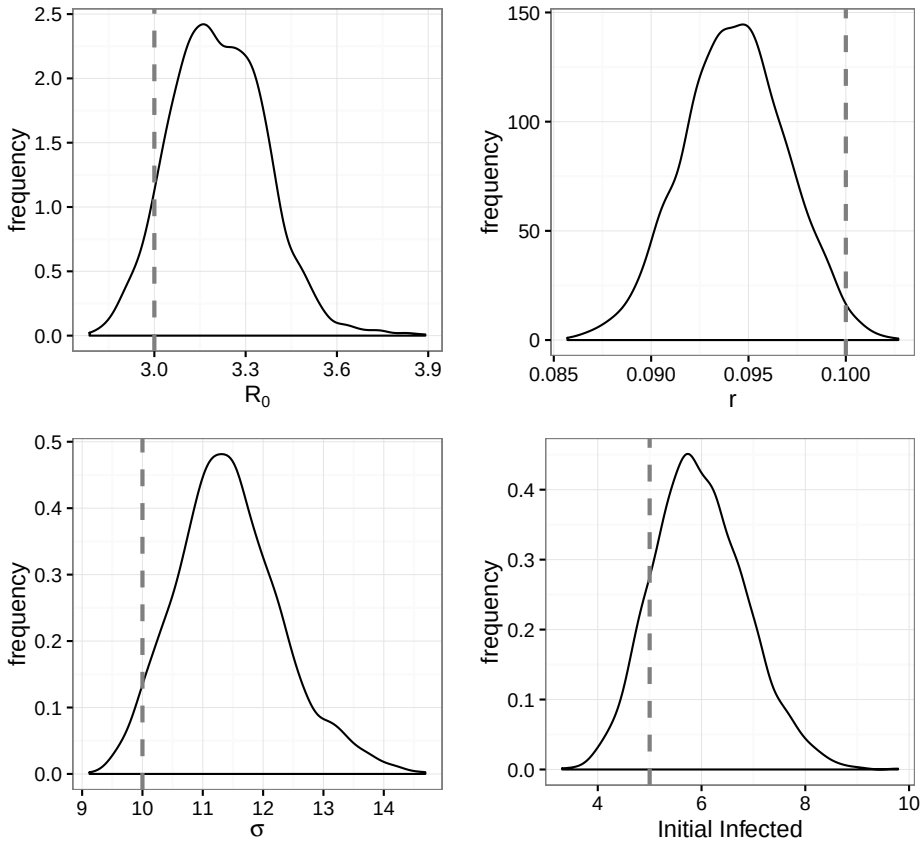

Figure 2.5: Kernel density estimates produced by Stan. Dashed lines show true parameter values.

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

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

And the observation process by

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

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

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

3.2 Algorithm

Now we will formalize the particle filter.

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

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

Algorithm 3: SIR particle filter

/* Select a starting point */

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

/* Setup */

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

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

  /* Evolve */

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

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

  /* Weight */

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

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

  /* Normalize */

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

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

  /* Resample */

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

/* Samples from approximated posterior distribution */

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

3.3 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 [22][23] 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 [22], Iterated Filtering 2 [23] 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 */

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

/* Particle seeding distribution */

$\Theta \leftarrow P_0$

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

  /* Pass perturbation */

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

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

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

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

      /* Iteration perturbation */

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

      /* Evolve */

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

      /* Weight */

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

    /* Normalize */

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

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

    /* Resample */

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

  /* Collect particles for next pass */

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

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

/* Samples from approximated posterior distribution */

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

3.6 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 $\mathcal{R}_0 = 3.0, \gamma = 0.1, N = 500$, 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.

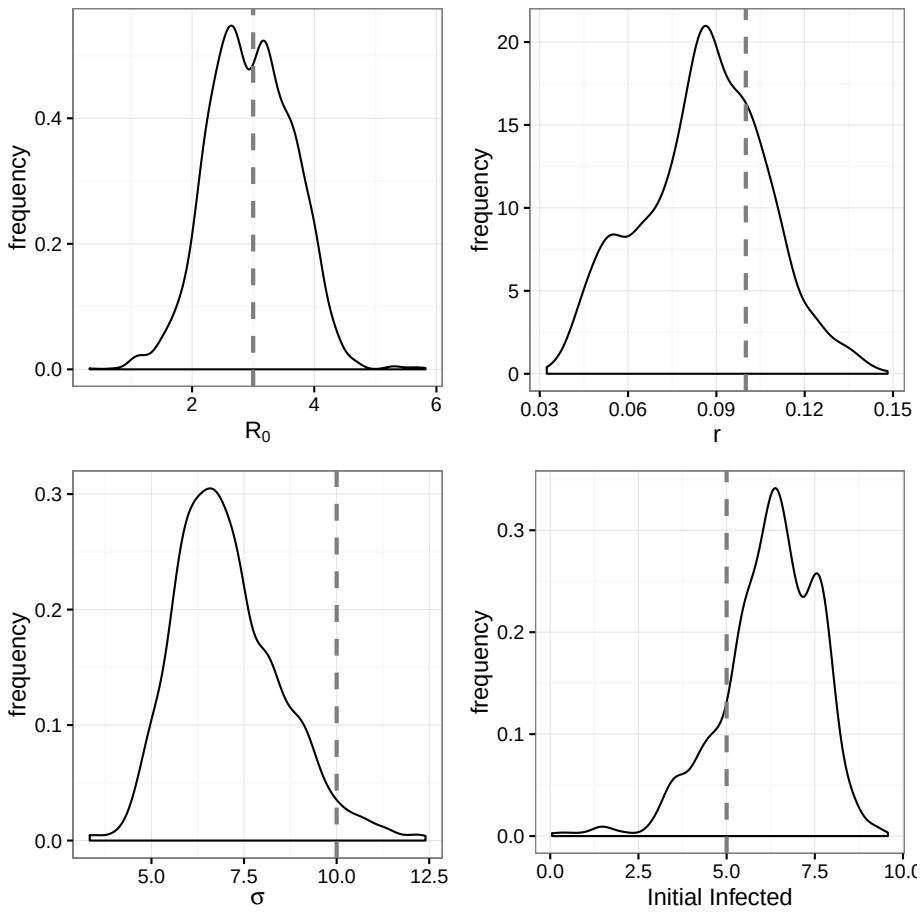

Figure 3.1: Kernel estimates for four essential system parameters. True values are indicated by dashed lines.

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 $\sigma = \sigma_E^2/(1-\rho^2)$.

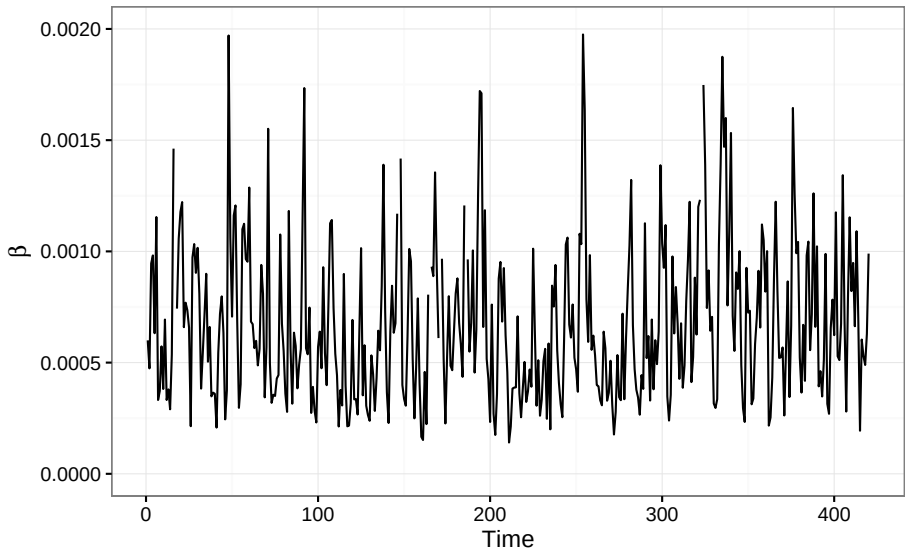

Figure 4.1: Simulated geometric autoregressive process shown in Equation [4.1].

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

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

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

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

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

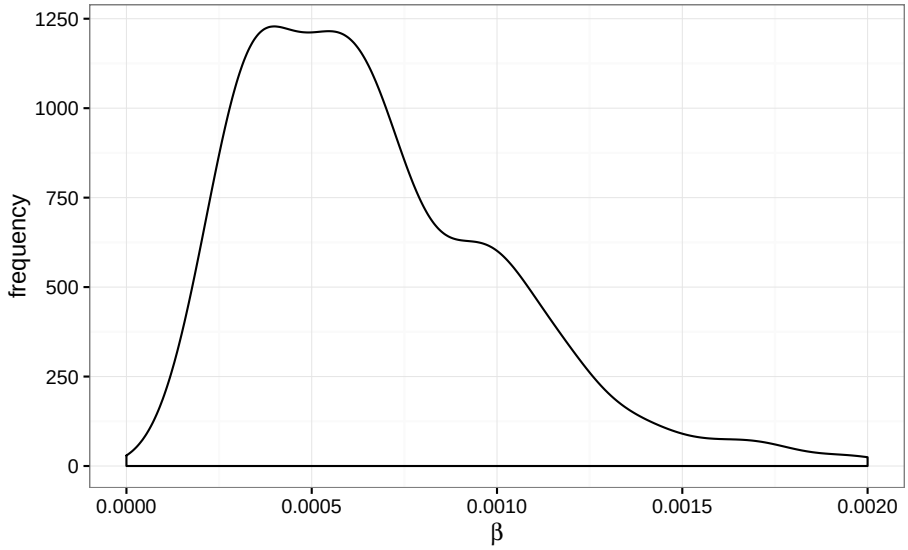

Figure 4.2: Density plot of values shown if Figure[4.1].

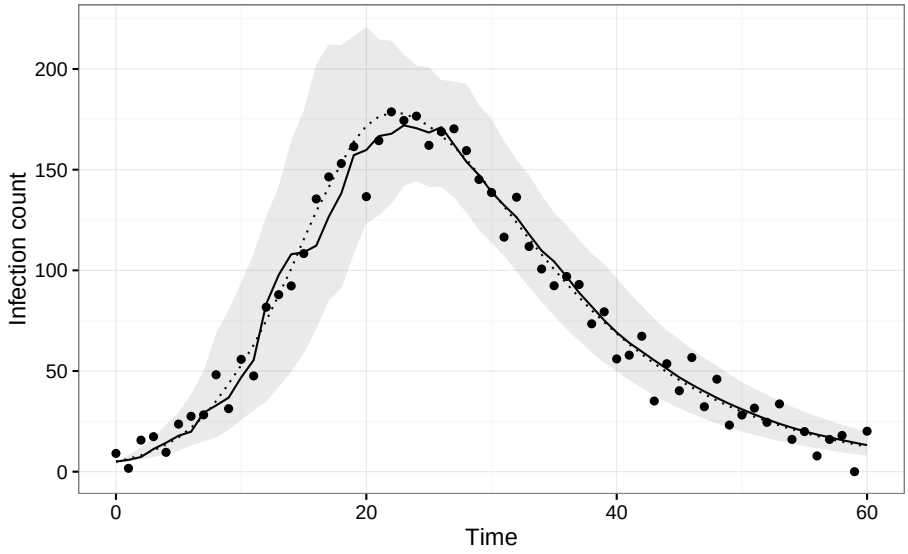

Figure 4.3: Stochastic SIR model simulated using an explicit Euler stepping scheme. The solid line is a single random trajectory, the dots show the data points obtained by adding in observation error defined as $\epsilon_{\text{obvs}} = \mathcal{N}(0, 10)$, and the grey ribbon is centre 95th quantile 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) [17].

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

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

where P is the number of particles.

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

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

4.3 IF2 Fitting

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

The code used here is a mix of R and C++ implemented using Rcpp. The fitting was undertaken using 2500 particles with 50 IF2 passes and a cooling schedule given by a reduction in particle spread determined by 0.975^p, where p is the pass number starting with 0. This geometric cooling scheme is standard for use with IF2 [24][26], with the cooling rate chosen to neatly scale the perturbation factor from 1 to 0.02 (almost 0) over 50 passes.

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

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

		IF2		HMC	
Name	True	Fit	Error	Fit	Error
$\mathcal{R}_0$	3.0	3.27	9.08×10^{-2}	3.12	1.05×10^{-1}
γ	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.

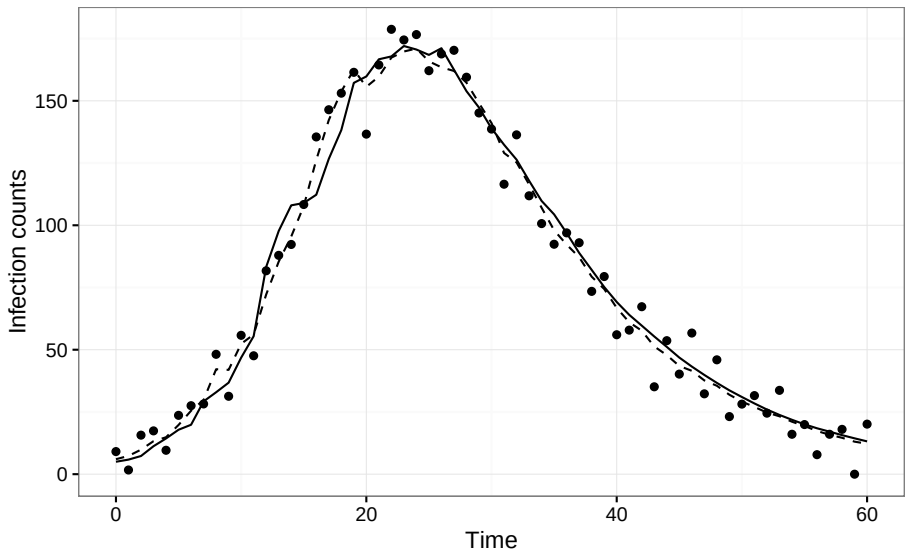

Figure 4.5: True system trajectory (solid line), observed data (dots), and IF2 estimated real state (dashed line).

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.

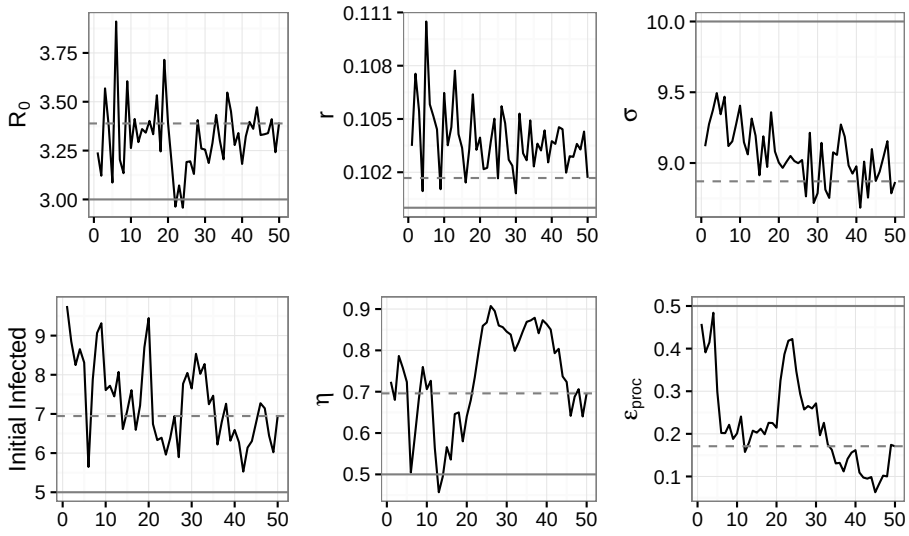

Figure 4.6: The horizontal axis shows the IF2 pass number. The solid black lines show the evolution of the ML estimates, the solid grey lines show the true value, and the dashed grey lines show the mean parameter estimates from the particle swarm after the final pass.

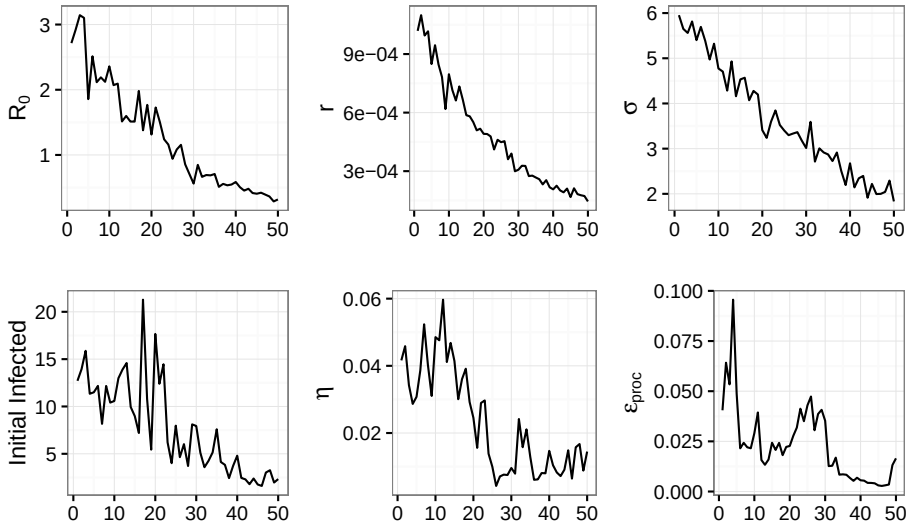

Figure 4.7: The horizontal axis shows the IF2 pass number and the solid black lines show the evolution of the standard deviations of the particle swarm values.

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.

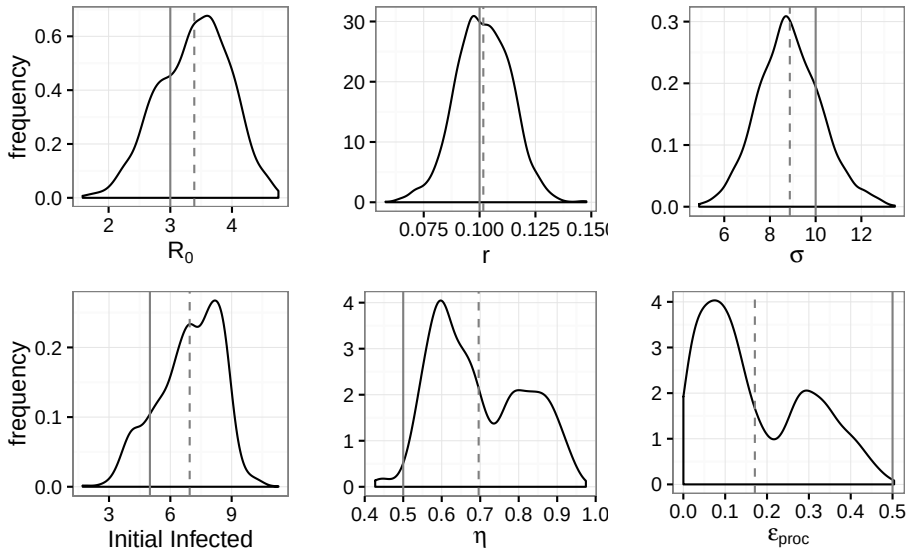

Figure 4.8: The solid grey lines show the true parameter values and the dashed grey lines show the density medians.

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.

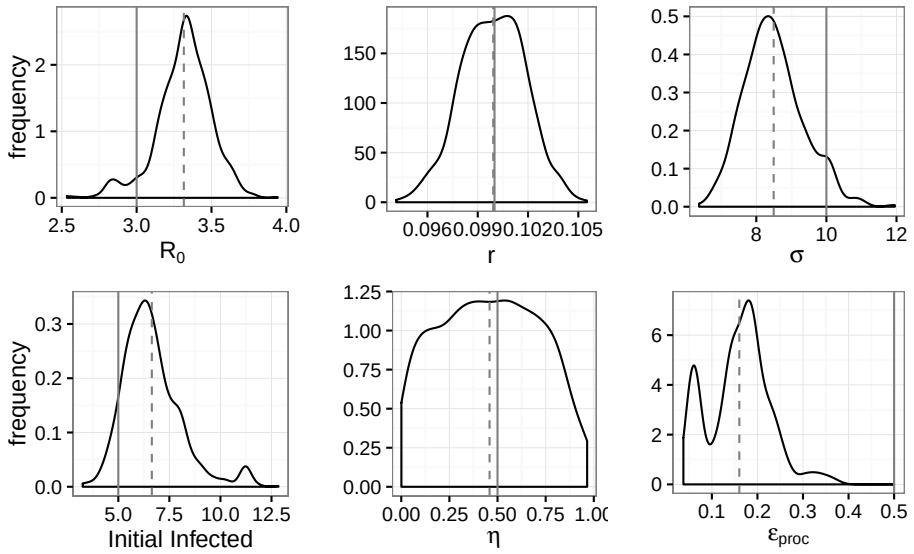

Figure 4.9: As before, the solid grey lines show the true parameter values and the dashed grey lines show the density medians.

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.

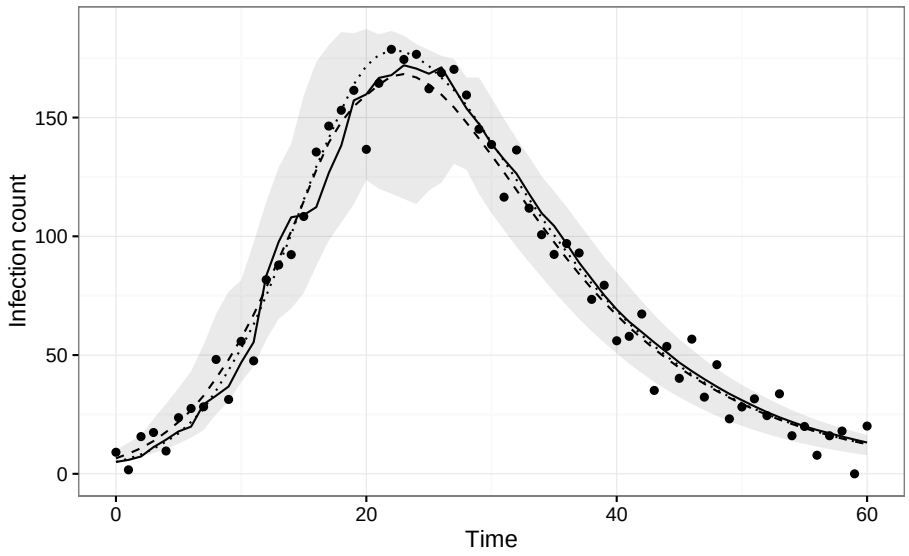

Figure 4.10: Result from 100 HMC bootstrap trajectories. The solid line shows the true states, the dots show the data, the dotted line shows the average system behaviour, the dashed line shows the bootstrap mean, and the grey ribbon shows the centre 95th quantile of the bootstrap trajectories.

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.

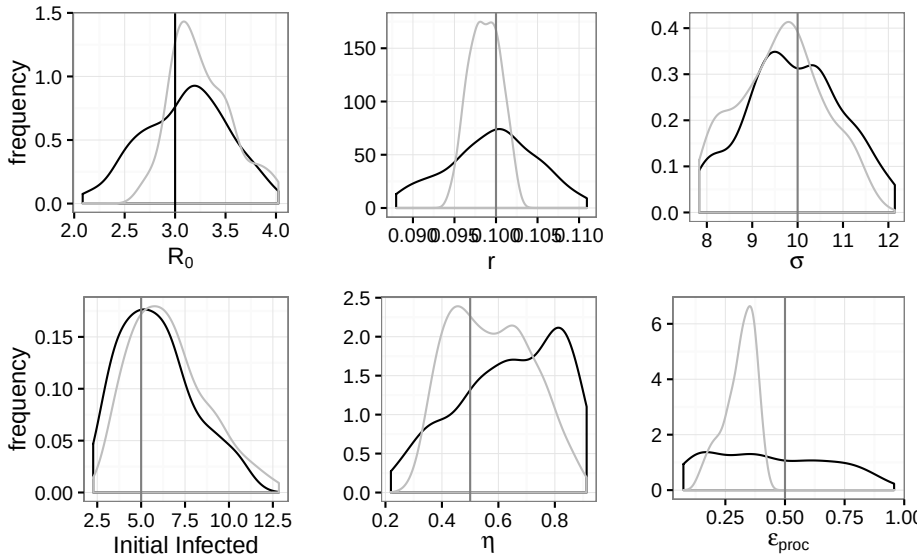

Figure 4.11: IF2 point estimate densities are shown in black and HMC point estimate densities are shown in grey. The vertical lines show the true parameter values.

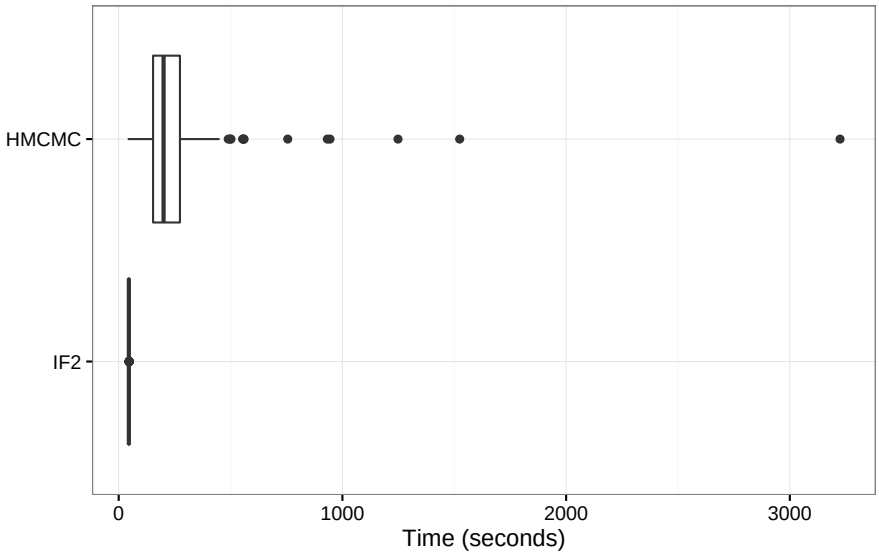

Figure 4.12: Fitting times for IF2 and HMC, in seconds. The centre box in each plot shows the centre 50th percent, with the bold centre line showing the median.

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.

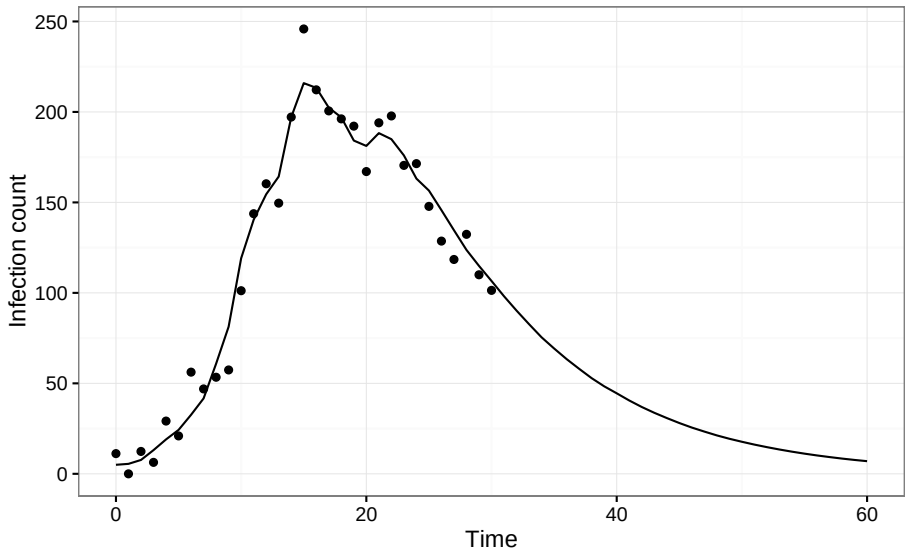

Figure 5.1: Infection count data truncated at $T = 30$. The solid line shows the true underlying system states, and the dots show those states with added observation noise. Parameters used were $\mathcal{R}_0 = 3.0$, $\gamma = 0.1$, $\eta = .05$, $\sigma_{proc} = 0.5$, and additive observation noise 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.

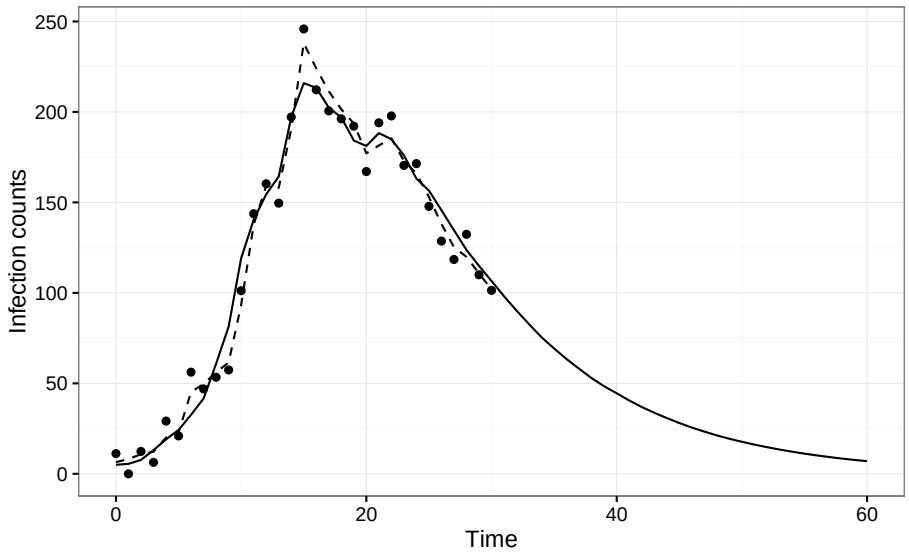

Figure 5.2: Infection count data truncated at $T = 30$ from Figure [5.1]. The dashed line shows IF2's attempt to reconstruct the true underlying state from the observed data points.

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 [14].

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 */

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

/* Generate artificial data sets */

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

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

/* Fit to new data sets */

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

  $\theta_i \leftarrow IF2(D_i)$

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

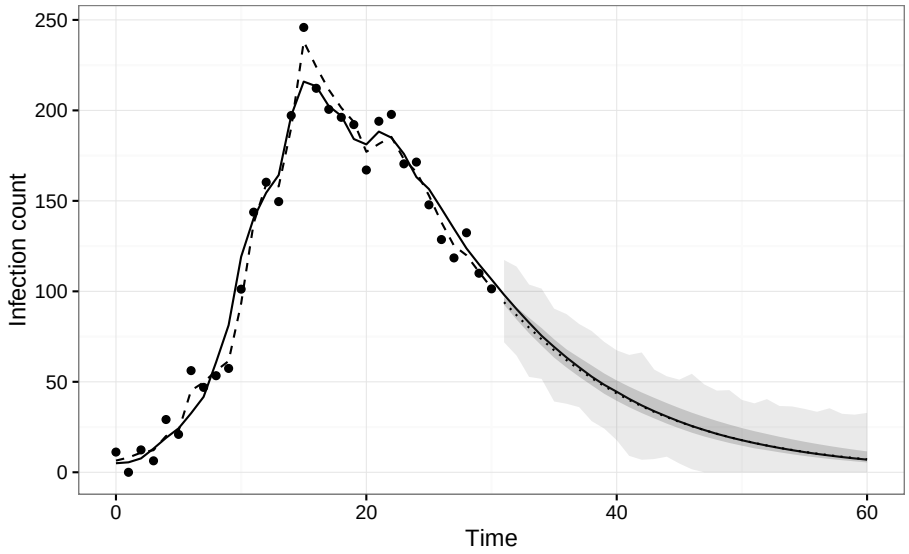

Figure 5.3: Forecast produced by the IF2 / parametric bootstrapping framework. The dotted line shows the mean estimate of the forecasts, the dark grey ribbon shows the 95% confidence interval based on the 0.025 and 0.975 quantiles on the true state estimates, and the lighter grey ribbon shows the same 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 $SSE = 608$.

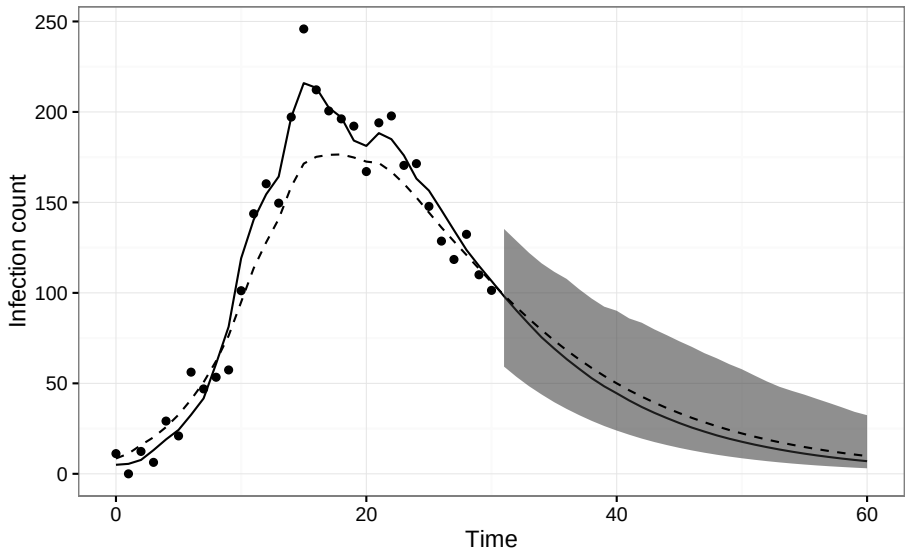

Figure 5.4: Forecast produced by the HMC / bootstrapping framework with $M = 200$ trajectories. The dotted line shows the mean estimate of the forecasts, and the grey ribbon shows the 95% confidence interval on the estimated true states as described in Figure [5.3].

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

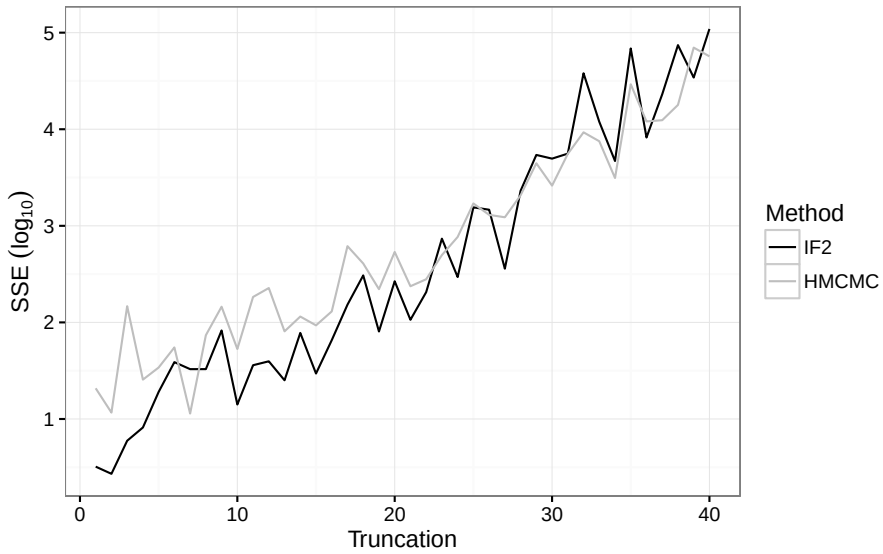

Figure 5.5: Error growth as a function of data truncation amount. Both methods used 200 bootstrap trajectories. Note that the y-axis shows the natural log of the averaged SSE, not the total SSE.

used here has this capability, but it was not utilised in the comparison so as to accurately 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 [38][39][21][15] 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 $\{\mathbf{x_i}\}$ */

for $i = E : T$ **do**
  $\mathbf{x_i} = (x_i, x_{i-1}, ..., x_{i-E-1})$

/* Construct mapping from library vectors to predictions */

for $i = 1 : (T_E + 1)$ **do**
  for $j = 1 : E$ **do**
    $A(i,j) = w(||\mathbf{x_i} - \mathbf{x_t}||)\mathbf{x_i}(j)$

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

/* Use SVD to solve the mapping system, Ac = b */

$SVD(Ac = b)$

/* Compute forecast */

$\hat{y}_t = \sum_{j=0}^{E} c_t(j)\mathbf{x_t}(j)$

/* Forecasted value in time series */

Output: Forecast $\hat{y}_t$

6.3 SIRS Model

In an epidemic or infectious disease context, the S-map algorithm will only really work on time series that appear cyclic. While there is nothing mechanically that prevents it from operating on a time series that do not appear cyclic, S-mapping requires a long time series in order to build a quality library. Without one the forecasting process would produce unreliable data.

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.

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

There are two new features here. We have a rate or waning immunity α through which people become able to be reinfected, and a seasonality factor function $\Gamma(t)$ defined as

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

where t is in days. 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 $\mathcal{N}(0, \sigma)$.

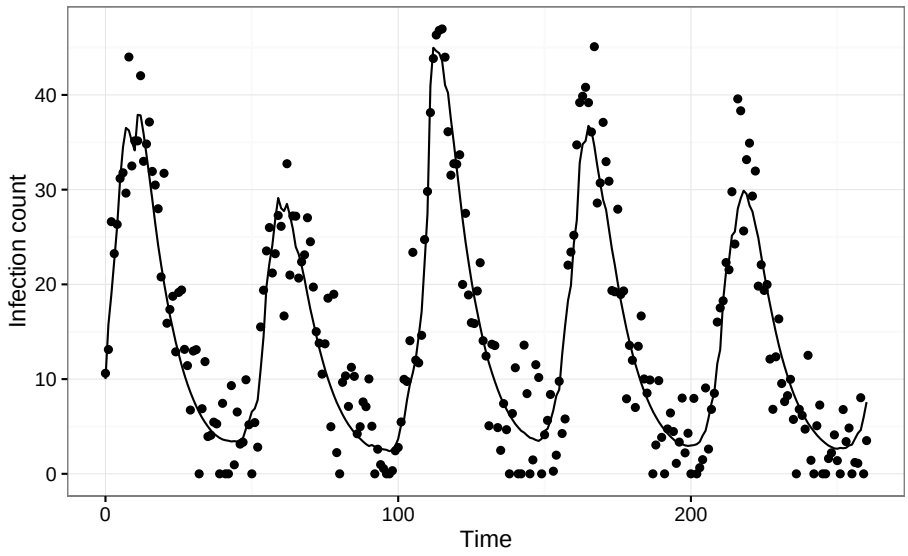

Figure 6.1: Five cycles generated by the SIRS function. The solid line the the true number of cases, dots show case counts with added observation noise. The parameter values were $\mathcal{R}_0 = 3.0$, $\gamma = 0.1$, $\eta = 1$, $\sigma = 5$, and 10 initial cases.

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

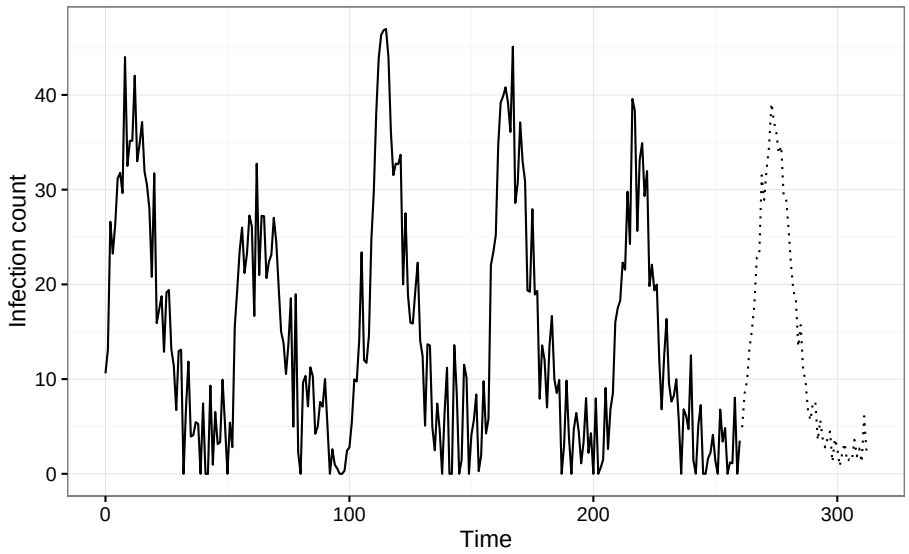

Figure 6.2: S-map applied to the data from the previous figure. The solid line shows the infection counts with observation noise from the previous plot, and the dotted line is the S-map forecast. Parameters chosen were $E = 14$ and $\theta = 3$.

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 [15]. 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

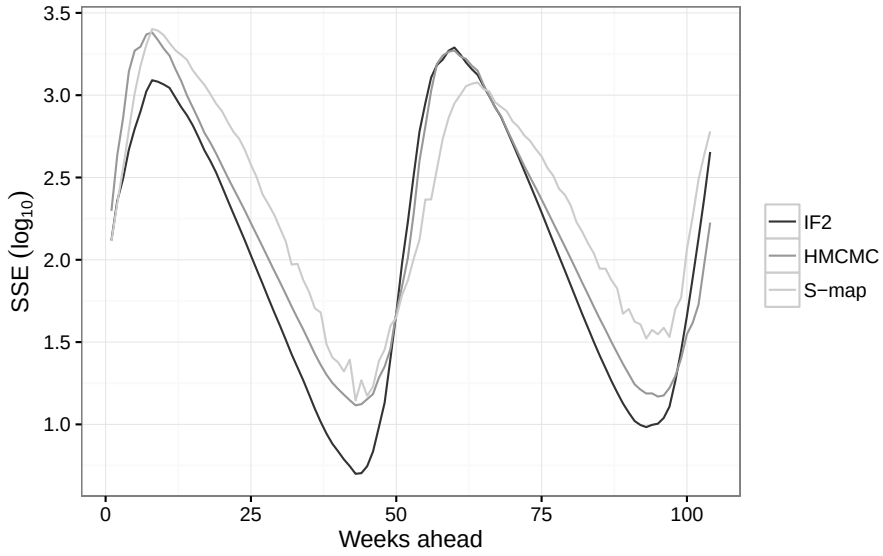

Figure 6.3: Error as a function of forecast length.

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

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

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

Additionally, we are interested in coverage. While a full coverage analysis would require roughly a 100-fold increase in computational complexity, we can use the trajectories generated by IF2 and HMC to display forecast coverage across data sets, given particular weeks in the forecast.

Figure [7.5] shows such plots for forecasts 10 and 45 weeks ahead. Week 10 represents the approximate first epidemic "spike" we are attempting to predict, a typically difficult task, and week 45 represents the first "trough", which should be easier to predict. We can see that the error bars are much wider when attempting to predict the spike, and much smaller for the trough. Also the trough coverage appears slightly better than the spike coverage.

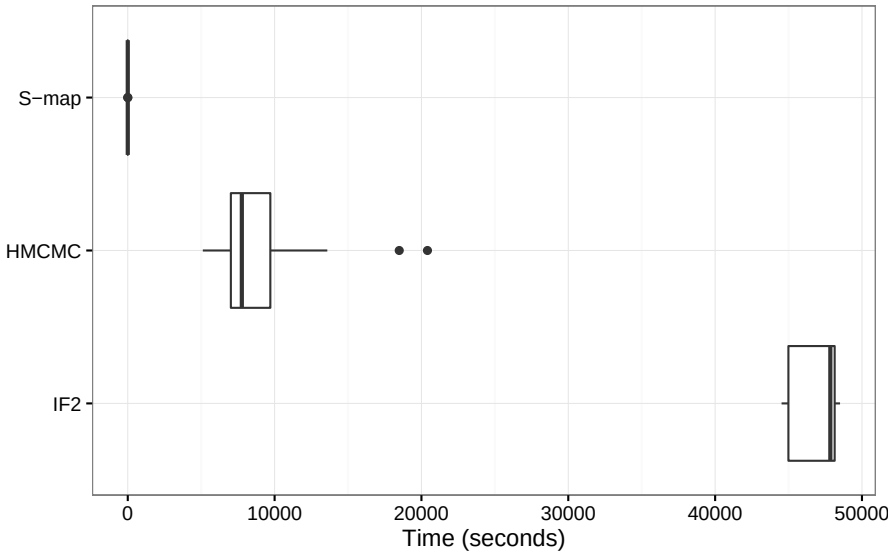

Figure 6.4: Runtimes for producing SIRS forecasts. The box shows the middle 50th percent, the bold line is the median, and the dots are outliers. Note that these are not "true" outliers, simply ones outside a threshold based on the interquartile range.

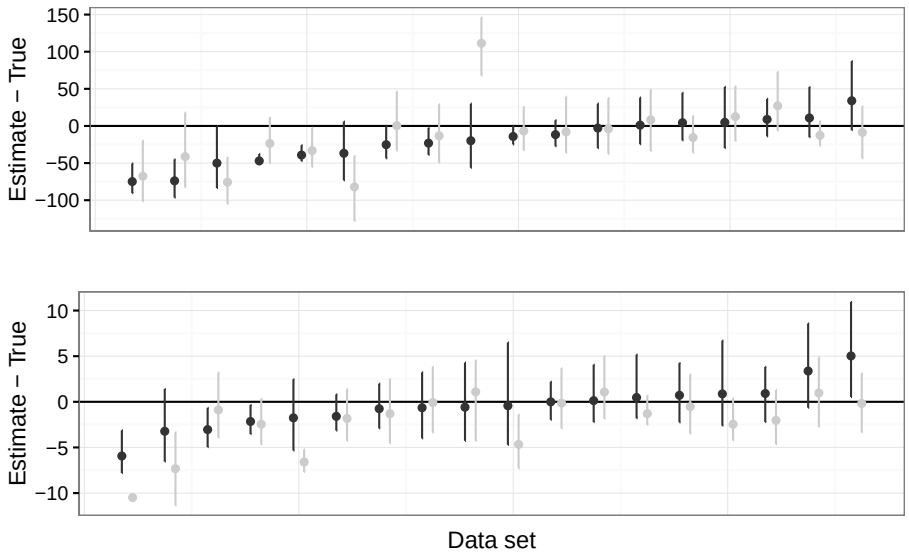

Figure 6.5: Coverage plots for forecast weeks 10 (top) and 45 (bottom). Black bars are from IF2 forecast trajectories, and grey bars are from HMC trajectories.

Chapter 7

Spatial Epidemics

7.1 Spatial SIR

Spatial epidemic models provide a way to capture not just the temporal trend in an epidemic, but to also integrate spatial data and infer how the infection is spreading in both space and time. One such model we can use is a dynamic spatiotemporal SIR model.

We wish to construct a model build upon the stochastic SIR compartment model described previously but one that consists of several connected spatial locations, each with its own set of compartments. Consider a set of locations numbered $i = 1, ..., N$, where N is the number of locations. Further, let N_i be the number of neighbours location i has. The model is then

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

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

As before we let β embark on a geometric random walk defined as

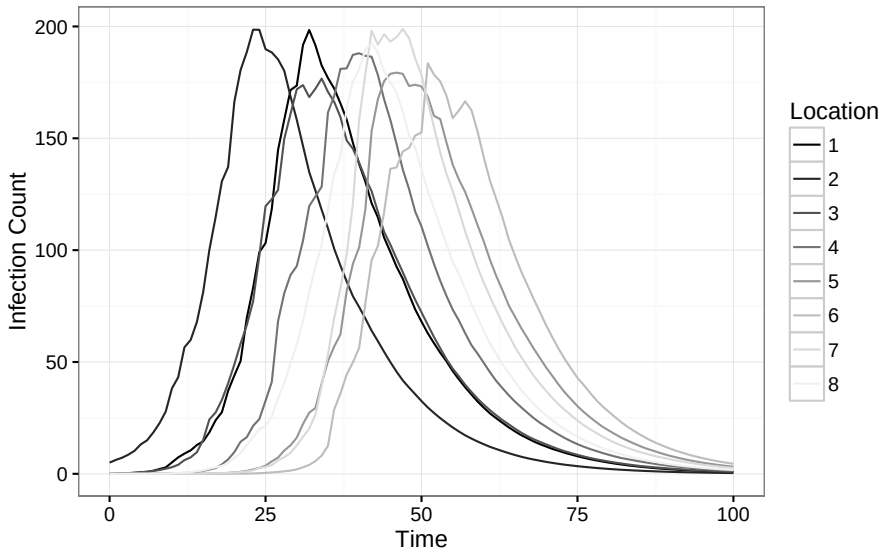

Figure 7.1: Evolution of a spatial epidemic in a ring topology. The outbreak was started with 5 cases in Location 2. Parameters were $\mathcal{R}_0 = 3.0$, $\gamma = 0.1$, $\eta = 0.5$, $\sigma_{err} = 0.5$, and $\phi = 0.5$.

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

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].

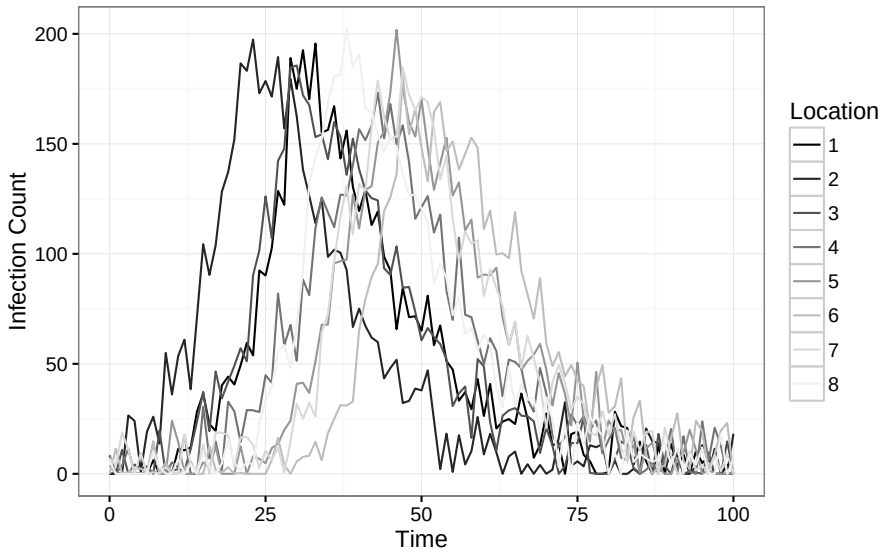

Figure 7.2: Evolution of a spatial epidemic as in Figure [7.1], with added observation noise drawn from $\mathcal{N}(0, 10)$.

7.2 Dewdrop Regression

Dewdrop regression [21] 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.

As before, we are interested in coverage. Again, a full coverage analysis would require roughly a 100-fold increase in computational complexity, but we can use the trajectories generated by IF2 and HMC to display forecast coverage across data sets, given particular weeks in the forecast.

Figure [7.5] shows such plots for forecasts 2 and 10 weeks ahead in location 8. Location 8 was used as it lands in the middle of the cohort of locations in terms of outbreak progression. We can see that the error bars are much wider when attempting to

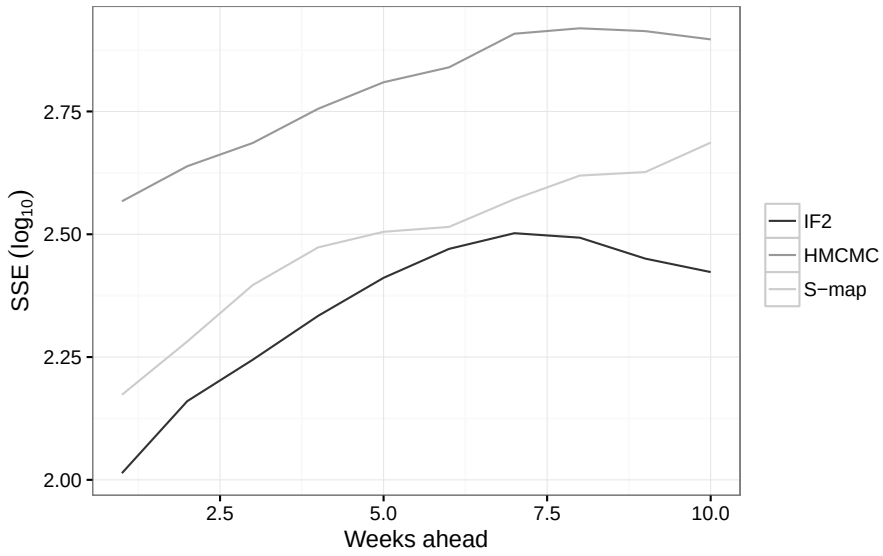

Figure 7.3: Average SSE (log scale) across each location and all trials as a function of the number of weeks ahead in the forecast.

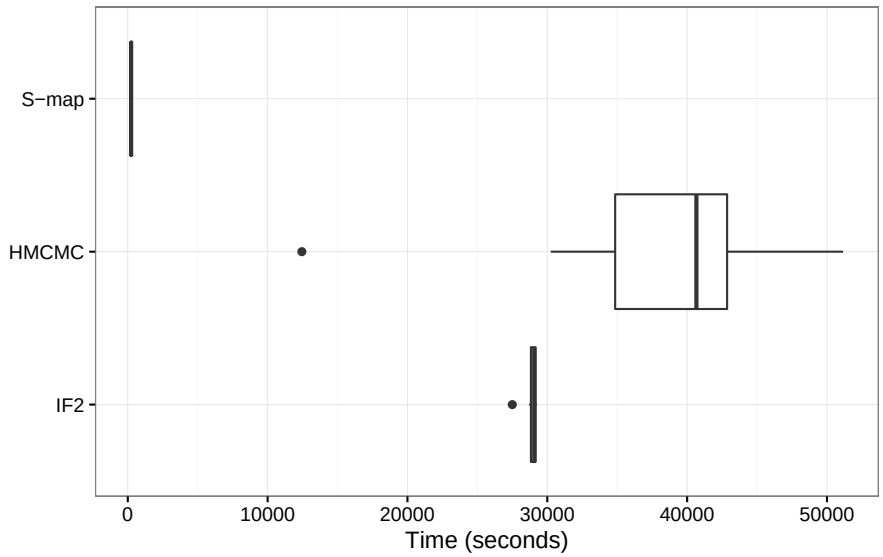

Figure 7.4: Runtimes for producing spatial SIR forecasts. The box shows the middle 50th percent, the bold line is the median, and the dots are outliers.

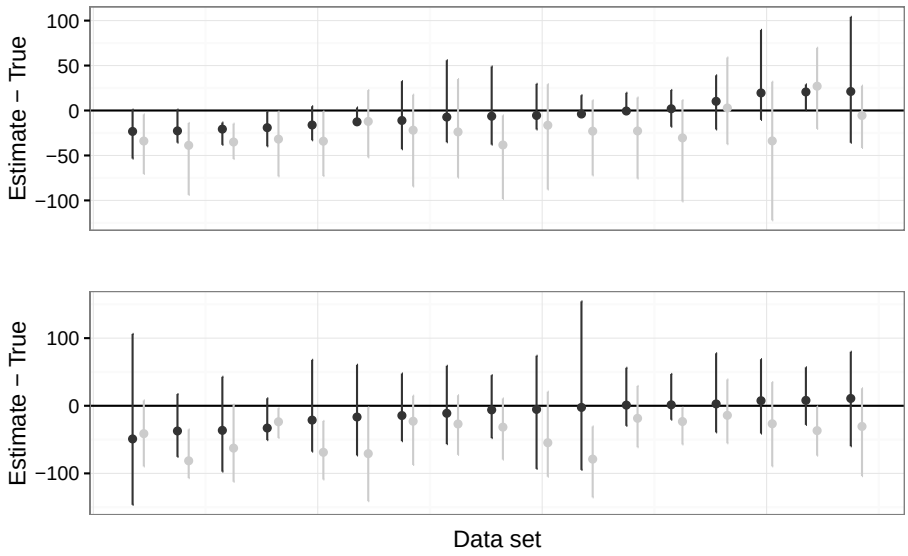

Figure 7.5: Coverage plots for forecast weeks 2 (top) and 10 (bottom) in location 8. Black bars are from IF2 forecast trajectories, and grey bars are from HMC trajectories.

predict further into the future. HMC is consistently underestimates the intensity of the epidemic in both forecast lengths, but produces smaller error bars for the longer forecast.

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 [41].

Recently, transistor count growth has begun 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 consist of 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 consist 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 method 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 $\mathcal{O}(\log(n))$ [18]. 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 were 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 [25] 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 represents only 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.

[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] Jonathan Dushoff et al. "Dynamical resonance can account for seasonality of influenza epidemics." In: *Proceedings of the National Academy of Sciences of the United States of America* 101.48 (2004), pp. 16915–16916. ISSN: 0027-8424. DOI: 10.1073/pnas.0407293101. URL: http://www.pnas.org/content/101/48/16915.full.

[13] 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/.

[14] 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.

[15] 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.

[16] 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.

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

[18] Mark Harris. *Optimizing parallel reduction in CUDA*. Sept. 2007. URL: http://docs.nvidia.com/cuda/samples/6_Advanced/reduction/doc/reduction.pdf.

[19] 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.

[20] 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/.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

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

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

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

[35] 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.

[36] 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.

[37] Stan Development Team. "Stan Modeling Language User's Guide and Reference Manual". In: 2.9.0 (2015).

[38] 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.

[39] 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.

[40] 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.

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

[42] 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.

[43] 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.

Appendix A

Hamiltonian MCMC

A.1 Full R code

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

```
## 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)))

    })

}
```

```
1
2  pars  <- c(R0  <- 3.0,    # average number of new infected individuals
3      per infectious person
4             r   <- 0.1,    # recovery rate
5             N   <- 500)    # population size
6
7  T <- 100
8  y_ini <- c(S = 495, I = 5, R = 0)
9  times <- seq(0, T, by = 1)
10
11 odeout <- ode(y_ini, times, SIR, pars)
12
13 set.seed(1001)
14 sigma <- 10
15 infec_counts_raw <- odeout[,3] + rnorm(T+1, 0, sigma)
16 infec_counts     <- ifelse(infec_counts_raw < 0, 0, infec_counts_raw)
17
18 g <- qplot(0:T, odeout[,3], geom = "line", xlab = "Time (weeks)", ylab
19      = "Infection Count") +
20     geom_point(aes(y = infec_counts)) +
21     theme_bw()
22
23 print(g)
24 ggsave(g, filename="dataplot.pdf", height=4, width=6.5)
25
26 sPw <- 7
27 datlen <- (T-1)*7 + 1
28
29 data <- matrix(data = -1, nrow = T+1, ncol = sPw)
30 data[,1] <- infec_counts
31 standata <- as.vector(t(data))[1:datlen]
32
33 sir_data <- list( T = datlen,   # simulation time
34                   y = standata, # infection count data
35                   N = 500,      # population size
36                   h = 1/sPw )   # step size per day
37
38 rstan_options(auto_write = TRUE)
39 options(mc.cores = parallel::detectCores())
40 stan_options <- list(   chains = 4,    # number of chains
41                         iter   = 2000, # iterations per chain
42                         warmup = 1000, # warmup interations
43                         thin   = 2)   # thinning number
44 fit <- stan(file    = "d_sirode_euler.stan",
45             data    = sir_data,
46             chains  = stan_options$chains,
47             iter    = stan_options$iter,
48             warmup  = stan_options$warmup,
49             thin    = stan_options$thin )
50
51 exfit <- extract(fit, permuted = TRUE, inc_warmup = FALSE)
```

```
1  79
2  80 R0points ← exfit$R0
3  81 R0kernel ← qplot(R0points, geom = "density", xlab = expression(R[0]),
4         ylab = "frequency") +
5  82     geom_vline(aes(xintercept=R0), linetype="dashed", size=1,
6            color="grey50") +
7  83     theme_bw()
8  84
9  85 print(R0kernel)
10 86 ggsave(R0kernel, filename="kernelR0.pdf", height=3, width=3.25)
11 87
12 88 rpoints ← exfit$r
13 89 rkernel ← qplot(rpoints, geom = "density", xlab = "r", ylab = "
14       frequency") +
15 90     geom_vline(aes(xintercept=r), linetype="dashed", size=1,
16            color="grey50") +
17 91     theme_bw()
18 92
19 93 print(rkernel)
20 94 ggsave(rkernel, filename="kernelr.pdf", height=3, width=3.25)
21 95
22 96 sigmapoints ← exfit$sigma
23 97 sigmakernel ← qplot(sigmapoints, geom = "density", xlab = expression(
24       sigma), ylab = "frequency") +
25 98     geom_vline(aes(xintercept=sigma), linetype="dashed", size=1,
26            color="grey50") +
27 99     theme_bw()
28 100
29 101 print(sigmakernel)
30 102 ggsave(sigmakernel, filename="kernelsigma.pdf", height=3, width=3.25)
31 103
32 104 infecpoints ← exfit$y0[,2]
33 105 infeckernel ← qplot(infecpoints, geom = "density", xlab = "Initial
34       Infected", ylab = "frequency") +
35 106     geom_vline(aes(xintercept=y_ini[['I']]), linetype="dashed",
36            size=1, color="grey50") +
37 107     theme_bw()
38 108
39 109 print(infeckernel)
40 110 ggsave(infeckernel, filename="kernelinfec.pdf", height=3, width=3.25)
41 111
42 112 exfit ← extract(fit, permuted = FALSE, inc_warmup = FALSE)
43 113 plotdata ← melt(exfit[,,"R0"])
44 114 tracefitR0 ← ggplot() +
45 115               geom_line(data = plotdata,
46 116                         aes(x = iterations,
47 117                         y = value,
48 118                         color = factor(chains, labels = 1:stan_
49                                options$chains))) +
50 119               labs(x = "Sample", y = expression(R[0]), color = "Chain
51                     ") +
```

```
            scale_color_brewer(palette="Greys") +
            theme_bw()

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

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

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

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

A.2 Full Stan code

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

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

data {

    int     <lower=1>   T;      // total integration steps
    real                y[T];   // observed number of cases
    int     <lower=1>   N;      // population size
    real                h;      // step size

}

parameters {

    real <lower=0, upper=10>    R0;     // R0
    real <lower=0, upper=10>    r;      // recovery rate
```

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

}

model {

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

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

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

    for (t in 2:T) {

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

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

    }

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

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

}
```

Appendix B

Iterated Filtering

B.1 Full R code

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

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

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

SIR ← function(Time, State, Pars) {

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

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

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

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

    })

}

```

```
T     ← 100
N     ← 500
sigma   ← 10
i_infec ← 5

## Generate true trajecory and synthetic data
##

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

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

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

set.seed(1001)

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

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

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

## Rcpp stuff
##

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

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

## Parameter density kernels
##

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

```

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

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

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

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

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

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

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

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

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

B.2 Full C++ code

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

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

```
    McMaster University
    2016

    */

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

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

#include <Rcpp.h>
using namespace Rcpp;

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

struct ParticleInfo {
    double R0mean;      double R0sd;
    double rmean;       double rsd;
    double sigmamean;   double sigmasd;
    double Sinitmean;   double Sinitsd;
    double Iinitmean;   double Iinitsd;
    double Rinitmean;   double Rinitsd;
};


int timeval_subtract (double *result, struct timeval *x, struct
    timeval *y);
int check_double(double x,double y);
void exp_euler_SIR(double h, double t0, double tn, int N, Particle *
```

```
    particle);
void copyParticle(Particle * dst, Particle * src);
void perturbParticles(Particle * particles, int N, int NP, int
    passnum, double coolrate);
bool isCollapsed(Particle * particles, int NP);
void particleDiagnostics(ParticleInfo * partInfo, Particle *
    particles, int NP);
NumericMatrix if2(NumericVector * data, int T, int N);
double randu();
double randn();

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

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

    int     i_infec     = I0;

    NumericMatrix paramdata(NP, 6);

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

    double w[NP];           // particle weights

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

    printf("Initializing particle states\n");

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

        double R0can, rcan, sigmacan, Iinitcan;

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

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

        do {
            sigmacan = merr + merr*randn();
        } while (sigmacan < 0);
```

```
1   99          particles[n].sigma = sigmacan;
2  100
3  101          do {
4  102              Iinitcan = i_infec + i_infec*randn();
5  103          } while (Iinitcan < 0 || N < Iinitcan);
6  104          particles[n].Sinit = N - Iinitcan;
7  105          particles[n].Iinit = Iinitcan;
8  106          particles[n].Rinit = 0.0;
9  107
10 108      }
11 109
12 110      // START PASSES THROUGH DATA
13 111
14 112      printf("Starting filter\n");
15 113      printf("---------------\n");
16 114      printf("Pass\n");
17 115
18 116
19 117      for (int pass = 0; pass < nPasses; pass++) {
20 118
21 119          printf("...%d / %d\n", pass, nPasses);
22 120
23 121          perturbParticles(particles, N, NP, pass, coolrate);
24 122
25 123          // initialize particle system states
26 124          for (int n = 0; n < NP; n++) {
27 125
28 126              particles[n].S = particles[n].Sinit;
29 127              particles[n].I = particles[n].Iinit;
30 128              particles[n].R = particles[n].Rinit;
31 129
32 130          }
33 131
34 132          // between-pass perturbations
35 133
36 134          for (int t = 1; t < T; t++) {
37 135
38 136              // between-iteration perturbations
39 137              perturbParticles(particles, N, NP, pass, coolrate);
40 138
41 139              // generate individual predictions and weight
42 140              for (int n = 0; n < NP; n++) {
43 141
44 142                  exp_euler_SIR(1.0/10.0, 0.0, 1.0, N, &particles[n]);
45 143
46 144                  double merr_par = particles[n].sigma;
47 145                  double y_diff   = data[t] - particles[n].I;
48 146
49 147                  w[n] = 1.0/(merr_par*sqrt(2.0*M_PI)) * exp( - y_diff*
50                         y_diff / (2.0*merr_par*merr_par) );
51 148
```

```
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 }
```

```
248
249
250 /*  Particle pertubation function to be run between iterations and
        passes
251
252     */
253 void perturbParticles(Particle * particles, int N, int NP, int
        passnum, double coolrate) {
254
255     double coolcoef = pow(coolrate, passnum);
256
257     double spreadR0     = coolcoef * R0true / 10.0;
258     double spreadr      = coolcoef * rtrue  / 10.0;
259     double spreadsigma  = coolcoef * merr   / 10.0;
260     double spreadIinit  = coolcoef * I0     / 10.0;
261
262     double R0can, rcan, sigmacan, Iinitcan;
263
264     for (int n = 0; n < NP; n++) {
265
266         do {
267             R0can = particles[n].R0 + spreadR0*randn();
268         } while (R0can < 0);
269         particles[n].R0 = R0can;
270
271         do {
272             rcan = particles[n].r + spreadr*randn();
273         } while (rcan < 0);
274         particles[n].r = rcan;
275
276         do {
277             sigmacan = particles[n].sigma + spreadsigma*randn();
278         } while (sigmacan < 0);
279         particles[n].sigma = sigmacan;
280
281         do {
282             Iinitcan = particles[n].Iinit + spreadIinit*randn();
283         } while (Iinitcan < 0 || Iinitcan > 500);
284         particles[n].Iinit = Iinitcan;
285         particles[n].Sinit = N - Iinitcan;
286
287     }
288
289 }
290
291
292 /*  Convinience function for particle resampling process
293
294     */
295 void copyParticle(Particle * dst, Particle * src) {
296
```

```
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 }
308
309
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.
313
314     */
315 bool isCollapsed(Particle * particles, int NP) {
316
317     bool retVal;
318
319     double R0mean = 0, rmean = 0, sigmamean = 0, Sinitmean = 0,
        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,
         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.

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

StocSIR ← 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']]

    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) ) {

```

```
        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.

```
# Dexter Barrows
# dbarrows.github.io
# McMaster University
# 2016
#
# IF2 parametric bootstrapping function

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

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

  source(stoc_sir_file)

  if (nTrials < 2)
    ntrials ← 2

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

```
statedata_parent ← data.frame( if2data_parent$statedata )
names(statedata_parent) ← c("S","I","R","B")
statemeans_parent ← colMeans(statedata_parent)
names(statemeans_parent) ← c("S","I","R","B")


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

  source(stoc_sir_file)

  ## draw new data
  ##

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

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

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

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

    ## refit using new data
    ##

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

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

```

    ## generate the rest of the trajectory
    ##

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

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

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


  }

  return(trajectories)

}
```

D.2 RStan Forward Simulator

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

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

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

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

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

```
  berr ← pars[['berr']]

  S ← y[['S']]
  I ← y[['I']]
  R ← y[['R']]

  B0 ← R0 * r / N
  B ← B0

  out[1,] ← c(S,I,R,B)

  h ← 1 / steps

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

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

    BSI ← B*S*I
    rI ← r*I

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

    S ← S + h*dS
    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) ) {

```

```
        #Bfac ← 1/2 - cos((2*pi/365)*i)/2
        Bfac ← exp(2*cos((2*pi/365)*i) - 2)

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

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

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

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

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

  }

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

}

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

E.2 SIRS HMC R Function Code

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

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

```
## McMaster University
## 2016

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

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

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

  S ← y[['S']]
  I ← y[['I']]
  R ← y[['R']]

  B0 ← R0 * r / N
  B ← B0

  out[1,] ← c(S,I,R,B)

  h ← 1 / steps

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

    Bfac ← exp(2*cos((2*pi/365)*i) - 2)

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

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

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

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

```

```
  }

  return(out)

}

### suggested parameters
#
# T     <- 200
# i_infec   <- 5
# steps   <- 7
# N     <- 500
# sigma   <- 5
#
# pars <- c(R0 = 3.0,  # new infected people per infected person
#           r = 0.1,  # recovery rate
#           gam = 2,  # new infected shock intensity
#       N = 500,    # population size
#       eta = 0.5,  # geometric random walk
#       berr = 0.5, # Beta geometric walk noise
#           re = 2)   # resuceptibility rate
```

E.3 SMAP Code

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

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

library(pracma) # needed for tiling function

smap <- function(data, E, theta, stepsAhead) {

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

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

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

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

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

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

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

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

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

      ssemean ← ssemean + (sse / nTrials)

    }

    ssemat[i,j] ← ssemean


  }
}

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

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

Emin ← Elist[mininds['row']]
thetamin ← thetalist[mininds['col']]

print(Emin)
print(thetamin)
```

E.5 RStan SIRS Code

This code implements a periodic SIRS model in Rstan.

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

```
## McMaster University
## 2016

data {

    int     <lower=1>   T;      // total integration steps
    real                y[T];   // observed number of cases
    int     <lower=1>   N;      // population size
    real                h;      // step size

}

parameters {

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

}

model {

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

    real pi;
    real Bfac;

    pi ← 3.1415926535;

    B0 ← R0 * r / N;

    B[1] ← B0;

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

    for (t in 2:T) {

        Bnoise[t] ~ normal(0,berr);
        Bfac ← exp(2*cos((2*pi/365)*t) - 2);
```

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

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

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

    }

    R0      ~ lognormal(1,1);
    r       ~ lognormal(1,1);
    sigma   ~ lognormal(1,1);
    re      ~ lognormal(1,1);
    Iinit   ~ normal(y[1], sigma);

}
```

E.6 IF2 SIRS Code

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

```
/*  Dexter Barrows
    dbarrows.github.io
    McMaster University
    2016

    */

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

#define Treal       100         // time to simulate over
#define R0true      3.0         // infectiousness
#define rtrue       0.1         // recovery rate
#define retrue      0.05        // resusceptibility rate
#define Nreal       500.0       // population size
#define etatrue     0.5         // real drift attraction strength
```

```
1  24 #define berrtrue    0.5         // real beta drift noise
2  25 #define merr        5.0         // expected measurement error
3  26 #define I0          5.0         // Initial infected individuals
4  27
5  28 #define PSC         0.5         // scale factor for more sensitive
6        parameters
7  29
8  30 #include <Rcpp.h>
9  31 using namespace Rcpp;
10 32
11 33 struct State {
12 34     double S;
13 35     double I;
14 36     double R;
15 37 };
16 38
17 39 struct Particle {
18 40     double R0;
19 41     double r;
20 42     double re;
21 43     double sigma;
22 44     double eta;
23 45     double berr;
24 46     double B;
25 47     double S;
26 48     double I;
27 49     double R;
28 50     double Sinit;
29 51     double Iinit;
30 52     double Rinit;
31 53 };
32 54
33 55 struct ParticleInfo {
34 56     double R0mean;      double R0sd;
35 57     double rmean;       double rsd;
36 58     double remean;      double resd;
37 59     double sigmamean;   double sigmasd;
38 60     double etamean;     double etasd;
39 61     double berrmean;    double berrsd;
40 62     double Sinitmean;   double Sinitsd;
41 63     double Iinitmean;   double Iinitsd;
42 64     double Rinitmean;   double Rinitsd;
43 65 };
44 66
45 67
46 68 int timeval_subtract (double *result, struct timeval *x, struct
47       timeval *y);
48 69 int check_double(double x,double y);
49 70 void exp_euler_SIRS(double h, double t0, double tn, int N, Particle *
50       particle);
51 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)) {
```

```
167             State sMeans;
168             getStateMeans(&sMeans, particles, NP);
169             statemeans(0,0) = sMeans.S;
170             statemeans(0,1) = sMeans.I;
171             statemeans(0,2) = sMeans.R;
172         }
173 
174         for (int t = 1; t < T; t++) {
175 
176             // generate individual predictions and weight
177             for (int n = 0; n < NP; n++) {
178 
179                 exp_euler_SIRS(1.0/7.0, (double) t-1, (double) t, N,
                        &particles[n]);
180 
181                 double merr_par = particles[n].sigma;
182                 double y_diff   = data[t] - particles[n].I;
183 
184                 w[n] = 1.0/(merr_par*sqrt(2.0*M_PI)) * exp( - y_diff*
                        y_diff / (2.0*merr_par*merr_par) );
185 
186             }
187 
188             // cumulative sum
189             for (int n = 1; n < NP; n++) {
190                 w[n] += w[n-1];
191             }
192 
193             // save particle states to resample from
194             for (int n = 0; n < NP; n++){
195                 copyParticle(&particles_old[n], &particles[n]);
196             }
197 
198             // resampling
199             for (int n = 0; n < NP; n++) {
200 
201                 double w_r = randu() * w[NP-1];
202                 int i = 0;
203                 while (w_r > w[i]) {
204                     i++;
205                 }
206 
207                 // i is now the index to copy state from
208                 copyParticle(&particles[n], &particles_old[i]);
209 
210             }
211 
212             // between-iteration perturbations, not after last time
                    step
213             if (t < (T-1))
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
```

```
1  412         do {
2  413             Iinitcan = particles[n].Iinit + spreadIinit*randn();
3  414         } while (Iinitcan < 0 || Iinitcan > 500);
4  415         particles[n].Iinit = Iinitcan;
5  416         particles[n].Sinit = N - Iinitcan;
6  417
7  418     }
8  419
9  420 }
10 421
11 422
12 423 /*  Convinience function for particle resampling process
13 424
14 425     */
15 426 void copyParticle(Particle * dst, Particle * src) {
16 427
17 428     dst->R0     = src->R0;
18 429     dst->r      = src->r;
19 430     dst->re     = src->re;
20 431     dst->sigma  = src->sigma;
21 432     dst->eta    = src->eta;
22 433     dst->berr   = src->berr;
23 434     dst->B      = src->B;
24 435     dst->S      = src->S;
25 436     dst->I      = src->I;
26 437     dst->R      = src->R;
27 438     dst->Sinit  = src->Sinit;
28 439     dst->Iinit  = src->Iinit;
29 440     dst->Rinit  = src->Rinit;
30 441
31 442 }
32 443
33 444 void particleDiagnostics(ParticleInfo * partInfo, Particle *
34     particles, int NP) {
35 445
36 446     double  R0mean      = 0.0,
37 447             rmean       = 0.0,
38 448             remean      = 0.0,
39 449             sigmamean   = 0.0,
40 450             etamean     = 0.0,
41 451             berrmean    = 0.0,
42 452             Sinitmean   = 0.0,
43 453             Iinitmean   = 0.0,
44 454             Rinitmean   = 0.0;
45 455
46 456     // means
47 457
48 458     for (int n = 0; n < NP; n++) {
49 459
50 460         R0mean      += particles[n].R0;
51 461         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.

```
## 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
```

```
    BSI ← numeric(nloc)
    rI ← numeric(nloc)

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

    B0 ← rep(R0*r/N, nloc)

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

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

    h ← 1 / steps

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

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

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

        rI ← r*I

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

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

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

```

```
    }

  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
#     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.

```
## 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)
StocSSIRstan <- function(ymat, pars, T, steps, neinum, neibmat, berrmat, bmatlim) {

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

```

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

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

    B0 ← rep(R0*r/N, nloc)

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

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

    h ← 1 / steps

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

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


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

        rI ← r*I

        dS ← -BSI
```

```
        dI ← BSI - rI
        dR ← rI

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

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

    }

  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
#       phi = 0.5 )   # interconnectivity degree
```

F.3 RStan Spatial SIR Code

This code implements a Spatial SIR model in Rstan.

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

data {

    int     <lower=1>   T;      // total integration steps
    int     <lower=1>   nloc;   // number of locations
    real                y[nloc, T];   // observed number of cases
    int     <lower=1>   N;      // population size
    real                h;      // step size
```

```
    int     <lower=0>   neinum[nloc];        // number of neighbors
        each location has
    int                 neibmat[nloc, nloc]; // neighbor list for
        each location

}

parameters {

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

}

model {

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

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

    B0 <- R0 * r / N;

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

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

            Bnoise[loc, t] ~ normal(0,berr);
```

```
            B[loc, t] <- exp( log(B[loc, t-1]) + eta * ( log(B0) - log
                (B[loc, t-1]) ) + Bnoise[loc, t] );

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

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

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

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

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

        }
    }

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

}
```

F.4 IF2 Spatial SIR Code

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

```
/*  Dexter Barrows
    dbarrows.github.io
    McMaster University
    2016

    */

#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);
double randu();
double randn();

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

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

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

    double w[NP];           // particle weights

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

    // START PASSES THROUGH DATA

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


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

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

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

```
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],
                        neinum, neibmat, nloc);
115
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
```

```
1  145                 // i is now the index to copy state from
2  146                 copyParticle(&particles[n], &particles_old[i], nloc);
3  147
4  148             }
5  149
6  150             // between-iteration perturbations, not after last time
                       step
8  151             if (t < (T-1))
9  152                 perturbParticles(particles, N, NP, nloc, pass,
10                     coolrate);
11 153
12 154             if (pass == (nPasses-1)) {
13 155                 double means[nloc];
14 156                 for (int loc = 0; loc < nloc; loc++) {
15 157                     means[loc] = 0.0;
16 158                     for (int n = 0; n < NP; n++) {
17 159                         means[loc] += particles[n].I[loc] / NP;
18 160                     }
19 161                     infecmeans(loc, t) = means[loc];
20 162                 }
21 163             }
22 164
23 165         }
24 166
25 167         // between-pass perturbations, not after last pass
26 168         if (pass < (nPasses + 1))
27 169             perturbParticles(particles, N, NP, nloc, pass, coolrate);
28 170
29 171     }
30 172
31 173     // pack parameter data (minus initial conditions)
32 174     for (int n = 0; n < NP; n++) {
33 175         paramdata(n, 0) = particles[n].R0;
34 176         paramdata(n, 1) = particles[n].r;
35 177         paramdata(n, 2) = particles[n].sigma;
36 178         paramdata(n, 3) = particles[n].eta;
37 179         paramdata(n, 4) = particles[n].berr;
38 180         paramdata(n, 5) = particles[n].phi;
39 181     }
40 182
41 183     // Pack initial condition data
42 184     for (int n = 0; n < NP; n++) {
43 185         for (int loc = 0; loc < nloc; loc++) {
44 186             initInfec(loc, n) = particles[n].Iinit[loc];
45 187         }
46 188     }
47 189
48 190     // Pack final state means data
49 191     double Smeans[nloc], Imeans[nloc], Rmeans[nloc], Bmeans[nloc];
50 192     for (int loc = 0; loc < nloc; loc++) {
51 193         Smeans[loc] = 0.0;
```

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


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



}


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

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

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

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

    // create last state vectors
    double S_last[nloc];
```

```
239     double I_last[nloc];
240     double R_last[nloc];
241     double B_last[nloc];
242
243     double R0   = particle->R0;
244     double r    = particle->r;
245     double B0   = R0 * r / N;
246     double eta  = particle->eta;
247     double berr = particle->berr;
248     double phi  = particle->phi;
249
250     for(int t = 0; t < num_steps; t++) {
251
252         for (int loc = 0; loc < nloc; loc++) {
253             S_last[loc] = S[loc];
254             I_last[loc] = I[loc];
255             R_last[loc] = R[loc];
256             B_last[loc] = B[loc];
257         }
258
259         for (int loc = 0; loc < nloc; loc++) {
260
261             B[loc] = exp( log(B_last[loc]) + eta*(log(B0) - log(
                    B_last[loc])) + berr*randn() );
262
263             int n = neinum[loc];
264             double sphi = 1.0 - phi*( (double) n/(n+1.0) );
265             double ophi = phi/(n+1.0);
266
267             double nBIsum = 0.0;
268             for (int j = 0; j < n; j++)
269                 nBIsum += B_last[(int) neibmat(loc, j) - 1] * I_last
                        [(int) neibmat(loc, j) - 1];
270
271             double BSI = S_last[loc]*( sphi*B_last[loc]*I_last[loc] +
                     ophi*nBIsum );
272             double rI  = r*I_last[loc];
273
274             // get derivatives
275             double dS = - BSI;
276             double dI = BSI - rI;
277             double dR = rI;
278
279             // step forward by h
280             S[loc] += h*dS;
281             I[loc] += h*dI;
282             R[loc] += h*dR;
283
284         }
285
286     }
```

```
287
288 }
289
290 /*  Initializes particles
291     */
292 void initializeParticles(Particle ** particles, int NP, int nloc, int
        N) {
293
294     // allocate space for doubles
295     *particles = (Particle*) malloc (NP*sizeof(Particle));
296
297     // allocate space for arays inside particles
298     for (int n = 0; n < NP; n++) {
299         (*particles)[n].S = (double*) malloc(nloc*sizeof(double));
300         (*particles)[n].I = (double*) malloc(nloc*sizeof(double));
301         (*particles)[n].R = (double*) malloc(nloc*sizeof(double));
302         (*particles)[n].B = (double*) malloc(nloc*sizeof(double));
303         (*particles)[n].Iinit = (double*) malloc(nloc*sizeof(double))
                ;
304     }
305
306     // initialize all all parameters
307     for (int n = 0; n < NP; n++) {
308
309         double R0can, rcan, sigmacan, Iinitcan, etacan, berrcan,
                phican;
310
311         do {
312             R0can = R0true + R0true*randn();
313         } while (R0can < 0);
314         (*particles)[n].R0 = R0can;
315
316         do {
317             rcan = rtrue + rtrue*randn();
318         } while (rcan < 0);
319         (*particles)[n].r = rcan;
320
321         for (int loc = 0; loc < nloc; loc++)
322             (*particles)[n].B[loc] = (double) R0can * rcan / N;
323
324         do {
325             sigmacan = merr + merr*randn();
326         } while (sigmacan < 0);
327         (*particles)[n].sigma = sigmacan;
328
329         do {
330             etacan = etatrue + PSC*etatrue*randn();
331         } while (etacan < 0 || etacan > 1);
332         (*particles)[n].eta = etacan;
333
334         do {
```

```
1  335             berrcan = berrtrue + PSC*berrtrue*randn();
2  336         } while (berrcan < 0);
3  337         (*particles)[n].berr = berrcan;
4  338
5  339         do {
6  340             phican = phitrue + PSC*phitrue*randn();
7  341         } while (phican <= 0 || phican >= 1);
8  342         (*particles)[n].phi = phican;
9  343
10 344         for (int loc = 0; loc < nloc; loc++) {
11 345             do {
12 346                 Iinitcan = I0 + I0*randn();
13 347             } while (Iinitcan < 0 || N < Iinitcan);
14 348             (*particles)[n].Iinit[loc] = Iinitcan;
15 349         }
16 350
17 351     }
18 352
19 353 }
20 354
21 355 /*  Particle pertubation function to be run between iterations and
22         passes
23 356
24 357     */
25 358 void perturbParticles(Particle * particles, int N, int NP, int nloc,
26         int passnum, double coolrate) {
27 359
28 360     //double coolcoef = exp( - (double) passnum / coolrate );
29 361     double coolcoef = pow(coolrate, passnum);
30 362
31 363     double spreadR0     = coolcoef * R0true / 10.0;
32 364     double spreadr      = coolcoef * rtrue / 10.0;
33 365     double spreadsigma  = coolcoef * merr / 10.0;
34 366     double spreadIinit  = coolcoef * I0 / 10.0;
35 367     double spreadeta    = coolcoef * etatrue / 10.0;
36 368     double spreadberr   = coolcoef * berrtrue / 10.0;
37 369     double spreadphi    = coolcoef * phitrue / 10.0;
38 370
39 371     double R0can, rcan, sigmacan, Iinitcan, etacan, berrcan, phican;
40 372
41 373     for (int n = 0; n < NP; n++) {
42 374
43 375         do {
44 376             R0can = particles[n].R0 + spreadR0*randn();
45 377         } while (R0can < 0);
46 378         particles[n].R0 = R0can;
47 379
48 380         do {
49 381             rcan = particles[n].r + spreadr*randn();
50 382         } while (rcan < 0);
51 383         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
```

```
}


double randu() {

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

}


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

    double x1, x2, w, y1;

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

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

    return y1;

}
```

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.

```
/*  Dexter Barrows
    dbarrows.github.io
    McMaster University
    2016

    */

#include <cuda.h>
#include <iostream>
#include <fstream>
```

```
1  11 #include <curand.h>
2  12 #include <curand_kernel.h>
3  13 #include <string>
4  14 #include <sstream>
5  15 #include <cmath>
6  16
7  17 #include "timer.h"
8  18 #include "rand.h"
9  19 #include "readdata.h"
10 20
11 21 #define NP          (2*2500)    // number of particles
12 22 #define N           500.0       // population size
13 23 #define R0true      3.0         // infectiousness
14 24 #define rtrue       0.1         // recovery rate
15 25 #define etatrue     0.5         // real drift attraction strength
16 26 #define berrtrue    0.5         // real beta drift noise
17 27 #define phitrue     0.5         // real connectivity strength
18 28 #define merr        10.0        // expected measurement error
19 29 #define I0          5.0         // Initial infected individuals
20 30 #define PSC         0.5         // sensitive parameter perturbation
21       scaling
22 31 #define NLOC        10
23 32
24 33 #define PI      3.141592654f
25 34
26 35 // Wrapper for CUDA calls, from CUDA API
27 36 // Modified to also print the error code and string
28 37 # define CUDA_CALL(x) do { if ((x) != cudaSuccess ) {
29          \
30 38     std::cout << " Error at " << __FILE__ << ":" << __LINE__ << std::
31          endl;      \
32 39     std::cout << " Error was " << x << " " << cudaGetErrorString(x)
33          << std::endl;   \
34 40     return EXIT_FAILURE ;}} while (0)
35            \
36 41
37 42 typedef struct {
38 43     float R0;
39 44     float r;
40 45     float sigma;
41 46     float eta;
42 47     float berr;
43 48     float phi;
44 49     float S[NLOC];
45 50     float I[NLOC];
46 51     float R[NLOC];
47 52     float B[NLOC];
48 53     float Iinit[NLOC];
49 54     curandState randState;  // PRNG state
50 55 } Particle;
51 56
```

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

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


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

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

    if (id < NP) {

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

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

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

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

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

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

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

        do {
```

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

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

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

        particles[id].randState = state;

    }

}

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

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

    if (id < NP) {

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

    }

}

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

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

    if (id < NP) {

        particles[id].R0 = R0true;
        particles[id].r = rtrue;
        particles[id].sigma = merr;
        particles[id].eta = etatrue;
        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 *
        neibmat, int nloc) {
166 
167     int id = blockIdx.x*blockDim.x + threadIdx.x;   // global id
168 
169     if (id < NP) {
170         // project forward
171         exp_euler_SSIR(1.0/7.0, 0.0, 1.0, &particles[id], neinum,
                neibmat, nloc);
172     }
173 
174 }
175 
176 __global__ void weight(float * data, Particle * particles, double * w
        , int t, int T, int nloc) {
177 
178     int id = blockIdx.x*blockDim.x + threadIdx.x;   // global id
179 
180     if (id < NP) {
181 
182         float merr_par = particles[id].sigma;
183 
184         // Get weight and save
185         double w_local = 1.0;
186         for (int loc = 0; loc < nloc; loc++) {
187             float y_diff = data[loc*T + t] - particles[id].I[loc];
188             w_local *= 1.0/(merr_par*sqrt(2.0*PI)) * exp( - y_diff*
                    y_diff / (2.0*merr_par*merr_par) );
189         }
190 
191         w[id] = w_local;
192 
193     }
194 
195 }
196 
197 __global__ void stashParticles (Particle * particles, Particle *
```

```
    particles_old, int nloc) {

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

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

}


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

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

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

}


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

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

    if (id < NP) {

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

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

    }

```

```
}

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

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

    if (id < nloc) {

        int loc = id;

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

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

    }

}

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

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

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

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

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

    if (id < NP) {

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

        do {
```

```
1  292             rcan = particles[id].r + spreadr*curand_normal(&particles
2                      [id].randState);
3  293         } while (rcan < 0);
4  294         particles[id].r = rcan;
5  295
6  296         do {
7  297             sigmacan = particles[id].sigma + spreadsigma*
8                      curand_normal(&particles[id].randState);
9  298         } while (sigmacan < 0);
10 299         particles[id].sigma = sigmacan;
11 300
12 301         do {
13 302             etacan = particles[id].eta + PSC*spreadeta*curand_normal
14                     (&particles[id].randState);
15 303         } while (etacan < 0 || etacan > 1);
16 304         particles[id].eta = etacan;
17 305
18 306         do {
19 307             berrcan = particles[id].berr + PSC*spreadberr*
20                     curand_normal(&particles[id].randState);
21 308         } while (berrcan < 0);
22 309         particles[id].berr = berrcan;
23 310
24 311         do {
25 312             phican = particles[id].phi + PSC*spreadphi*curand_normal
26                     (&particles[id].randState);
27 313         } while (phican <= 0 || phican >= 1);
28 314         particles[id].phi = phican;
29 315
30 316         for (int loc = 0; loc < nloc; loc++) {
31 317             do {
32 318                 Iinitcan = particles[id].Iinit[loc] + spreadIinit*
33                         curand_normal(&particles[id].randState);
34 319             } while (Iinitcan < 0 || Iinitcan > 500);
35 320             particles[id].Iinit[loc] = Iinitcan;
36 321         }
37 322
38 323     }
39 324
40 325 }
41 326
42 327
43 328 int main (int argc, char *argv[]) {
44 329
45 330
46 331     int T, nloc;
47 332
48 333     double restime;
49 334     struct timeval tdr0, tdr1, tdrMaster;
50 335
51 336     // Parse arguments ********************************************
```

```
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
```

```
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,
                    d_countmeans, 0, T, nloc);
479             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,
                    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
                    ************************
```

```
1  515
2  516             nThreads    = 32;
3  517             nBlocks     = ceil( (float) NP / nThreads);
4  518
5  519             stashParticles <<< nBlocks, nThreads >>> (particles,
6                      particles_old, nloc);
7  520             CUDA_CALL( cudaGetLastError() );
8  521             CUDA_CALL( cudaDeviceSynchronize() );
9  522
10 523
11 524             // Resampling
12                    ************************************************
13 525
14 526             nThreads    = 32;
15 527             nBlocks     = ceil( (float) NP/ nThreads);
16 528
17 529             resample <<< nBlocks, nThreads >>> (particles,
18                     particles_old, w, nloc);
19 530             CUDA_CALL( cudaGetLastError() );
20 531             CUDA_CALL( cudaDeviceSynchronize() );
21 532
22 533             // Reduction
23                    ************************************************
24 534
25 535             if (pass == 49) {
26 536
27 537                 nThreads = 1;
28 538                 nBlocks  = 10;
29 539
30 540                 reduceStates <<< nBlocks, nThreads >>> (particles,
31                        d_countmeans, t, T, nloc);
32 541                 CUDA_CALL( cudaGetLastError() );
33 542                 CUDA_CALL( cudaDeviceSynchronize() );
34 543
35 544             }
36 545
37 546             // Perturb particles
38                    ***************************************
39 547
40 548             nThreads    = 32;
41 549             nBlocks     = ceil( (float) NP/ nThreads);
42 550
43 551             perturbParticles <<< nBlocks, nThreads >>> (particles,
44                     nloc, pass, 0.975);
45 552             CUDA_CALL( cudaGetLastError() );
46 553             CUDA_CALL( cudaDeviceSynchronize() );
47 554
48 555
49 556         } // end time
50 557
51 558     } // end pass
```

```
559
560     std::cout.precision(10);
561
562     countmeans = (float*) malloc (nloc*T*sizeof(float));
563     cudaMemcpy(countmeans, d_countmeans, nloc*T*sizeof(float),
            cudaMemcpyDeviceToHost);
564
565     // stop master timer and print
566
567     gettimeofday (&tdrMaster, NULL);
568     timeval_subtract(&restime, &tdrMaster, &tdr0);
569     std::cout << "Time: " << getHRtime(restime) << std::endl;
570     std::cout << "Rawtime: " << restime << std::endl;
571
572     // Write results out
573
574     std::string filename = arg4;
575
576     std::cout << "Writing results to file '" << filename << "' ..."
            << std::endl;
577
578     std::ofstream outfile;
579     outfile.open(filename.c_str());
580
581     for(int loc = 0; loc < nloc; loc++) {
582         for (int t = 0; t < T; t++) {
583             outfile << countmeans[loc*T + t] << " ";
584         }
585         outfile << std::endl;
586     }
587
588     outfile.close();
589
590     cudaFree(d_data);
591     cudaFree(particles);
592     cudaFree(particles_old);
593     cudaFree(w);
594     cudaFree(d_neinum);
595     cudaFree(d_neibmat);
596     cudaFree(d_countmeans);
597
598     exit (EXIT_SUCCESS);
599
600 }
601
602
603 /*  Use the Explicit Euler integration scheme to integrate SIR model
        forward in time
604     float h     - time step size
605     float t0    - start time
606     float tn    - stop time
```

```
1  607     float * y   - current system state; a three-component vector
2             representing [S I R], susceptible-infected-recovered
3  608     */
4  609 __device__ void exp_euler_SSIR(float h, float t0, float tn, Particle
5        * particle, int * neinum, int * neibmat, int nloc) {
6  610
7  611     int num_steps = floor( (tn-t0) / h );
8  612
9  613     float * S = particle->S;
10 614     float * I = particle->I;
11 615     float * R = particle->R;
12 616     float * B = particle->B;
13 617
14 618     // create last state vectors
15 619     float * S_last = (float*) malloc (nloc*sizeof(float));
16 620     float * I_last = (float*) malloc (nloc*sizeof(float));
17 621     float * R_last = (float*) malloc (nloc*sizeof(float));
18 622     float * B_last = (float*) malloc (nloc*sizeof(float));
19 623
20 624     float R0    = particle->R0;
21 625     float r     = particle->r;
22 626     float B0    = R0 * r / N;
23 627     float eta   = particle->eta;
24 628     float berr  = particle->berr;
25 629     float phi   = particle->phi;
26 630
27 631     for(int t = 0; t < num_steps; t++) {
28 632
29 633         for (int loc = 0; loc < nloc; loc++) {
30 634             S_last[loc] = S[loc];
31 635             I_last[loc] = I[loc];
32 636             R_last[loc] = R[loc];
33 637             B_last[loc] = B[loc];
34 638         }
35 639
36 640         for (int loc = 0; loc < nloc; loc++) {
37 641
38 642             B[loc] = exp( log(B_last[loc]) + eta*(log(B0) - log(
39                    B_last[loc])) + berr*curand_normal(&(particle->
40                    randState)) );
41 643
42 644             int n = neinum[loc];
43 645             float sphi = 1.0 - phi*( (float) n/(n+1.0) );
44 646             float ophi = phi/(n+1.0);
45 647
46 648             float nBIsum = 0.0;
47 649             for (int j = 0; j < n; j++)
48 650                 nBIsum += B_last[neibmat[nloc*loc + j]-1] * I_last[
49                        neibmat[nloc*loc + j]-1];
50 651
51 652             float BSI = S_last[loc]*( sphi*B_last[loc]*I_last[loc] +
```

```
1                 ophi*nBIsum );
2   653           float rI  = r*I_last[loc];
3   654
4   655           // get derivatives
5   656           float dS = - BSI;
6   657           float dI = BSI - rI;
7   658           float dR = rI;
8   659
9   660           // step forward by h
10  661           S[loc] += h*dS;
11  662           I[loc] += h*dI;
12  663           R[loc] += h*dR;
13  664
14  665       }
15  666
16  667   }
17  668
18  669   free(S_last);
19  670   free(I_last);
20  671   free(R_last);
21  672   free(B_last);
22  673
23  674 }
24  675
25  676 /*  Convinience function for particle resampling process
26  677     */
27  678 __device__ void copyParticle(Particle * dst, Particle * src, int nloc
28          ) {
29  679
30  680   dst->R0     = src->R0;
31  681   dst->r      = src->r;
32  682   dst->sigma  = src->sigma;
33  683   dst->eta    = src->eta;
34  684   dst->berr   = src->berr;
35  685   dst->phi    = src->phi;
36  686
37  687   for (int n = 0; n < nloc; n++) {
38  688       dst->S[n]       = src->S[n];
39  689       dst->I[n]       = src->I[n];
40  690       dst->R[n]       = src->R[n];
41  691       dst->B[n]       = src->B[n];
42  692       dst->Iinit[n]   = src->Iinit[n];
43  693   }
44  694
45  695 }
46  696
47  697 /*  Convert memory size in bytes to human-readable format
48  698     */
49  699 std::string getHRmemsize (size_t memsize) {
50  700
51  701   std::stringstream ss;
```

```
    std::string valstring;

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

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

    valstring = ss.str();

    return valstring;

}


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

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

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

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

    valstring = ss.str();

    return valstring;

}
```

Figure [F.1] shows the running times for parameter fitting as compared to IF2 and HMC.

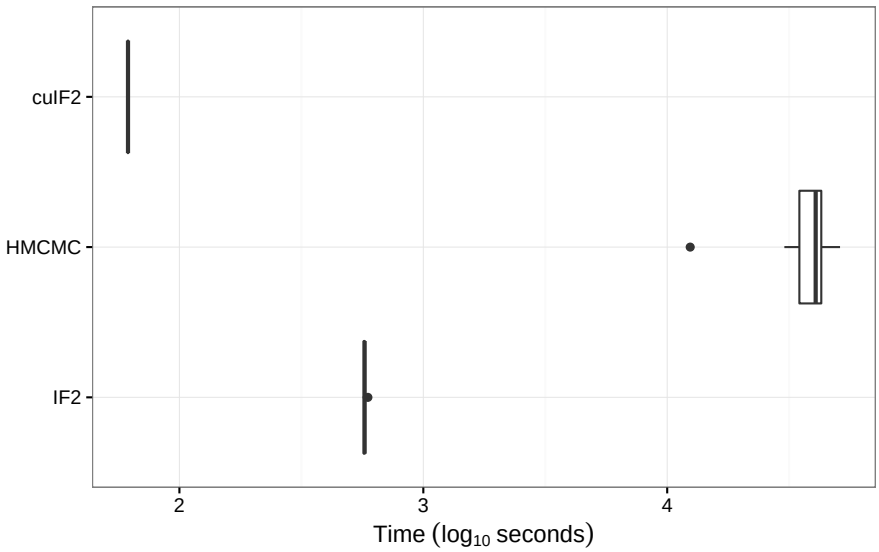

Figure F.1: Running times for fitting the spatial SIR model to data.

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