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

McMaster University ©Copyright by Dexter Barrows, April 2016

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, 145

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: IF2, Hamiltonian MCMC, 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 HMCMC, 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 [27][9][39]. Successful intervention relies on accurate predictions of the number of cases, when they will occur, and where. Without this information it is difficult to efficiently allocate resources, a critical step in curbing the size and breadth of an epidemic.

Despite the importance of reliable forecasts, obtaining them remains a challenge from both a theoretical and practical standpoint [27]. 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 [33], thus uncertainty in what we observe in building mathematical models of disease spread must be accounted for from the get-go. 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 [22]. With these caveats and concerns acknowledged, we can turn to a discussion of technique.

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

Phenomenological methods operate purely on data, fitting models that do not try to reconstruct disease dynamics, but rather focus purely on trend. A long-standing and widely-used example is the Autoregressive Integrated Moving Average (ARIMA)

model. ARIMA assumes a linear underlying process and Gaussian error distributions. It uses three parameters representing the degree of autoregression (p), integration (trend removal) (d), and the moving average (q), where the orders of the autoregression and the moving average are determined through the use of an autocorrelation function (ACF) and partial autocorrelation function (PACF), respectively, applied to the the data *a priori* [40].

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

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

where S, I, and R are the number of individuals in each compartment, β is the "force" of infection acting on the susceptible population, and γ is a recovery rate. We also let $\beta = R_0 r/N$, where R_0 is the number of secondary cases per infected individual, 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 are 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].

Combining the phenomenological and mechanistic approaches are the semi-mechanistic techniques. These methods use a model to define the expected underlying dynamics of the system, but integrate data into the model in order to refine estimates of the model parameters and produce more accurate forecasts. Typically the first step in implementing such a technique is fitting the desired model to existing data. There are many ways to do this, most of which fall into two main categories: particle filter-based (PF) methods [3][31][39], and Markov chain Monte Carlo-based (MCMC) methods [2][26]. 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 di-

rectly integrated without the need to refit the entire model. Normally, MCMC-based machinery must refit the entire model whereas PF-based approaches can sometimes integrate data in an on-line fashion.

Another, broader, distinction among techniques can be drawn between those that rely on assumptions of linearity, and those that make no such assumption. As epidemic dynamics are highly non-linear, it can be questionable as to even consider linear approaches to epidemic forecasting at all. In particular, stalwart approaches such as ARIMA and the venerable Kalman filter face a distinct (at least theoretical) disadvantage in the face of newer PF-based methods [39][36]. Additionally, these methods are very-well-studied, and further work showing their viability would likely prove extraneous in the modern academic landscape.

Somewhat frustratingly, there exists no "gold standard" in forecasting [27][39][9]. 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 SSE, peak time/duration/intensity, correlation tests, or RMSE, among others [28][9]. Thus is is difficult to select the best tool for the job when faced with a forecasting problem.

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

From MCMC-based methods we have selected Hamiltonian MCMC [26], a (slightly) less cutting-edge but nonetheless highly effective technique. We are using HMCMC through an implementation in the R package RStan [8], which at its core uses HMCMC, 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 HMCMC backend. In fact the developers' origin''al 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 HMCMC. In the end, the scope of the project grew to include the development and subsequent integration of the No-U-Turn Sampler (NUTS) [17], and an implementation of automatic differentiation machinery [34].

For PF-based methods we have selected IF2 [20], 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 [19], but aims to be simpler, faster, and more accurate. Theoretical justification and synthetic testing indicates that IF2 meets these goals, and as such the authors recommend skipping MIF and jumping straight to IF2 if an algorithm of their variety is sought. And so, we are doing just that. We wrote our own IF2 implementation in C++ and integrated it into R using the Rcpp package [12]. The developers of MIF and IF2 have their own R package that implements MIF and IF2, Partially Observed Markov Processes (POMP) [21][23], 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) [36][35][18][13], combined with Dewdrop Regression [13]. These methods stand on their own as a unique take on the forecasting problem, and bear little resemblance to other methodology. The virtues of these techniques have been long-extolled by their developers, but their efficacy when compared to competing methods has not been well-studied. This work will mark one of the first times this has been done.

This paper will begin with begin with descriptions of HMCMC and IF2 with examples of simple model fittings 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 HMCMC, 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 HMCMC frameworks, along with Dewdrop Regression combined with S-mapping. Finally, a summary of these results, and a discussion of parallel computing and future directions will finish the paper in Chapter 8.

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

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

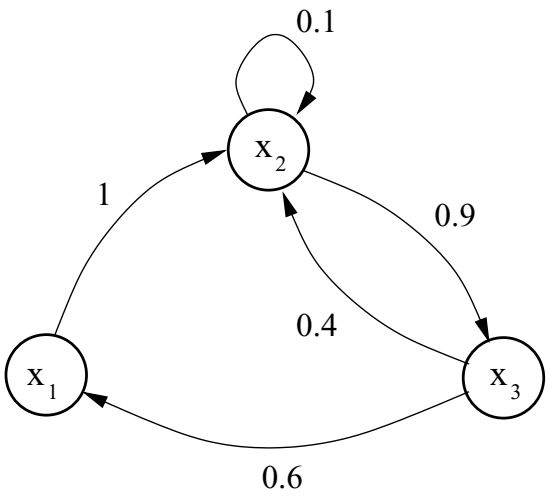

Figure 2.1: A finite state machine. States are shown as graph nodes, and the probability of transitioning from one particular state to another is shown as a weighted graph edge. [2]

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 some standard Maximum Likelihood approaches, $\mathcal{L}(\theta)$ is searched to find a value of θ that maximizes $\mathcal{L}(\theta)$, then this θ is taken to be the most likely true value. Here our aim is to not just maximize the likelihood but to also explore the area around it [2].

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

Algorithm 1: Metropolis MCMC

/* Select a starting point */
Input : Initialize $\theta^{(1)}$
for $i = 2 : N$ **do**
  /* Sample */
  $\theta^* \sim q(\cdot|\theta^{(i-1)})$
  $u \sim \mathcal{U}(0, 1)$
  /* Evaluate acceptance ratio */
  $r \leftarrow \frac{\mathcal{L}(\theta^*)p(\theta^*)}{\mathcal{L}(\theta)p(\theta)}$
  /* Step acceptance criterion */
  if $u < \min\{1, r\}$ **then**
    $\theta^{(i)} = \theta^*$
  else
    $\theta^{(i)} = \theta^{(i-1)}$
/* Samples from approximated posterior distribution */
Output: Chain of samples $(\theta^{(1)}, \theta^{(2)}, ..., \theta^{(N)})$

2.6 Burn-in

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

Burning in a chain is the act of running the MCMC algorithm normally without saving first M samples. As we are seeking a chain of length N, the total computation will be equivalent to generating a chain of length $M + N$ [2].

2.7 Thinning

Some models will require very long chains to get a good approximation of the posterior, which will consequently require a non-trivial amount of computer storage. One way to reduce the burden of storing so many samples is by thinning. This involves saving only every n^{th} step, which should still give a decent approximate of the posterior (since the chain has time to explore a large portion of the parameter space), but require less room to store [24].

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 or HMCMC) [26]. Prior to the advent of HMCMC, some work was conducted exploring adaptive step-sizing using MCMC-based methods, but lack strong theoretical justification, and can lead to some samples being drawn from an incorrect distribution [26].

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

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

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

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

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

The Hamiltonian of the system is defined as

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

where the Hamiltonian dynamics of the combined system can be simulated using the following system of ODEs:

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

.

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

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

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

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

Together, we have Algorithm [2].

Note that the parameters ε and L have to be tuned in order to maintain stability and maximize efficiency, a sometimes non-trivial process [26]. However, some recent algorithms, such as the No U-Turn sampler implemented in RStan, and adaptively select appropriate values automatically during the sampling process [17].

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))
```

2.9 RStan Fitting

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

The synthetic data was produced by taking the solution to a basic SIR ODE model, sampling it at regular intervals, and perturbing those values by adding in observation noise. The SIR model used was outlined in the introduction in Equation [1.1].

The solution to this system was obtained using the `ode()` function from the `deSolve` package. The required derivative array function in the format required by `ode()` was specified as the gradient in Equation [1.1].

The true parameter values were set to $R_0 = 3.0, r = 0.1, N = 500$. The initial conditions were set to 5 infectious individuals, 495 people susceptible to infection, and no one had yet recovered from infection and been removed. The system was integrated over $[0, 100]$ with infected counts drawn at each integer time step.

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

Figure [2.2] shows the system simulation results.

The Hamiltonian MCMC model fitting was done using Stan (`http://mc-stan.org/`), a program written in `C++` that does Baysian statistical inference using Hamiltonian MCMC. Stan's R interface (`http://mc-stan.org/interfaces/rstan.html`) was used to ease implementation.

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

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

Figure [2.3] shows the traceplot for the the post-warmup chain data returned by the `stan()` function in the `fit` object. We see that the chains are mixing well and convergence has likely been reached.

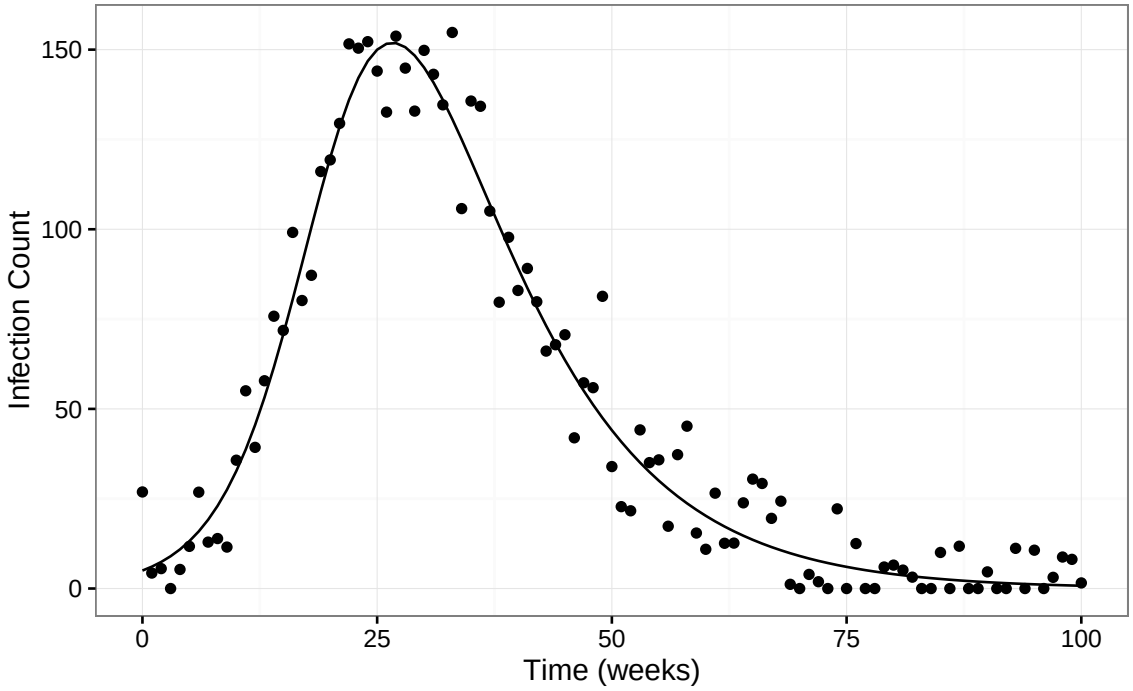

Figure 2.2: True SIR ODE solution infected counts, and with added observation noise.

Figure [2.4] shows the chain data including the warmup samples in. We can see why is 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.

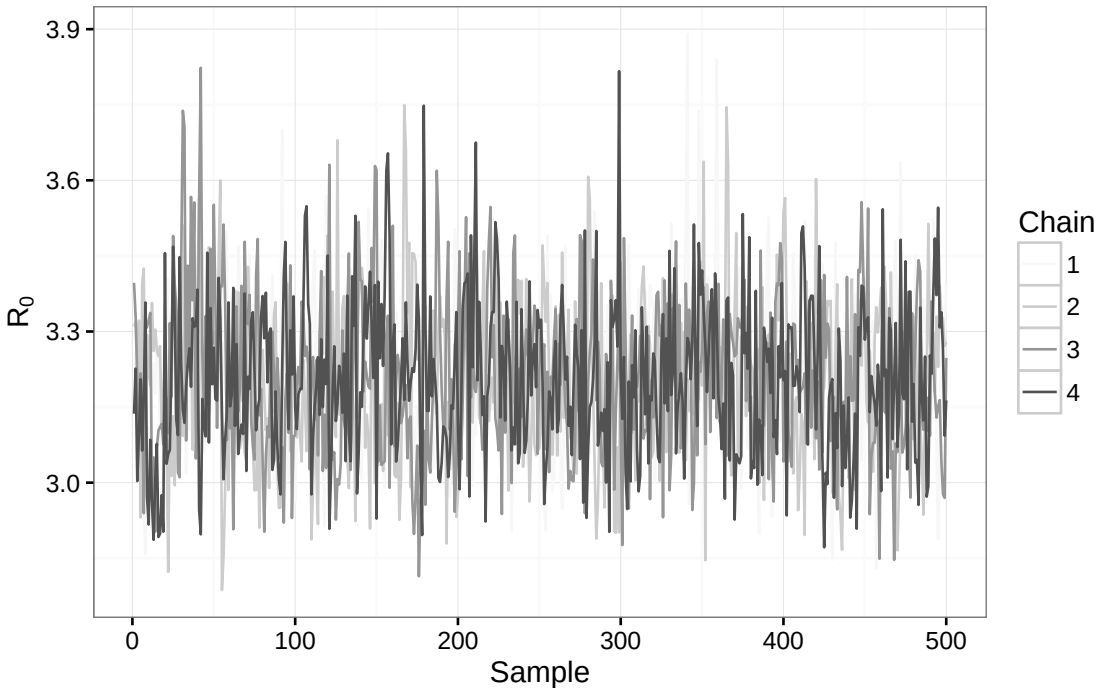

Figure 2.3: Traceplot of samples drawn for parameter R_0, excluding warmup.

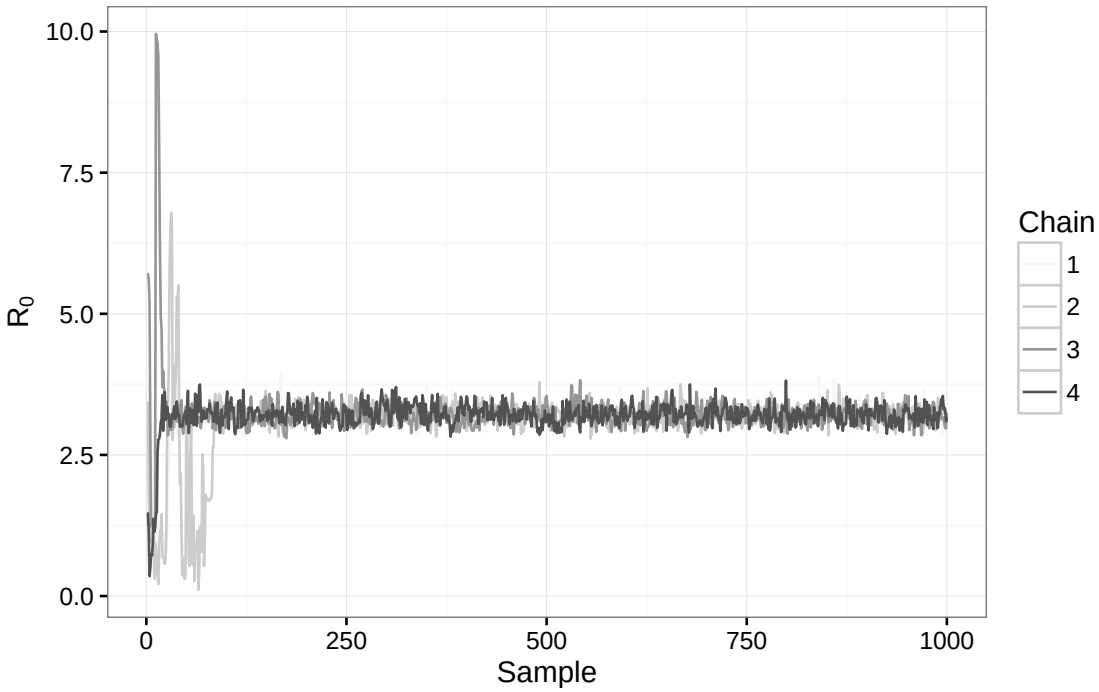

Figure 2.4: Traceplot of samples drawn for parameter R_0, including warmup.

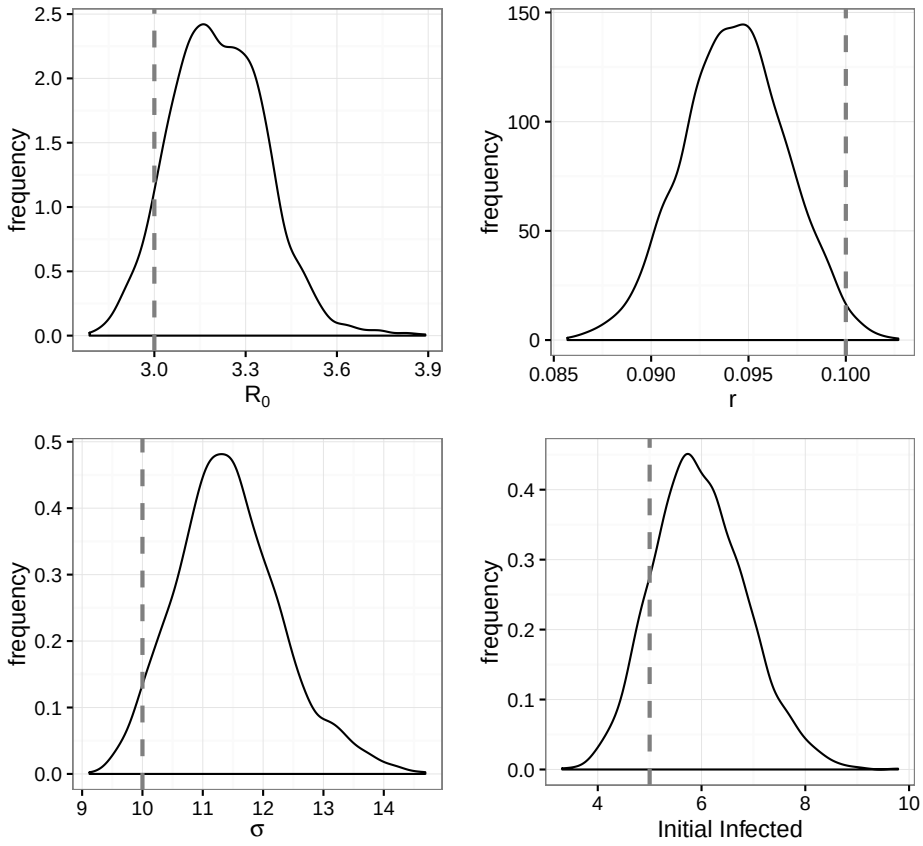

Figure 2.5: Kernel density estimates produced by Stan. Dashed lines show true parameter values.

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 [19][20] 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 [19], Iterated Filtering 2 [20] 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 $R_0 = 3.0, r = 0.1, N = 500$, initial conditions were set to 5 infectious individuals, 495 people susceptible to infection, and no one had yet recovered from infection and been removed, and observation error was taken to be $\varepsilon_{obs} \sim \mathcal{N}(0, \sigma)$, where individual values were drawn for each synthetic data point.

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

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

There are three primary reasons we implemented our own version of IF2 instead of using POMP. First, POMP does not provide final particle state distributions, making it difficult to calibrate the algorithm parameters against the parameters used in RStan (this procedure is described in the next chapter). Second, it is prudent to cross-check the validity of an algorithm using another implementation. Third, this code can then serve as a jumping-off point for further development using Graphics Processing Unit acceleration (outlined in Chapter 8). We must acknowledge the disadvantages as well: POMP has been extensively vetted with real-world usage, and using it would require far lees 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 HMCMC, the distributions are not perfect, but are promising. Unlike with HMCMC, these distributions are not meant to consist of samples from the true posterior distribution, but rather serve a diagnostic role.

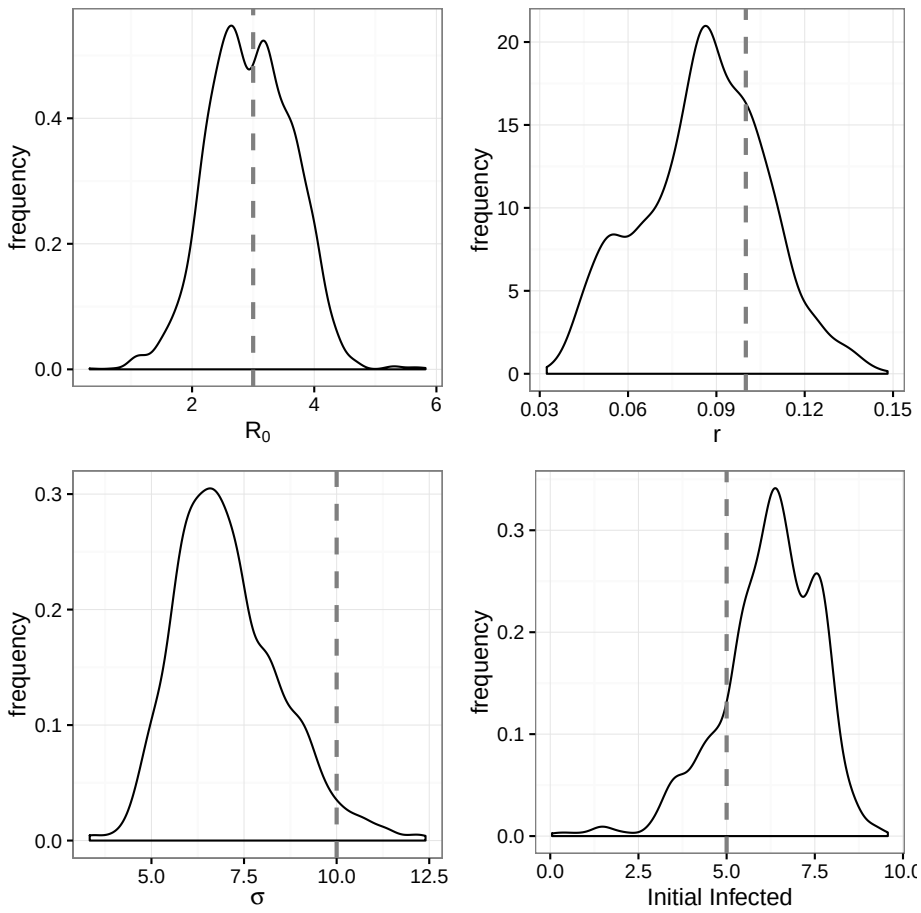

Figure 3.1: Kernel estimates for four essential system parameters. True values are indicated by dashed lines.

Chapter 4

Parameter Fitting

4.1 Fitting Setup

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

To form such a model, we will take a deterministic SIR ODE model specified in Equation [1.1] and add process noise by allowing β to follow a geometric random walk given by

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

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

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

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

where ϵ_t is normally distributed noise with mean 0 and standard deviation σ_E. This process has a theoretical expected mean of $\mu = c/(1-\rho)$ and variance $\sigma = \sigma_E^2/(1-\rho^2)$.

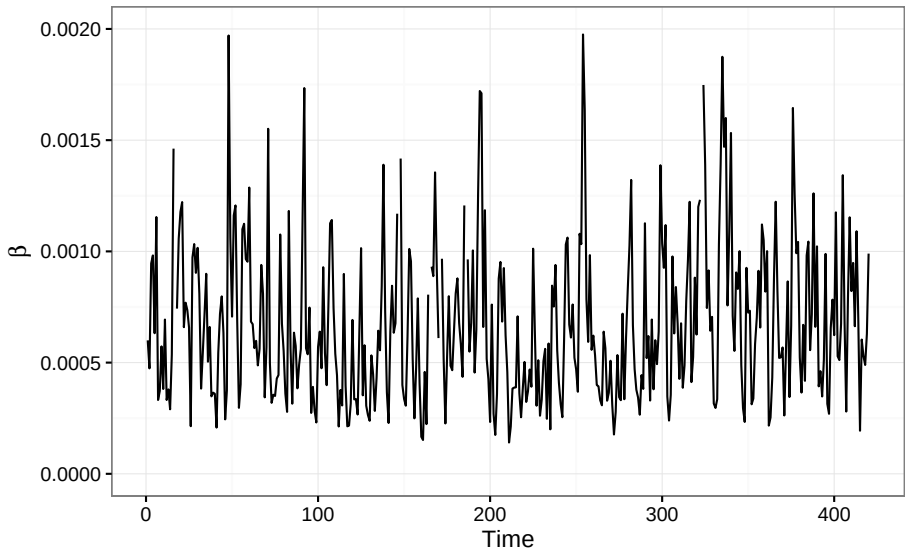

Figure 4.1: Simulated geometric autoregressive process show in Equation [4.1].

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].

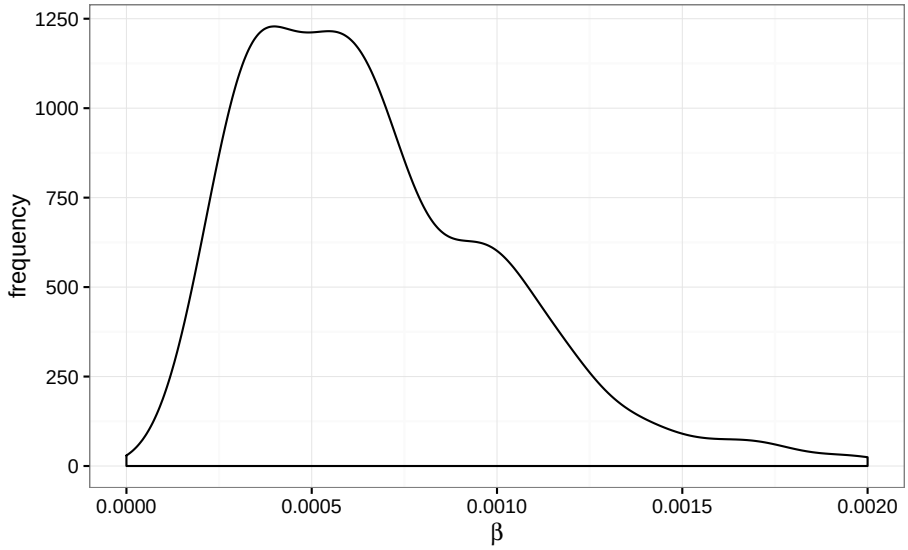

Figure 4.2: Density plot of values shown if Figure[4.1].

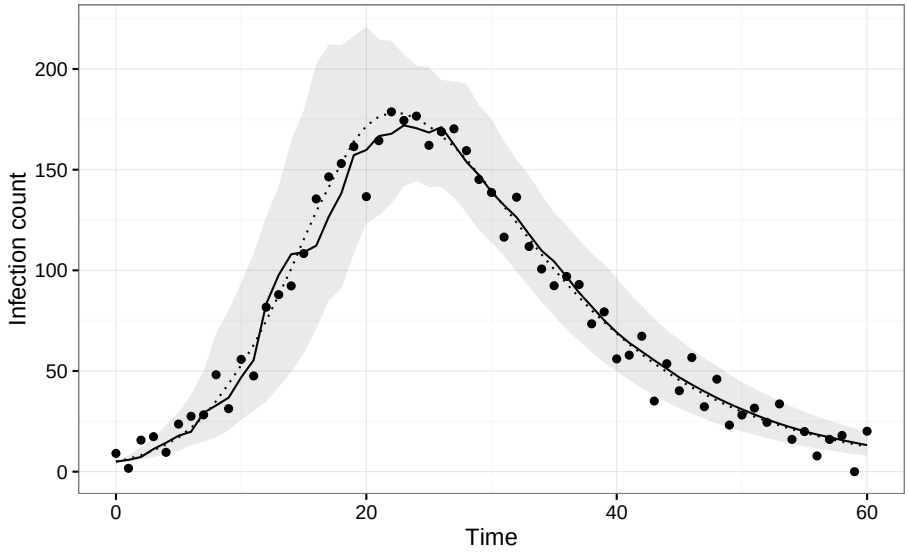

Figure 4.3: Stochastic SIR model simulated using an explicit Euler stepping scheme. The solid line is a single random trajectory, the dots show the data points obtained by adding in observation error defined as $\epsilon_{\text{obvs}} = \mathcal{N}(0, 10)$, and the grey ribbon is centre 95th quantile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

4.2 Calibrating Samples

In order to compare HMCMC and IF2 we need to set up a fair and theoretically justified way to select the number of samples to draw for the HMCMC iterations and the number of particles to use for IF2. 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 HMCMC, but HMCMC produces draws that more efficiently cover the sampling space. Thus we cannot just set the number of HMCMC 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) [15].

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

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

4.3 IF2 Fitting

Now we will use an implementation of the IF2 algorithm to attempt to fit the stochastic SIR model to the previous data. The goal here is just parameter inference, but since IF2 works by applying a series 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 [21][23], 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		HMCMC	
Name	True	Fit	Error	Fit	Error
R_0	3.0	3.27	9.08×10^{-2}	3.12	1.05×10^{-1}
r	10^{-1}	1.04×10^{-1}	3.61×10^{-2}	9.99×10^{-2}	-7.56×10^{-4}
Initial Infected	5	7.90	5.80×10^{-1}	6.64	3.28×10^{-1}
σ	10	8.84	-1.15×10^{-1}	8.5	-1.50×10^{-1}
η	5×10^{-1}	5.87×10^{-1}	1.73×10^{-1}	4.57×10^{-1}	-8.27×10^{-2}
ε_{err}	5×10^{-1}	1.63×10^{-1}	-6.73×10^{-1}	1.60×10^{-1}	-6.80×10^{-1}

Figure 4.4: Fitting errors.

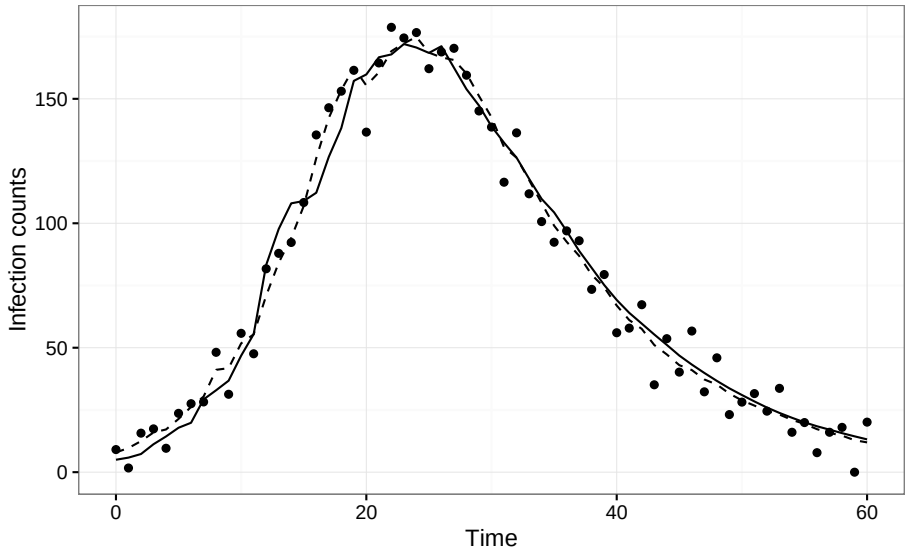

Figure 4.5: True system trajectory (solid line), observed data (dots), and IF2 estimated real state (dashed line).

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.

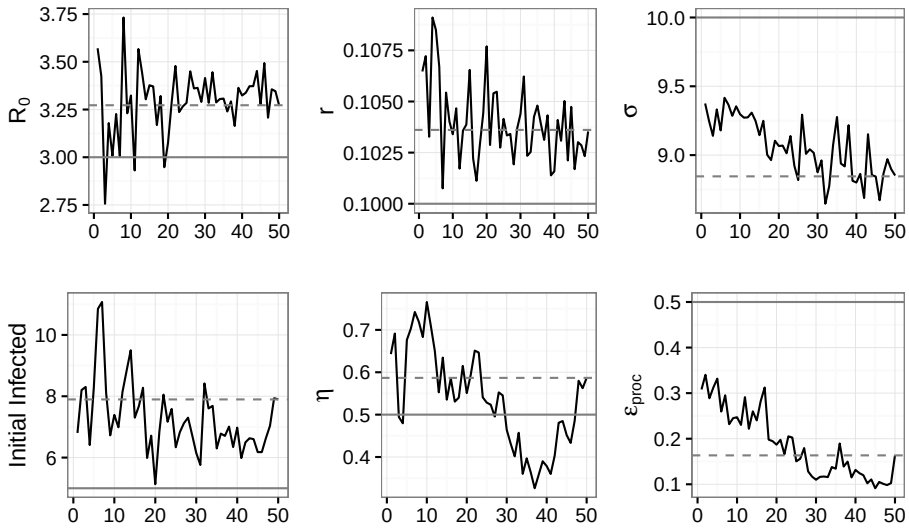

Figure 4.6: The horizontal axis shows the IF2 pass number. The solid black lines show the evolution of the ML estimates, the solid grey lines show the true value, and the dashed grey lines show the mean parameter estimates from the particle swarm after the final pass.

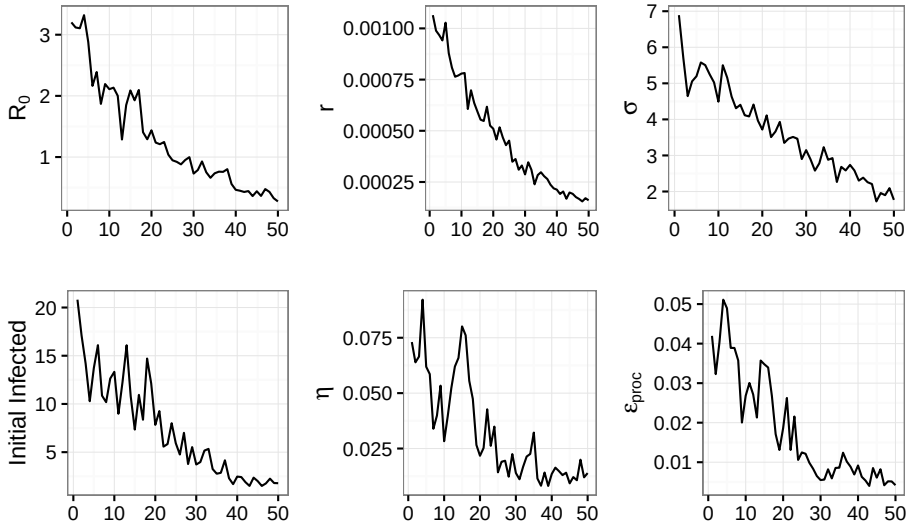

Figure 4.7: The horizontal axis shows the IF2 pass number and the solid black lines show the evolution of the standard deviations of the particle swarm values.

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.

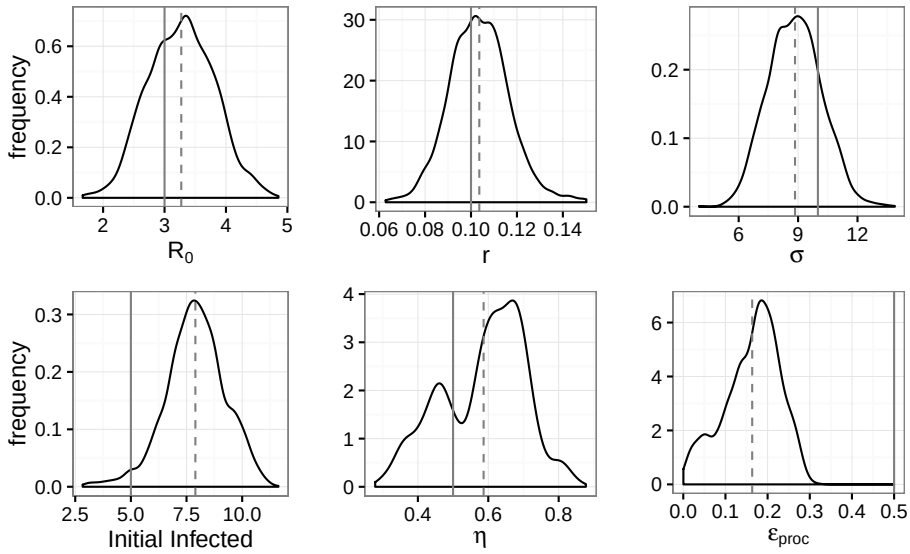

Figure 4.8: As before, the solid grey lines show the true parameter values and the dashed grey lines show the density means.

4.6 HMCMC Fitting

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

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

4.7 HMCMC Densities

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

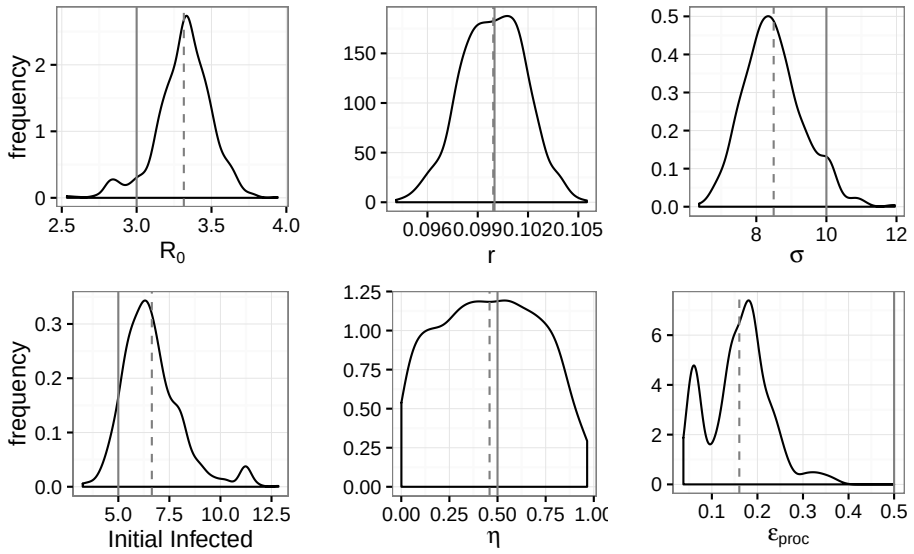

Figure 4.9: As before, the solid grey lines show the true parameter values and the dashed grey lines show the density means.

4.8 HMCMC 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 HMCMC samples.

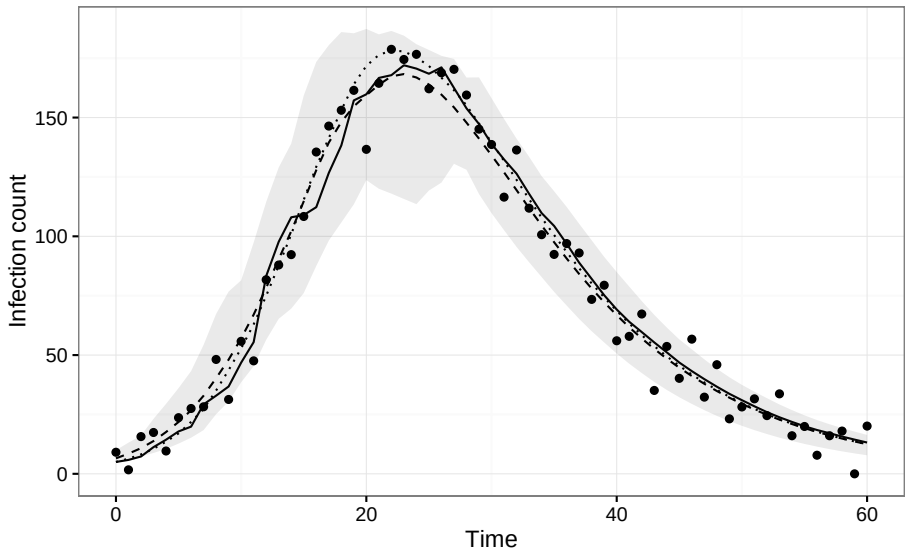

Figure 4.10: Result from 100 HMCMC bootstrap trajectories. The solid line shows the true states, the dots show the data, the dotted line shows the average system behaviour, the dashed line shows the bootstrap mean, and the grey ribbon shows the centre 95th quantile of the bootstrap trajectories.

4.9 Multi-trajectory Parameter Estimation

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

Figure [4.11] shows the results of the mean parameter fits from IF2 and HMCMC 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 HMCMC 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 HMCMC respectively, representing a 5.7x speedup for IF2 over HMCMC. While IF2 may be able to fit the model to data faster than HMCMC, we are obtaining less information; this will become important in the next section. Further, the results in Figure [4.12] show that while the running time for IF2 is relatively fixed, the times for HMCMC are anything but, showing a wide spread of potential times.

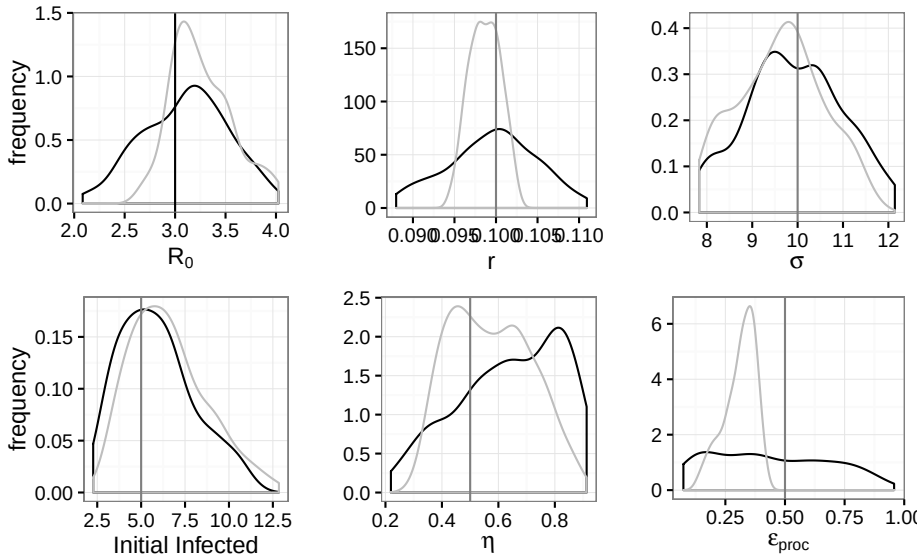

Figure 4.11: IF2 point estimate densities are shown in black and HMCMC point estimate densities are shown in grey. The vertical black lines show the true parameter values.

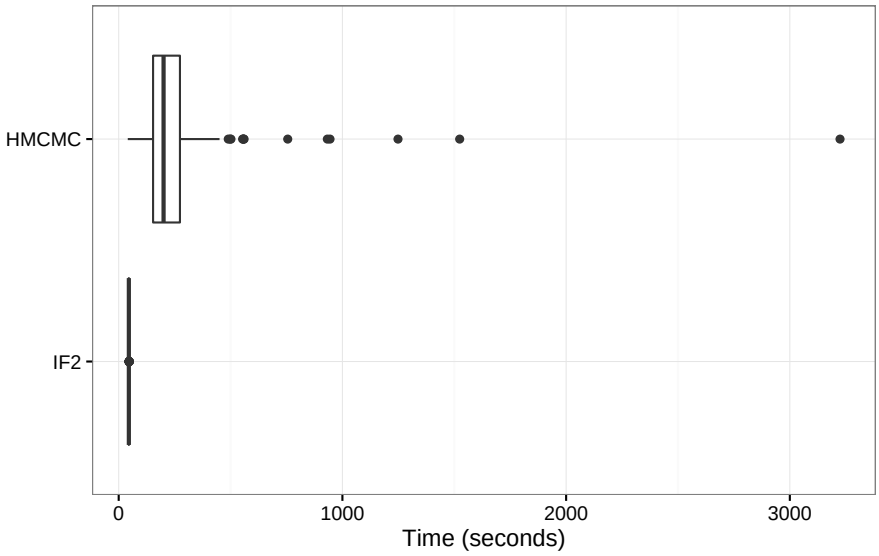

Figure 4.12: Fitting times for IF2 and HMCMC, in seconds. The centre box in each plot shows the centre 50th quantile, with the bold centre line showing the median.

Chapter 5

Forecasting Frameworks

5.1 Data Setup

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

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 HMCMC only the data points and have it reconstruct the entirety of the true system states.

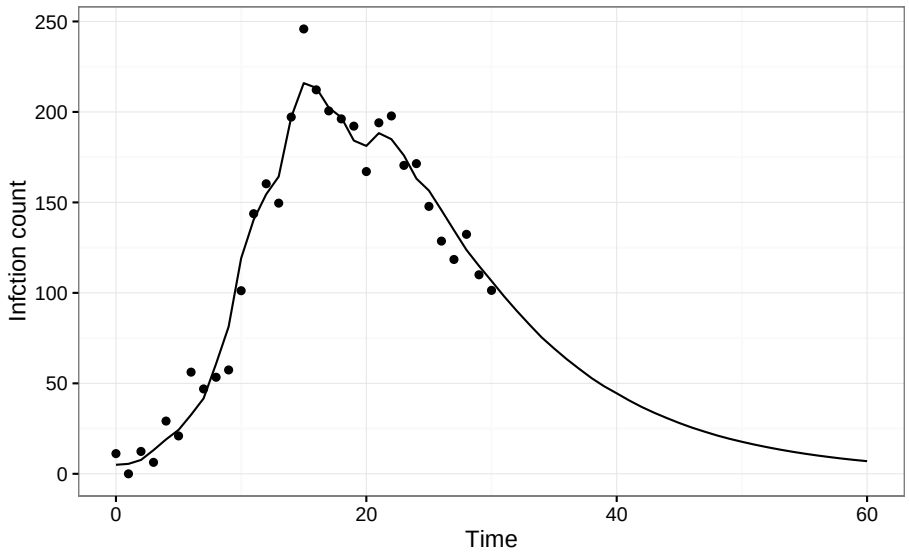

Figure 5.1: Infection count data truncated at $T = 30$. The solid line shows the true underlying system states, and the dots show those states with added observation noise. Parameters used were $R_0 = 3.0$, $r = 0.1$, $\eta = .05$, $\sigma_{proc} = 0.5$, and additive observation noise was drawn from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

5.2 IF2

For IF2, we will take advantage of the fact that the particle filter will produce state estimates for every datum in the time series given to it, as well as producing 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 he 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.

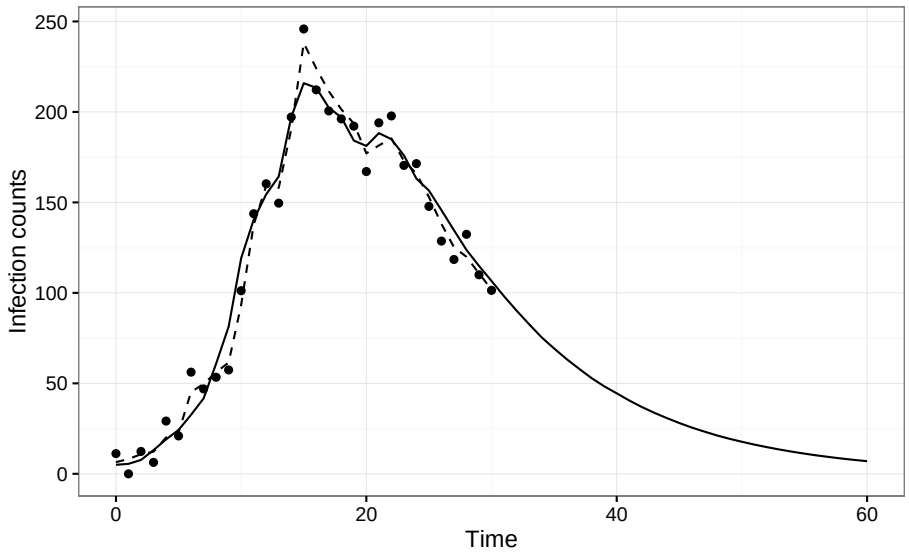

Figure 5.2: Infection count data truncated at $T = 30$ from Figure [5.1]. The dashed line shows IF2's attempt to reconstruct the true underlying state from the observed data points.

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.

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 SSE

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

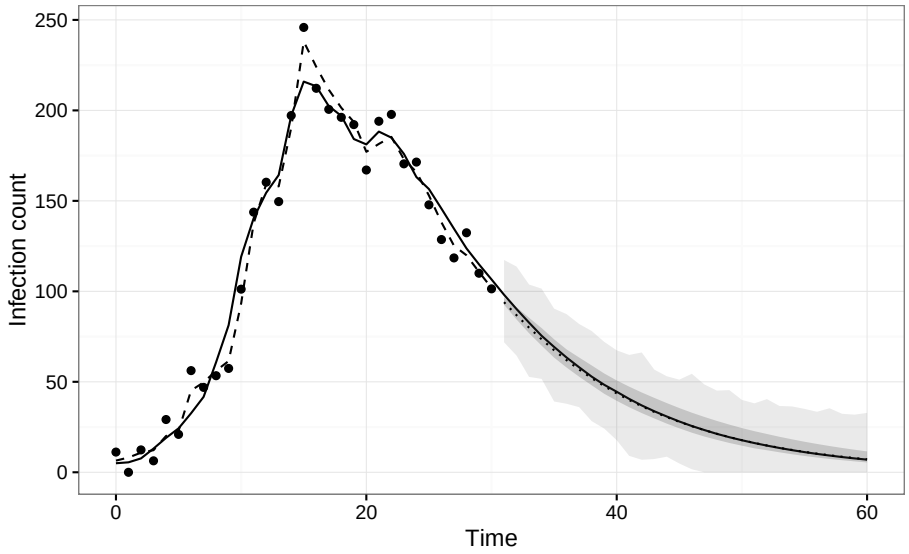

Figure 5.3: Forecast produced by the IF2 / parametric bootstrapping framework. The dotted line shows the mean estimate of the forecasts, the dark grey ribbon shows the 95% confidence interval based on the 0.025 and 0.975 quantiles on the true state estimates, and the lighter grey ribbon shows the same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

and dividing that value by the number of values predicted to get the average squared error per point. For the data in Figure [5.3] the value was $\overline{SSE} = 1.67$. 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 HMCMC

For HMCMC 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 HMCMC forecasting framework as applied to the data from Figure [5.1].

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

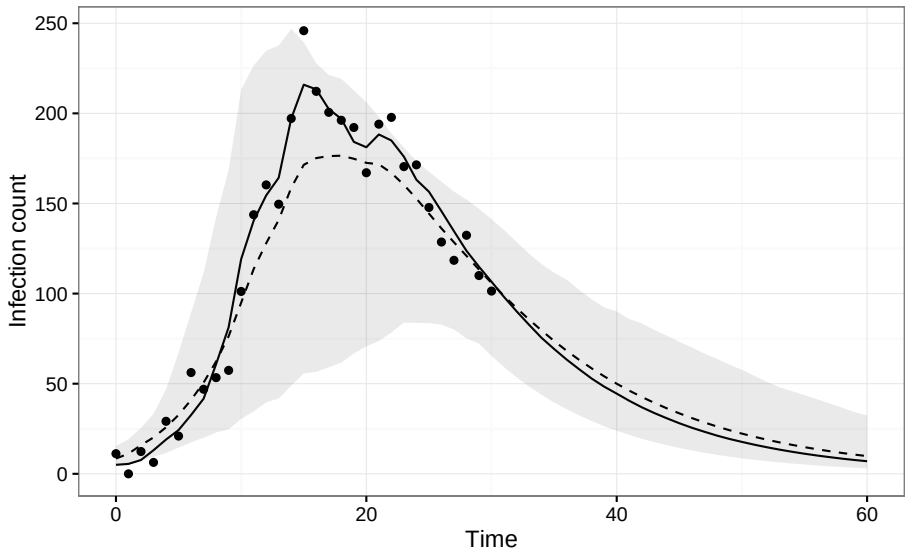

Figure 5.4: Forecast produced by the HMCMC / bootstrapping framework with $M = 200$ trajectories. The dotted line shows the mean estimate of the forecasts, and the grey ribbon shows the 95% confidence interval on the estimated true states as described in Figure [5.3].

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

Since the parametric bootstrapping approach used by IF2 requires a significant number of additional fits, its computational cost is significantly higher than the simpler bootstrapping approach used by the HMCMC framework, about 35.5x as expensive. However the now much longer running time can somewhat alleviated by parallelizing the parametric bootstrapping process; as each of the parametric bootstrap fittings in entirely independent, this can be done without a great deal of additional effort.

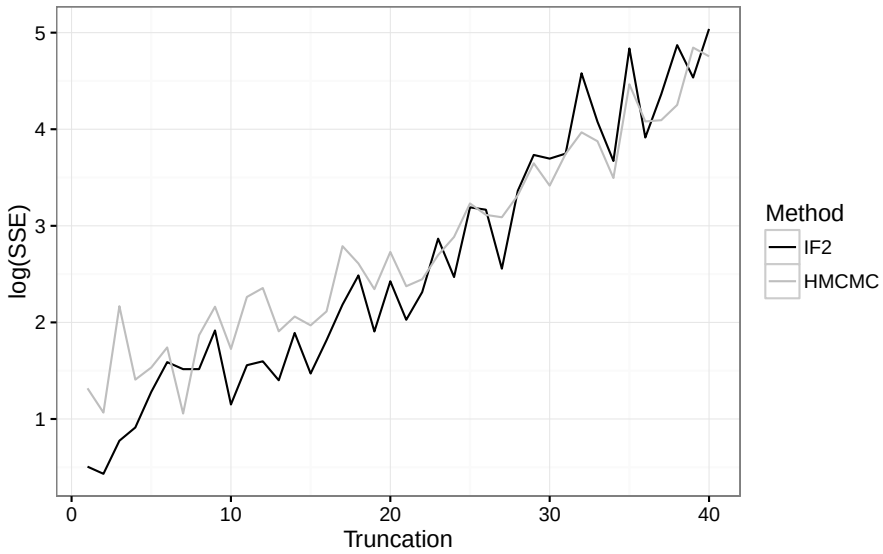

Figure 5.5: Error growth as a function of data truncation amount. Both methods used 200 bootstrap trajectories. Note that the y-axis shows the natural log of the averaged SSE, not the total SSE.

The code 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 [35][36][18][13] 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 HMCMC 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 HMCMC is to generate data from a SIRS model with a seasonal component, and have all methods operate on the resulting time series.

The basic skeleton of the SIRS model is similar to the stochastic SIR model described previously, with one small addition. The deterministic ODE component of the model is as follows.

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

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

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

This function oscillates between 1 and e^{-4} (close to 0) and is meant to represent transmission damping during the off-season, for example summer for influenza. Further, it displays flatter troughs and sharper peaks to exaggerate its effect in peak season.

As before, β is allowed to walk restricted by a geometric mean, described by

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

Figure [6.1] shows the SIRS model simulated for the equivalent of 5 years (260 weeks) and adding noise drawn from $\mathcal{N}(0, \sigma)$.

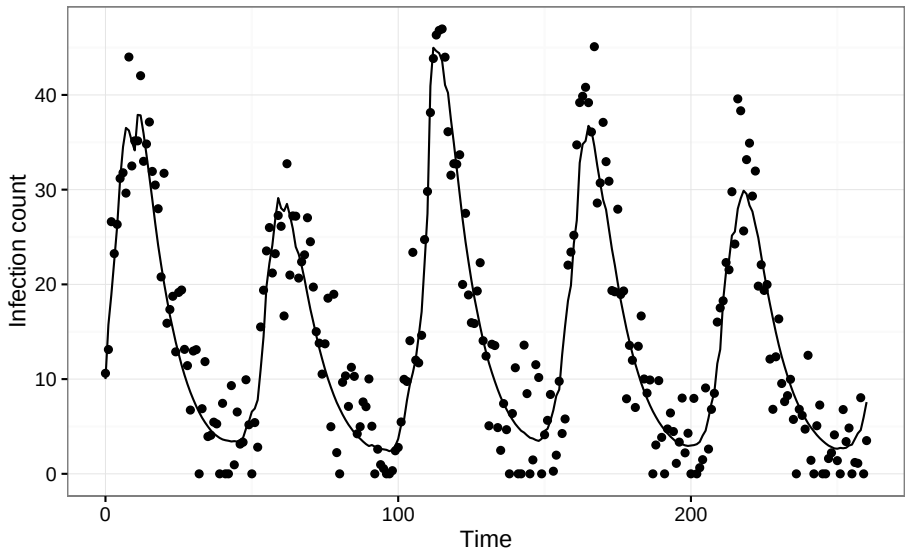

Figure 6.1: Five cycles generated by the SIRS function. The solid line the the true number of cases, dots show case counts with added observation noise. The parameter values were $R_0 = 3.0$, $\gamma = 0.1$, $\eta = 1$, $\sigma = 5$, and 10 initial cases.

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

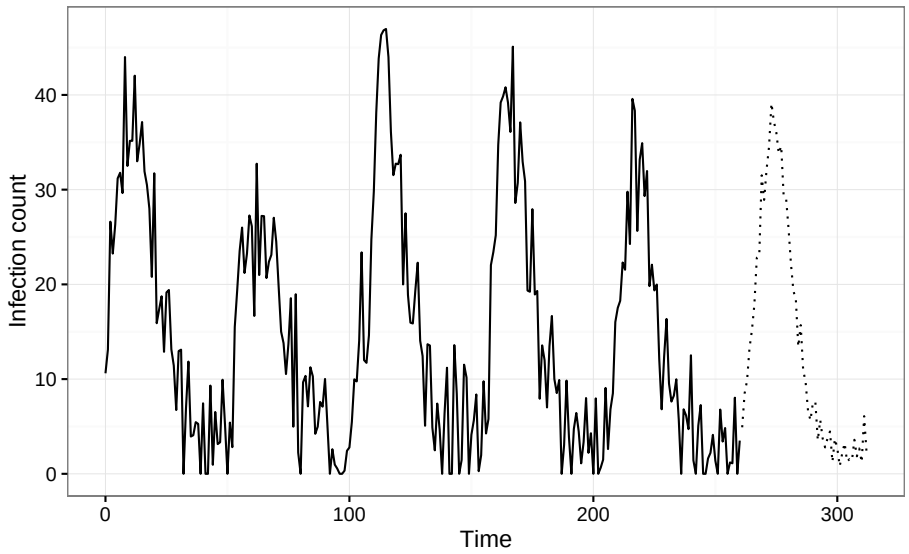

Figure 6.2: S-map applied to the data from the previous figure. The solid line shows the infection counts with observation noise from the previous plot, and the dotted line is the S-map forecast. Parameters chosen were $E = 14$ and $\theta = 3$.

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 [13]. 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 HMCMC.

To do this we generated a series of artificial time series of length 260 meant to represent 5 years of weekly incidence counts and used each method to forecast up to 2 years into the future. Our goal here was to determine how forecast error changed with forecast length.

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 HMCMC and the S-map for the majority of forecast lengths, with the

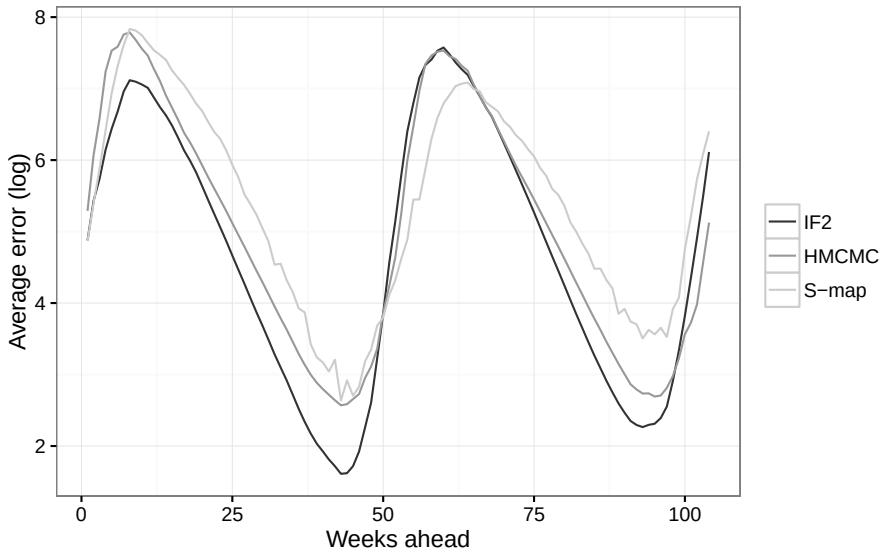

Figure 6.3: Error as a function of forecast length.

S-map 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 HMCMC, 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 HMCMC it is about $9,200$. This is a speed-up of over 316,000x compared to IF2 and over 61,800x compared to HMCMC.

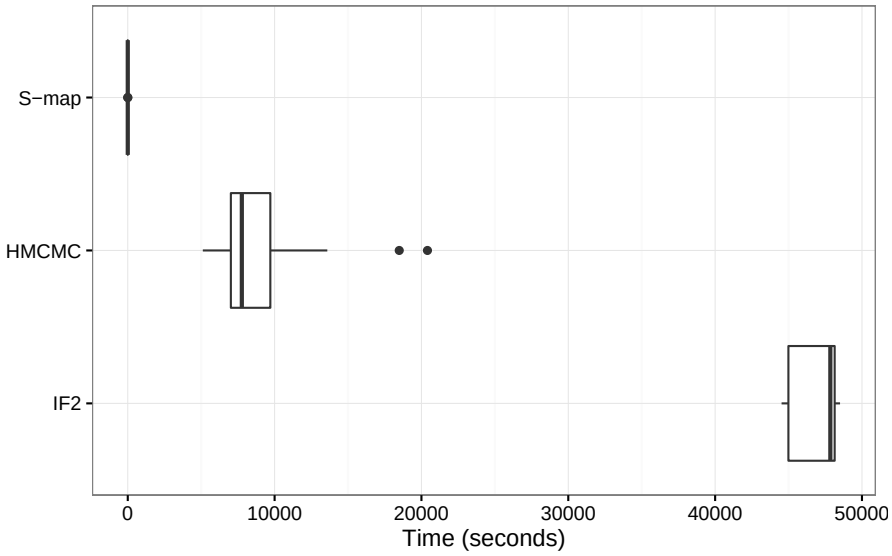

Figure 6.4: Runtimes for producing SIRS forecasts. The box shows the middle 50th quantile, the bold line is the median, and the dots are outliers. Note that these are not "true" outliers, simply ones outside a ranges based on the interquartile range.

Chapter 7

Spatial Epidemics

7.1 Spatial SIR

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

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

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

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

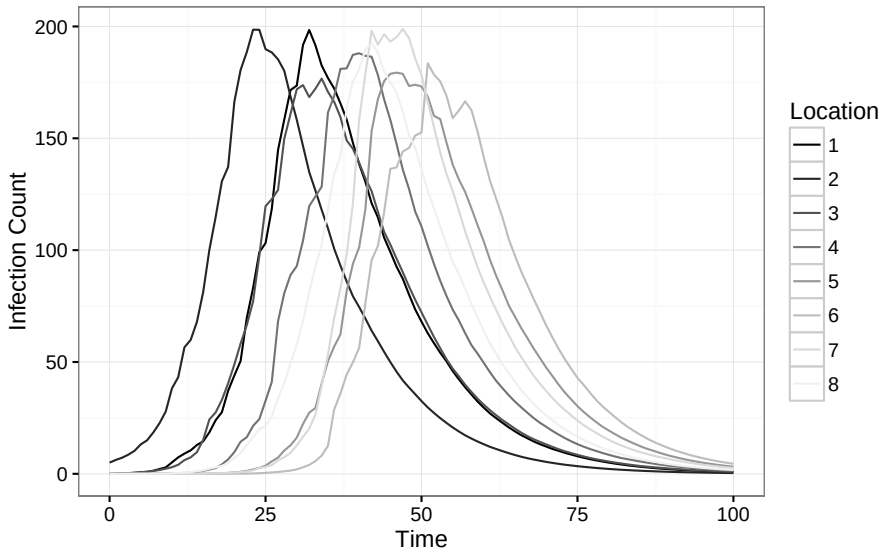

Figure 7.1: Evolution of a spatial epidemic in a ring topology. The outbreak was started with 5 cases in Location 2. Parameters were $R_0 = 3.0$, $\gamma = 0.1$, $\eta = 0.5$, $\sigma_{err} = 0.5$, and $\phi = 0.5$.

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

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 10 locations is connected to exactly two neighbours (i.e. location 1 is connected to locations N and 2, location 2 is connected to locations 1 and 3, etc.), and we start each location with completely susceptible populations except for a handful of infected individuals in one of the locations, we obtain a plot of the outbreak progression in Figure [7.1].

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

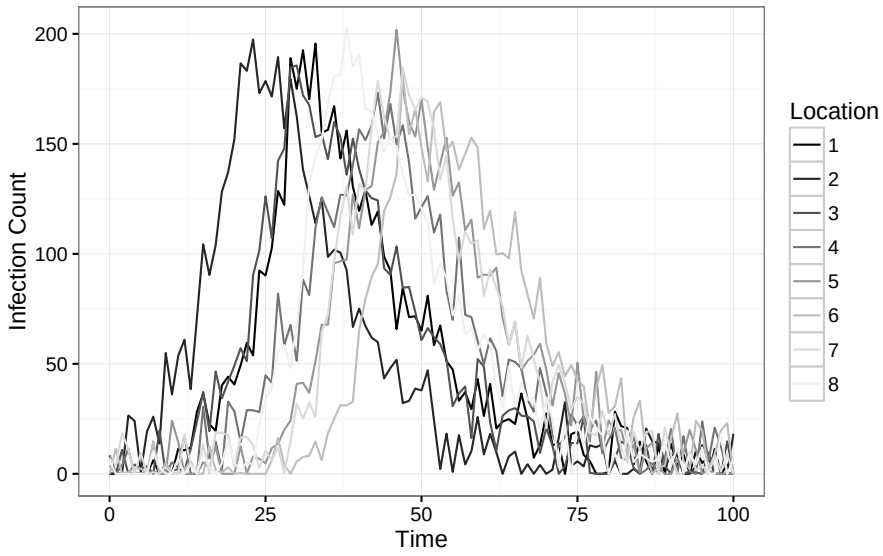

Figure 7.2: Evolution of a spatial epidemic as in Figure [7.1], with added observation noise drawn from $\mathcal{N}(0, 10)$.

7.2 Dewdrop Regression

Dewdrop regression [18] 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 HMCMC, we generated 20 independent spatial data sets up to time $T = 50$ weeks in each of $L = 10$ locations and forecasted 10 weeks into the future. Forecasts were compared to that of the true model evolution, and the average SSE for each week ahead in the forecast were computed. The number of bootstrapping trajectories used by IF2 and HMCMC was reduced from 200 to 50 to curtail running times.

The results are shown in Figure [7.3].

The results show a clear delineation in forecast fidelity between methods. IF2 maintains an advantage regardless of how long the forecast produced. Interestingly, Dewdrop Regression with S-mapping performs almost as well as IF2, and outperforms HMCMC. HMCMC lags behind both methods by a healthy margin.

If we examine the runtimes for each forecast framework, we obtain the data in Figure [7.4].

As before, the S-map with Dewdrop Regression runs faster than the other two methods with a huge margin. It is again hard to see exactly how large the margin is from the figure due to the scale, but we can examine the average values: the average running time for S-mapping with Dewdrop Regression was about 249 seconds, whereas the average times for IF2 and HMCMC were about 2.90×10^4 and 3.88×10^4, respectively. This is a speed-up of just over 116x over IF2 and 156x over HMCMC.

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

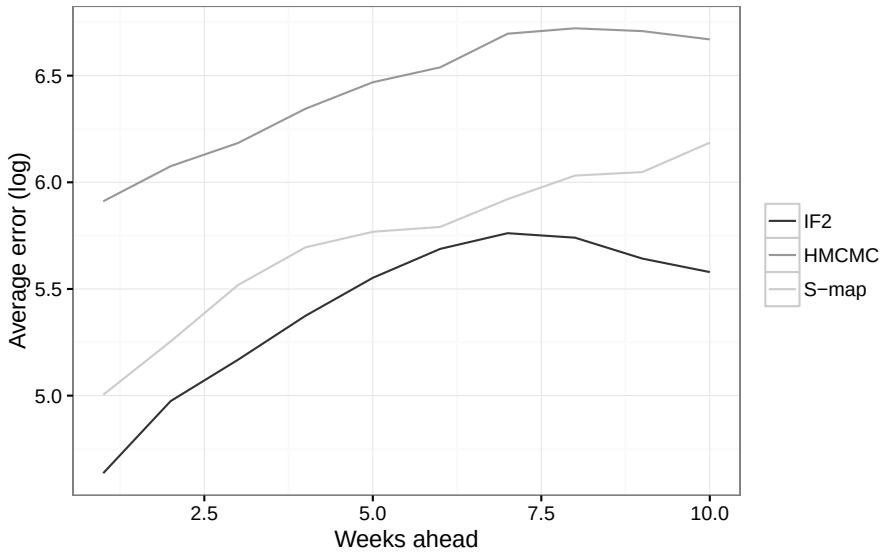

Figure 7.3: Average SSE (log scale) across each location and all trials as a function of the number of weeks ahead in the forecast.

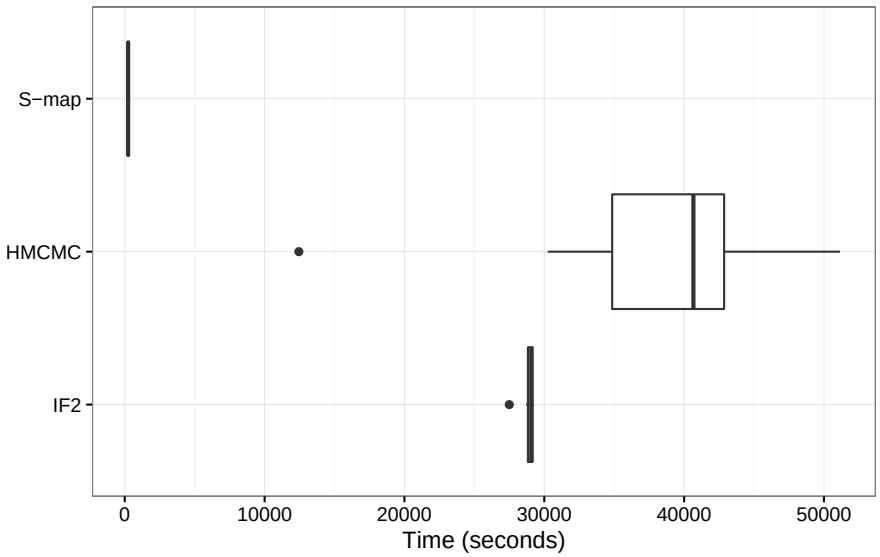

Figure 7.4: Runtimes for producing spatial SIR forecasts. The box shows the middle 50th quantile, the bold line is the median, and the dots are outliers.

Chapter 8

Discussion and Future Directions

Right off the bat, we can see that the IF2 / parametric bootstrapping framework produces great results. This framework consistently out-performs both the HMCMC framework and S-mapping by itself or with Dewdrop Regression. This isn't to say that the results produced by the other methods are poor, but rather that the ones produced by IF2 are noticibly 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. And in the spatial scenario it performs almost as well as IF2, and much better than HMCMC. 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 the other methods could even be set up.

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 [38].

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

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

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

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

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

Hamiltonian MCMC is cursed with high dependence between iterations. While HMCMC has an advantage over "vanilla" MCMC formulations in terms of efficiency of step acceptance and ease of exploration of the parameter per number of samples, each sample still depends entirely on the preceding one, and at a conceptual level the construction of a Markov Chain *requires* iterative dependence. We cannot simply take an accepted step, compute several proposed steps accept/reject them independently – doing so would break the chain construction and could potentially bias our posterior estimate to boot. We can, however, process multiple chains simultaneously and merge the resulting samples. If the required number of samples for a problem were large and

the required burn-in time were low, this methods could prove effective. However, the parallel burn-in sampling is still inefficient as it is a duplication of effort with limited pay-off – in the sense that the saved sample to discarded burn-in sample ratio would not be as efficient as running a single long chain. Thus while parallelism via multiple independent chains would help with a reduction in wall clock running times, it would result in an *increase* in total computer time.

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

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

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

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

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

A future offshoot of this project would be a good parallel implementation of both the IF2 fitting process and the parametric bootstrapping framework. And ideal platform for this work would be NVIDIA's Compute Unified Device Architecture (CUDA) Graphics Processing Unit (GPU) computing framework. While a CUDA implementa-

tion of a spatial epidemic IF2 parameter fitting algorithm was implemented, it lacked a good front-end implementation, R integration, and a parametric bootstrapping framework and so was not included in the main results of this paper. The code, however, as well as some preliminary results, are included in the appendices.

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

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

8.2 IF2, Bootstrapping, and Forecasting Methodology

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

A recent paper utilising IF2 for forecasting [22] 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 results such as IF2, HMCMC, and parametric bootstrapping in order to produce the cleanest forecast possible. And this only represents three data points in a larger picture. There are a wide variety of methods that are similar but not identical to methods explored here, each with their own positive and negative attributes, their own advantages and disadvantages, and that are ultimately likely to fill out our spectrum of methods more completely. Thus future work should focus on attempting further direct comparison across a wider swath of techniques, and implementing those techniques in a parallel fashion to take advantage of the current and future landscape of high-performance computing.

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

[13] 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.

[14] 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.

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

[16] 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.

[17] 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/.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

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

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.

[33] 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.

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

[35] 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.

[36] 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.

[37] 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.

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

[39] 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.

[40] 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
## McMaster University
## 2016

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

SIR ← function(Time, State, Pars) {

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

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

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

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

    })

}

```

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

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

odeout ← ode(y_ini, times, SIR, pars)

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

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

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

sPw ← 7
datlen ← (T-1)*7 + 1

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

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

rstan_options(auto_write = TRUE)
options(mc.cores = parallel::detectCores())
stan_options ← list(   chains = 4,    # number of chains
                       iter   = 2000, # iterations per chain
                       warmup = 1000, # warmup interations
                       thin   = 2)   # thinning number
fit ← stan(file    = "d_sirode_euler.stan",
           data    = sir_data,
           chains  = stan_options$chains,
           iter    = stan_options$iter,
           warmup  = stan_options$warmup,
           thin    = stan_options$thin )

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

```

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

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

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

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

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

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

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

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

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

```
            theme_bw()

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

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

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

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

A.2 Full Stan code

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

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

data {

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

}

parameters {

    real <lower=0, upper=10>    R0;     // R0
    real <lower=0, upper=10>    r;      // recovery rate
    real <lower=0, upper=20>    sigma;  // observation error
    real <lower=0, upper=500>   y0[3];  // initial conditions
```

```
1  20
2  21 }
3  22
4  23 model {
5  24
6  25     real S[T];
7  26     real I[T];
8  27     real R[T];
9  28
10 29     S[1] <- y0[1];
11 30     I[1] <- y0[2];
12 31     R[1] <- y0[3];
13 32
14 33     y[1] ~ normal(y0[2], sigma);
15 34
16 35     for (t in 2:T) {
17 36
18 37         S[t] <- S[t-1] + h*( - S[t-1]*I[t-1]*R0*r/N );
19 38         I[t] <- I[t-1] + h*( S[t-1]*I[t-1]*R0*r/N  - I[t-1]*r );
20 39         R[t] <- R[t-1] + h*( I[t-1]*r );
21 40
22 41         if (y[t] > 0) {
23 42             y[t] ~ normal( I[t], sigma );
24 43         }
25 44
26 45     }
27 46
28 47     y0[1] ~ normal(N - y[1], sigma);
29 48     y0[2] ~ normal(y[1], sigma);
30 49
31 50     R0      ~ lognormal(1,1);
32 51     r       ~ lognormal(1,1);
33 52     sigma   ~ lognormal(1,1);
34 53
35 54 }
36
```

Appendix B

Iterated Filtering

B.1 Full R code

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

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

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

SIR ← function(Time, State, Pars) {

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

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

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

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

    })

}

T     ← 100
N     ← 500
```

```
sigma   ← 10
i_infec ← 5

## Generate true trajecory and synthetic data
##

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

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

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

set.seed(1001)

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

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

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

## Rcpp stuff
##

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

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

## Parameter density kernels
##

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

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

```

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

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

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

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

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

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

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

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

B.2 Full C++ code

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

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

```

```
    */

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

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

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

#include <Rcpp.h>
using namespace Rcpp;


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

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


int timeval_subtract (double *result, struct timeval *x, struct
   timeval *y);
int check_double(double x,double y);
```

```
54 void exp_euler_SIR(double h, double t0, double tn, int N, Particle *
       particle);
55 void copyParticle(Particle * dst, Particle * src);
56 void perturbParticles(Particle * particles, int N, int NP, int
       passnum, double coolrate);
57 bool isCollapsed(Particle * particles, int NP);
58 void particleDiagnostics(ParticleInfo * partInfo, Particle *
       particles, int NP);
59 NumericMatrix if2(NumericVector * data, int T, int N);
60 double randu();
61 double randn();
62
63 // [[Rcpp::export]]
64 NumericMatrix if2(NumericVector data, int T, int N) {
65
66     int     NP          = 2500;
67     int     nPasses     = 50;
68     double  coolrate    = 0.975;
69
70     int     i_infec     = I0;
71
72     NumericMatrix paramdata(NP, 6);
73
74     srand(time(NULL));      // Seed PRNG with system time
75
76     double w[NP];           // particle weights
77
78     Particle particles[NP];     // particle estimates for current
           step
79     Particle particles_old[NP]; // intermediate particle states for
           resampling
80
81     printf("Initializing particle states\n");
82
83     // initialize particle parameter states (seeding)
84     for (int n = 0; n < NP; n++) {
85
86         double R0can, rcan, sigmacan, Iinitcan;
87
88         do {
89             R0can = R0true + R0true*randn();
90         } while (R0can < 0);
91         particles[n].R0 = R0can;
92
93         do {
94             rcan = rtrue + rtrue*randn();
95         } while (rcan < 0);
96         particles[n].r = rcan;
97
98         do {
99             sigmacan = merr + merr*randn();
```

```
        } while (sigmacan < 0);
        particles[n].sigma = sigmacan;

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

    }

    // START PASSES THROUGH DATA

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


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

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

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

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

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

        }

        // between-pass perturbations

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

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

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

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

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

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

```
150
151             }
152
153             // cumulative sum
154             for (int n = 1; n < NP; n++) {
155                 w[n] += w[n-1];
156             }
157
158             // save particle states to resample from
159             for (int n = 0; n < NP; n++){
160                 copyParticle(&particles_old[n], &particles[n]);
161             }
162
163             // resampling
164             for (int n = 0; n < NP; n++) {
165
166                 double w_r = randu() * w[NP-1];
167                 int i = 0;
168                 while (w_r > w[i]) {
169                     i++;
170                 }
171
172                 // i is now the index to copy state from
173                 copyParticle(&particles[n], &particles_old[i]);
174
175             }
176
177         }
178
179     }
180
181     ParticleInfo pInfo;
182     particleDiagnostics(&pInfo, particles, NP);
183
184     printf("Parameter results (mean | sd)\n");
185     printf("-----------------------------\n");
186     printf("R0        %f %f\n", pInfo.R0mean, pInfo.R0sd);
187     printf("r         %f %f\n", pInfo.rmean, pInfo.rsd);
188     printf("sigma     %f %f\n", pInfo.sigmamean, pInfo.sigmasd);
189     printf("S_init  %f %f\n", pInfo.Sinitmean, pInfo.Sinitsd);
190     printf("I_init    %f %f\n", pInfo.Iinitmean, pInfo.Iinitsd);
191     printf("R_init    %f %f\n", pInfo.Rinitmean, pInfo.Rinitsd);
192
193     printf("\n");
194
195
196
197     // Get particle results to pass back to R
198
199     for (int n = 0; n < NP; n++) {
200
```

```
1  201         paramdata(n, 0) = particles[n].R0;
2  202         paramdata(n, 1) = particles[n].r;
3  203         paramdata(n, 2) = particles[n].sigma;
4  204         paramdata(n, 3) = particles[n].Sinit;
5  205         paramdata(n, 4) = particles[n].Iinit;
6  206         paramdata(n, 5) = particles[n].Rinit;
7  207
8  208     }
9  209
10 210     return paramdata;
11 211
12 212 }
13 213
14 214
15 215 /*  Use the Explicit Euler integration scheme to integrate SIR model
16        forward in time
17 216     double h    - time step size
18 217     double t0   - start time
19 218     double tn   - stop time
20 219     double * y  - current system state; a three-component vector
21                      representing [S I R], susceptible-infected-recovered
22 220
23 221     */
24 222 void exp_euler_SIR(double h, double t0, double tn, int N, Particle *
25        particle) {
26 223
27 224     int num_steps = floor( (tn-t0) / h );
28 225
29 226     double S = particle->S;
30 227     double I = particle->I;
31 228     double R = particle->R;
32 229
33 230     double R0   = particle->R0;
34 231     double r    = particle->r;
35 232     double B    = R0 * r / N;
36 233
37 234     for(int i = 0; i < num_steps; i++) {
38 235         // get derivatives
39 236         double dS = - B*S*I;
40 237         double dI = B*S*I - r*I;
41 238         double dR = r*I;
42 239         // step forward by h
43 240         S += h*dS;
44 241         I += h*dI;
45 242         R += h*dR;
46 243     }
47 244
48 245     particle->S = S;
49 246     particle->I = I;
50 247     particle->R = R;
51 248
```

```
}


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

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

    double coolcoef = pow(coolrate, passnum);

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

    double R0can, rcan, sigmacan, Iinitcan;

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

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

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

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

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

    }

}


/*  Convinience function for particle resampling process

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

```
298
299     dst->R0     = src->R0;
300     dst->r      = src->r;
301     dst->sigma  = src->sigma;
302     dst->S      = src->S;
303     dst->I      = src->I;
304     dst->R      = src->R;
305     dst->Sinit  = src->Sinit;
306     dst->Iinit  = src->Iinit;
307     dst->Rinit  = src->Rinit;
308
309 }
310
311
312 /*  Checks to see if particles are collapsed
313     This is done by checking if the standard deviations between the
        particles' parameter
314     values are significantly close to one another. Spread threshold
        may need to be tuned.
315
316     */
317 bool isCollapsed(Particle * particles, int NP) {
318
319     bool retVal;
320
321     double R0mean = 0, rmean = 0, sigmamean = 0, Sinitmean = 0,
        Iinitmean = 0, Rinitmean = 0;
322
323     // means
324
325     for (int n = 0; n < NP; n++) {
326
327         R0mean      += particles[n].R0;
328         rmean       += particles[n].r;
329         sigmamean   += particles[n].sigma;
330         Sinitmean   += particles[n].Sinit;
331         Iinitmean   += particles[n].Iinit;
332         Rinitmean   += particles[n].Rinit;
333
334     }
335
336     R0mean      /= NP;
337     rmean       /= NP;
338     sigmamean   /= NP;
339     Sinitmean   /= NP;
340     Iinitmean   /= NP;
341     Rinitmean   /= NP;
342
343     double  R0sd = 0, rsd = 0, sigmasd = 0, Sinitsd = 0, Iinitsd = 0,
         Rinitsd = 0;
344
```

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

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

```
434     partInfo->Sinitmean = Sinitmean;
435     partInfo->Sinitsd   = Sinitsd;
436     partInfo->Iinitmean = Iinitmean;
437     partInfo->Iinitsd   = Iinitsd;
438     partInfo->Rinitmean = Rinitmean;
439     partInfo->Rinitsd   = Rinitsd;
440
441 }
442
443 double randu() {
444
445     return (double) rand() / (double) RAND_MAX;
446
447 }
448
449
450 /*  Return a normally distributed random number with mean 0 and
        standard deviation 1
451     Uses the polar form of the Box-Muller transformation
452     From http://www.design.caltech.edu/erik/Misc/Gaussian.html
453     */
454 double randn() {
455
456     double x1, x2, w, y1;
457
458     do {
459         x1 = 2.0 * randu() - 1.0;
460         x2 = 2.0 * randu() - 1.0;
461         w = x1 * x1 + x2 * x2;
462     } while ( w >= 1.0 );
463
464     w = sqrt( (-2.0 * log( w ) ) / w );
465     y1 = x1 * w;
466
467     return y1;
468
469 }
```

Appendix C

Parameter Fitting

Appendix D

Forecasting Frameworks

D.1 IF2 Parametric Bootstrapping Function

The parametric bootstrapping machinery used to produce forecasts.

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

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

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

  source(stoc_sir_file)

  if (nTrials < 2)
    ntrials ← 2

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

```
1  27  names(statemeans_parent) ← c("S","I","R","B")
2  28
3  29
4  30  ## use parametric bootstrapping to generate forcasts
5  31  ##
6  32  trajectories ← foreach( i = 1:nTrials, .combine = rbind, .packages
7         = "Rcpp") %dopar% {
8  33
9  34    source(stoc_sir_file)
10 35
11 36    ## draw new data
12 37    ##
13 38
14 39    pars ← with( as.list(parmeans_parent),
15 40                 c(R0 = R0,
16 41                 r = r,
17 42                 N = N,
18 43                 eta = eta,
19 44                 berr = berr) )
20 45
21 46    init_cond ← with( as.list(parmeans_parent),
22 47                      c(S = Sinit,
23 48                        I = Iinit,
24 49                        R = Rinit) )
25 50
26 51    # generate trajectory
27 52    sdeout ← StocSIR(init_cond, pars, Tlim + 1, steps)
28 53    colnames(sdeout) ← c('S','I','R','B')
29 54
30 55    # add noise
31 56    counts_raw ← sdeout[,'I'] + rnorm(dim(sdeout)[1], 0, parmeans_
32           parent[['sigma']])
33 57      counts     ← ifelse(counts_raw < 0, 0, counts_raw)
34 58
35 59      ## refit using new data
36 60      ##
37 61
38 62      rm(if2) # because stupid things get done in packages
39 63      sourceCpp(if2file)
40 64      if2time ← system.time( if2data ← if2(counts, Tlim+1, N, NP,
41           nPasses, coolrate) )
42 65
43 66      paramdata ← data.frame( if2data$paramdata )
44 67    names(paramdata) ← c("R0", "r", "sigma", "eta", "berr", "Sinit",
45         "Iinit", "Rinit")
46 68    parmeans ← colMeans(paramdata)
47 69    names(parmeans) ← c("R0", "r", "sigma", "eta", "berr", "Sinit", "
48         Iinit", "Rinit")
49 70
50 71    ## generate the rest of the trajectory
51 72    ##
```

```

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

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

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


  }

  return(trajectories)

}
```

D.2 RStan Forward Simulator

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

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

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

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

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

  B0 ← R0 * r / N
  B ← B0
```

```
1  17
2  18    out[1,] ← c(S,I,R,B)
3  19
4  20    h ← 1 / steps
5  21
6  22    for ( i in 1:(T*steps) ) {
7  23
8  24        if (i <= bveclim) {
9  25          B ← exp( log(B) + eta*(log(B0) - log(B)) + berrvec[i])
10 26        } else {
11 27            B ← exp( log(B) + eta*(log(B0) - log(B)) + rnorm(1, 0, berr
12                   ))
13 28        }
14 29
15 30      BSI ← B*S*I
16 31      rI ← r*I
17 32
18 33      dS ← -BSI
19 34      dI ← BSI - rI
20 35      dR ← rI
21 36
22 37      S ← S + h*dS  #newInf
23 38      I ← I + h*dI  #newInf - h*dR
24 39      R ← R + h*dR  #h*dR
25 40
26 41      if (i %% steps == 0)
27 42        out[i/steps+1,] ← c(S,I,R,B)
28 43
29 44    }
30 45
31 46    return(out)
32 47
33 48  }
34
```

Appendix E

S-map and SIRS

E.1 SIRS R Function Code

R code to simulate the outlines SIRS function.

```
StocSIRS ← function(y, pars, T, steps) {

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

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

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

  B0 ← R0 * r / N
  B ← B0

  out[1,] ← c(S,I,R,B)

  h ← 1 / steps

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

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

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

```

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

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

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

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

  }

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

}

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

E.2 SMAP Code

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

```
library(pracma)

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

    # construct library
    tseries ← as.vector(data)
    liblen  ← length(tseries) - E + 1 - stepsAhead
```

```
    lib     <- matrix(NA, liblen, E)

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

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

    #allPredictees <- matrix(NA, stepsAhead, E)

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

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

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

        # construct A, B

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

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

        # solve system for C

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

        # get prediction

        predsum <- sum(C * c(1,predictee))

        # save

        predictions[i] <- predsum

        # next predictee

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

    }
```

```

    return(predictions)

}
```

E.3 SMAP Parameter Optimization Code

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

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

set.seed(1010)

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




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

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

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

## trial parameter values to check.options
```

```
##
Elist ← 1:20
thetalist ← 10*exp(-(seq(0,9.5,0.5)))
nTrials ← 100

ssemat ← matrix(NA, 20, 20)

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

    ssemean ← 0

    for (k in 1:nTrials) {

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

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

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

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

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

      ssemean ← ssemean + (sse / nTrials)

    }

    ssemat[i,j] ← ssemean


  }
}

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

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

```

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

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

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

print(Emin)
print(thetamin)

```

E.4 RStan SIRS Code

This code implements a periodic SIRS model in Rstan.

```
data {

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

}

parameters {

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

}

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

model {

    real S[T];
```

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

    real pi;
    real Bfac;

    pi ← 3.1415926535;

    B0 ← R0 * r / N;

    B[1] ← B0;

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

    for (t in 2:T) {

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

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

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

    }

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

}
```

E.5 IF2 SIRS Code

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

```
/*  Author: Dexter Barrows
```

```
    Github: dbarrows.github.io

    */

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

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

#define Treal       100         // time to simulate over
#define R0true      3.0         // infectiousness
#define rtrue       0.1         // recovery rate
#define retrue      0.05        // resusceptibility rate
#define Nreal       500.0       // population size
#define etatrue     0.5         // real drift attraction strength
#define berrtrue    0.5         // real beta drift noise
#define merr        5.0         // expected measurement error
#define I0          5.0         // Initial infected individuals

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

#include <Rcpp.h>
using namespace Rcpp;

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

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

```
    double Iinit;
    double Rinit;
};

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


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

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

    int npar = 9;

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

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

    double w[NP];           // particle weights

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

```
1          resampling
2  97
3  98      printf("Initializing particle states\n");
4  99
5  100     // initialize particle parameter states (seeding)
6  101     for (int n = 0; n < NP; n++) {
7  102
8  103         double R0can, rcan, recan, sigmacan, Iinitcan, etacan,
9                  berrcan;
10 104
11 105         do {
12 106             R0can = R0true + R0true*randn();
13 107         } while (R0can < 0);
14 108         particles[n].R0 = R0can;
15 109
16 110         do {
17 111             rcan = rtrue + rtrue*randn();
18 112         } while (rcan < 0);
19 113         particles[n].r = rcan;
20 114
21 115         do {
22 116             recan = retrue + retrue*randn();
23 117         } while (recan < 0);
24 118         particles[n].re = recan;
25 119
26 120         particles[n].B = (double) R0can * rcan / N;
27 121
28 122         do {
29 123             sigmacan = merr + merr*randn();
30 124         } while (sigmacan < 0);
31 125         particles[n].sigma = sigmacan;
32 126
33 127         do {
34 128             etacan = etatrue + PSC*etatrue*randn();
35 129         } while (etacan < 0 || etacan > 1);
36 130         particles[n].eta = etacan;
37 131
38 132         do {
39 133             berrcan = berrtrue + PSC*berrtrue*randn();
40 134         } while (berrcan < 0);
41 135         particles[n].berr = berrcan;
42 136
43 137         do {
44 138             Iinitcan = I0 + I0*randn();
45 139         } while (Iinitcan < 0 || N < Iinitcan);
46 140         particles[n].Sinit = N - Iinitcan;
47 141         particles[n].Iinit = Iinitcan;
48 142         particles[n].Rinit = 0.0;
49 143
50 144     }
51 145
```

```
146     // START PASSES THROUGH DATA
147
148     printf("Starting filter\n");
149     printf("---------------\n");
150     printf("Pass\n");
151
152
153     for (int pass = 0; pass < nPasses; pass++) {
154
155         printf("...%d / %d\n", pass, nPasses);
156
157         // reset particle system evolution states
158         for (int n = 0; n < NP; n++) {
159
160             particles[n].S = particles[n].Sinit;
161             particles[n].I = particles[n].Iinit;
162             particles[n].R = particles[n].Rinit;
163             particles[n].B = (double) particles[n].R0 * particles[n].
                    r / N;
164
165         }
166
167         if (pass == (nPasses-1)) {
168             State sMeans;
169             getStateMeans(&sMeans, particles, NP);
170             statemeans(0,0) = sMeans.S;
171             statemeans(0,1) = sMeans.I;
172             statemeans(0,2) = sMeans.R;
173         }
174
175         for (int t = 1; t < T; t++) {
176
177             // generate individual predictions and weight
178             for (int n = 0; n < NP; n++) {
179
180                 exp_euler_SIRS(1.0/7.0, (double) t-1, (double) t, N,
                        &particles[n]);
181
182                 double merr_par = particles[n].sigma;
183                 double y_diff   = data[t] - particles[n].I;
184
185                 w[n] = 1.0/(merr_par*sqrt(2.0*M_PI)) * exp( - y_diff*
                        y_diff / (2.0*merr_par*merr_par) );
186
187             }
188
189             // cumulative sum
190             for (int n = 1; n < NP; n++) {
191                 w[n] += w[n-1];
192             }
193
```

```
194             // save particle states to resample from
195             for (int n = 0; n < NP; n++){
196                 copyParticle(&particles_old[n], &particles[n]);
197             }
198 
199             // resampling
200             for (int n = 0; n < NP; n++) {
201 
202                 double w_r = randu() * w[NP-1];
203                 int i = 0;
204                 while (w_r > w[i]) {
205                     i++;
206                 }
207 
208                 // i is now the index to copy state from
209                 copyParticle(&particles[n], &particles_old[i]);
210 
211             }
212 
213             // between-iteration perturbations, not after last time
                    step
214             if (t < (T-1))
215                 perturbParticles(particles, N, NP, pass, coolrate);
216 
217             if (pass == (nPasses-1)) {
218                 State sMeans;
219                 getStateMeans(&sMeans, particles, NP);
220                 statemeans(t,0) = sMeans.S;
221                 statemeans(t,1) = sMeans.I;
222                 statemeans(t,2) = sMeans.R;
223             }
224 
225         }
226 
227         ParticleInfo pInfo;
228         particleDiagnostics(&pInfo, particles, NP);
229 
230         means(pass, 0) = pInfo.R0mean;
231         means(pass, 1) = pInfo.rmean;
232         means(pass, 2) = pInfo.remean;
233         means(pass, 3) = pInfo.sigmamean;
234         means(pass, 4) = pInfo.etamean;
235         means(pass, 5) = pInfo.berrmean;
236         means(pass, 6) = pInfo.Sinitmean;
237         means(pass, 7) = pInfo.Iinitmean;
238         means(pass, 8) = pInfo.Rinitmean;
239 
240         sds(pass, 0) = pInfo.R0sd;
241         sds(pass, 1) = pInfo.rsd;
242         sds(pass, 2) = pInfo.resd;
243         sds(pass, 3) = pInfo.sigmasd;
```

```
244         sds(pass, 4) = pInfo.etasd;
245         sds(pass, 5) = pInfo.berrsd;
246         sds(pass, 6) = pInfo.Sinitsd;
247         sds(pass, 7) = pInfo.Iinitsd;
248         sds(pass, 8) = pInfo.Rinitsd;
249 
250         // between-pass perturbations, not after last pass
251         if (pass < (nPasses + 1))
252             perturbParticles(particles, N, NP, pass, coolrate);
253 
254     }
255 
256     ParticleInfo pInfo;
257     particleDiagnostics(&pInfo, particles, NP);
258 
259     printf("Parameter results (mean | sd)\n");
260     printf("-----------------------------\n");
261     printf("R0        %f %f\n", pInfo.R0mean, pInfo.R0sd);
262     printf("r         %f %f\n", pInfo.rmean, pInfo.rsd);
263     printf("re        %f %f\n", pInfo.remean, pInfo.resd);
264     printf("sigma     %f %f\n", pInfo.sigmamean, pInfo.sigmasd);
265     printf("eta       %f %f\n", pInfo.etamean, pInfo.etasd);
266     printf("berr    %f %f\n", pInfo.berrmean, pInfo.berrsd);
267     printf("S_init  %f %f\n", pInfo.Sinitmean, pInfo.Sinitsd);
268     printf("I_init    %f %f\n", pInfo.Iinitmean, pInfo.Iinitsd);
269     printf("R_init    %f %f\n", pInfo.Rinitmean, pInfo.Rinitsd);
270 
271     printf("\n");
272 
273 
274 
275     // Get particle results to pass back to R
276 
277     for (int n = 0; n < NP; n++) {
278 
279         paramdata(n, 0) = particles[n].R0;
280         paramdata(n, 1) = particles[n].r;
281         paramdata(n, 2) = particles[n].re;
282         paramdata(n, 3) = particles[n].sigma;
283         paramdata(n, 4) = particles[n].eta;
284         paramdata(n, 5) = particles[n].berr;
285         paramdata(n, 6) = particles[n].Sinit;
286         paramdata(n, 7) = particles[n].Iinit;
287         paramdata(n, 8) = particles[n].Rinit;
288 
289     }
290 
291     for (int n = 0; n < NP; n++) {
292 
293         statedata(n, 0) = particles[n].S;
294         statedata(n, 1) = particles[n].I;
```

```
295         statedata(n, 2) = particles[n].R;
296         statedata(n, 3) = particles[n].B;
297 
298     }
299 
300 
301 
302     return Rcpp::List::create(  Rcpp::Named("paramdata") = paramdata,
303                                 Rcpp::Named("means") = means,
304                                 Rcpp::Named("statemeans") =
                                        statemeans,
305                                 Rcpp::Named("statedata") = statedata,
306                                 Rcpp::Named("sds") = sds);
307 
308 }
309 
310 
311 /*  Use the Explicit Euler integration scheme to integrate SIR model
        forward in time
312     double h    - time step size
313     double t0   - start time
314     double tn   - stop time
315     double * y  - current system state; a three-component vector
            representing [S I R], susceptible-infected-recovered
316 
317     */
318 void exp_euler_SIRS(double h, double t0, double tn, int N, Particle *
        particle) {
319 
320     int num_steps = floor( (tn-t0) / h );
321 
322     double S = particle->S;
323     double I = particle->I;
324     double R = particle->R;
325 
326     double R0   = particle->R0;
327     double r    = particle->r;
328     double re   = particle->re;
329     double B0   = R0 * r / N;
330     double eta  = particle->eta;
331     double berr  = particle->berr;
332 
333     double B = particle->B;
334 
335     for(int i = 0; i < num_steps; i++) {
336 
337         //double Bfac = 0.5 - 0.95*cos( (2.0*M_PI/365)*(t0*num_steps+
                i) )/2.0;
338         double Bfac = exp(2*cos((2*M_PI/365)*(t0*num_steps+i)) - 2);
339         B = exp( log(B) + eta*(log(B0) - log(B)) + berr*randn() );
340 
```

```
341         double BSI = Bfac*B*S*I;
342         double rI  = r*I;
343         double reR = re*R;
344 
345         // get derivatives
346         double dS = - BSI + reR;
347         double dI = BSI - rI;
348         double dR = rI - reR;
349 
350         // step forward by h
351         S += h*dS;
352         I += h*dI;
353         R += h*dR;
354 
355     }
356 
357     particle->S = S;
358     particle->I = I;
359     particle->R = R;
360     particle->B = B;
361 
362 }
363 
364 
365 /*  Particle pertubation function to be run between iterations and
        passes
366 
367     */
368 void perturbParticles(Particle * particles, int N, int NP, int
        passnum, double coolrate) {
369 
370     //double coolcoef = exp( - (double) passnum / coolrate );
371     double coolcoef = pow(coolrate, passnum);
372 
373 
374     double spreadR0     = coolcoef * R0true / 10.0;
375     double spreadr      = coolcoef * rtrue / 10.0;
376     double spreadre     = coolcoef * retrue / 10.0;
377     double spreadsigma  = coolcoef * merr / 10.0;
378     double spreadIinit  = coolcoef * I0 / 10.0;
379     double spreadeta    = coolcoef * etatrue / 10.0;
380     double spreadberr   = coolcoef * berrtrue / 10.0;
381 
382 
383     double R0can, rcan, recan, sigmacan, Iinitcan, etacan, berrcan;
384 
385     for (int n = 0; n < NP; n++) {
386 
387         do {
388             R0can = particles[n].R0 + spreadR0*randn();
389         } while (R0can < 0);
```

```
1  390         particles[n].R0 = R0can;
2  391
3  392         do {
4  393             rcan = particles[n].r + spreadr*randn();
5  394         } while (rcan < 0);
6  395         particles[n].r = rcan;
7  396
8  397         do {
9  398             recan = particles[n].re + spreadre*randn();
10 399         } while (recan < 0);
11 400         particles[n].re = recan;
12 401
13 402         do {
14 403             sigmacan = particles[n].sigma + spreadsigma*randn();
15 404         } while (sigmacan < 0);
16 405         particles[n].sigma = sigmacan;
17 406
18 407         do {
19 408             etacan = particles[n].eta + PSC*spreadeta*randn();
20 409         } while (etacan < 0 || etacan > 1);
21 410         particles[n].eta = etacan;
22 411
23 412         do {
24 413             berrcan = particles[n].berr + PSC*spreadberr*randn();
25 414         } while (berrcan < 0);
26 415         particles[n].berr = berrcan;
27 416
28 417         do {
29 418             Iinitcan = particles[n].Iinit + spreadIinit*randn();
30 419         } while (Iinitcan < 0 || Iinitcan > 500);
31 420         particles[n].Iinit = Iinitcan;
32 421         particles[n].Sinit = N - Iinitcan;
33 422
34 423     }
35 424
36 425 }
37 426
38 427
39 428 /*  Convinience function for particle resampling process
40 429
41 430     */
42 431 void copyParticle(Particle * dst, Particle * src) {
43 432
44 433     dst->R0     = src->R0;
45 434     dst->r      = src->r;
46 435     dst->re     = src->re;
47 436     dst->sigma  = src->sigma;
48 437     dst->eta    = src->eta;
49 438     dst->berr   = src->berr;
50 439     dst->B      = src->B;
51 440     dst->S      = src->S;
```

```
441     dst->I      = src->I;
442     dst->R      = src->R;
443     dst->Sinit  = src->Sinit;
444     dst->Iinit  = src->Iinit;
445     dst->Rinit  = src->Rinit;
446
447 }
448
449 void particleDiagnostics(ParticleInfo * partInfo, Particle *
        particles, int NP) {
450
451     double  R0mean      = 0.0,
452             rmean       = 0.0,
453             remean      = 0.0,
454             sigmamean   = 0.0,
455             etamean     = 0.0,
456             berrmean    = 0.0,
457             Sinitmean   = 0.0,
458             Iinitmean   = 0.0,
459             Rinitmean   = 0.0;
460
461     // means
462
463     for (int n = 0; n < NP; n++) {
464
465         R0mean      += particles[n].R0;
466         rmean       += particles[n].r;
467         remean      += particles[n].re;
468         etamean     += particles[n].eta,
469         berrmean    += particles[n].berr,
470         sigmamean   += particles[n].sigma;
471         Sinitmean   += particles[n].Sinit;
472         Iinitmean   += particles[n].Iinit;
473         Rinitmean   += particles[n].Rinit;
474
475     }
476
477     R0mean      /= NP;
478     rmean       /= NP;
479     remean      /= NP;
480     sigmamean   /= NP;
481     etamean     /= NP;
482     berrmean    /= NP;
483     Sinitmean   /= NP;
484     Iinitmean   /= NP;
485     Rinitmean   /= NP;
486
487     // standard deviations
488
489     double  R0sd    = 0.0,
490             rsd     = 0.0,
```

```
491             resd    = 0.0,
492             sigmasd = 0.0,
493             etasd   = 0.0,
494             berrsd  = 0.0,
495             Sinitsd = 0.0,
496             Iinitsd = 0.0,
497             Rinitsd = 0.0;
498
499     for (int n = 0; n < NP; n++) {
500
501         R0sd    += ( particles[n].R0 - R0mean ) * ( particles[n].R0 -
                R0mean );
502         rsd     += ( particles[n].r - rmean ) * ( particles[n].r -
                rmean );
503         resd    += ( particles[n].re - rmean ) * ( particles[n].re -
                rmean );
504         sigmasd += ( particles[n].sigma - sigmamean ) * ( particles[n
                ].sigma - sigmamean );
505         etasd   += ( particles[n].eta - etamean ) * ( particles[n].
                eta - etamean );
506         berrsd  += ( particles[n].berr - berrmean ) * ( particles[n].
                berr - berrmean );
507         Sinitsd += ( particles[n].Sinit - Sinitmean ) * ( particles[n
                ].Sinit - Sinitmean );
508         Iinitsd += ( particles[n].Iinit - Iinitmean ) * ( particles[n
                ].Iinit - Iinitmean );
509         Rinitsd += ( particles[n].Rinit - Rinitmean ) * ( particles[n
                ].Rinit - Rinitmean );
510
511     }
512
513     R0sd        /= NP;
514     rsd         /= NP;
515     resd        /= NP;
516     sigmasd     /= NP;
517     etasd       /= NP;
518     berrsd      /= NP;
519     Sinitsd     /= NP;
520     Iinitsd     /= NP;
521     Rinitsd     /= NP;
522
523     partInfo->R0mean    = R0mean;
524     partInfo->R0sd      = R0sd;
525     partInfo->rmean     = rmean;
526     partInfo->rsd       = rsd;
527     partInfo->remean    = remean;
528     partInfo->resd      = resd;
529     partInfo->sigmamean = sigmamean;
530     partInfo->sigmasd   = sigmasd;
531     partInfo->etamean   = etamean;
532     partInfo->etasd     = etasd;
```

```
533     partInfo->berrmean  = berrmean;
534     partInfo->berrsd    = berrsd;
535     partInfo->Sinitmean = Sinitmean;
536     partInfo->Sinitsd   = Sinitsd;
537     partInfo->Iinitmean = Iinitmean;
538     partInfo->Iinitsd   = Iinitsd;
539     partInfo->Rinitmean = Rinitmean;
540     partInfo->Rinitsd   = Rinitsd;
541
542 }
543
544 double randu() {
545
546     return (double) rand() / (double) RAND_MAX;
547
548 }
549
550 void getStateMeans(State * state, Particle* particles, int NP) {
551
552     double Smean = 0, Imean = 0, Rmean = 0;
553
554     for (int n = 0; n < NP; n++) {
555         Smean += particles[n].S;
556         Imean += particles[n].I;
557         Rmean += particles[n].R;
558     }
559
560     state->S = (double) Smean / NP;
561     state->I = (double) Imean / NP;
562     state->R = (double) Rmean / NP;
563
564 }
565
566
567 /*  Return a normally distributed random number with mean 0 and
        standard deviation 1
568     Uses the polar form of the Box-Muller transformation
569     From http://www.design.caltech.edu/erik/Misc/Gaussian.html
570     */
571 double randn() {
572
573     double x1, x2, w, y1;
574
575     do {
576         x1 = 2.0 * randu() - 1.0;
577         x2 = 2.0 * randu() - 1.0;
578         w = x1 * x1 + x2 * x2;
579     } while ( w >= 1.0 );
580
581     w = sqrt( (-2.0 * log( w ) ) / w );
582     y1 = x1 * w;
```

```
583
584     return y1;
585
586 }
```

Appendix F

Spatial Epidemics

F.1 Spatial SIR R Function Code

R code to simulate the outlined Spatial SIR function.

```
## ymat:  Contains the initial conditions where:
#       - rows are locations
#       - columns are S, I, R
## pars:  Contains the parameters: global values for R0, r, N, eta, berr
## T:     The stop time. Since 0 in included, there should be T+1 time steps in the simulation
## neinum:  Number of neighbors for each location, in order
## neibmat: Contains lists of neighbors for each location
#     - rows are parent locations (nodes)
#       - columns are locations each parent is attached to (edges)
StocSSIR ← function(ymat, pars, T, steps, neinum, neibmat) {

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

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

    ## extract parameters
    R0 ← pars[['R0']]
    r ← pars[['r']]
    N ← pars[['N']]
```

```
    eta ← pars[['eta']]
    berr ← pars[['berr']]
    phi ← pars[['phi']]

    B0 ← rep(R0*r/N, nloc)

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

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

    h ← 1 / steps

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

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

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

        #if(i == 1)
        # print(BSI)

        rI ← r*I

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

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

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

    }

    #out[,,2] ← cbind(S,I,R,B)

```

```
  return(out)

}

### Suggested parameters
#
# T       <- 60
# i_infec <- 5
# steps   <- 7
# N       <- 500
# sigma   <- 10
#
# pars <- c(R0 = 3.0,     # new infected people per infected person
#           r = 0.1,       # recovery rate
#           N = 500,       # population size
#           eta = 0.5,     # geometric random walk
#           berr = 0.5)    # Beta geometric walk noise
```

F.2 RStan Spatial SIR Code

This code implements a Spatial SIR model in Rstan.

```
data {

    int     <lower=1>   T;      // total integration steps
    int     <lower=1>   nloc;   // number of locations
    real                y[nloc, T];   // observed number of cases
    int     <lower=1>   N;      // population size
    real                h;      // step size
    int     <lower=0>   neinum[nloc];        // number of neighbors
       each location has
    int                 neibmat[nloc, nloc]; // neighbor list for
       each location

}

parameters {

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

```
    real <lower=0, upper=1>         phi;    // interconnectivity strength

}

model {

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

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

    B0 ← R0 * r / N;

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

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

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

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

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

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

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

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

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

        }
    }

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

}
```

F.3 IF2 Spatial SIR Code

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

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

    */

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

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

#define Treal       100         // time to simulate over
#define R0true      3.0         // infectiousness
#define rtrue       0.1         // recovery rate
#define Nreal       500.0       // population size
#define etatrue     0.5         // real drift attraction strength
#define berrtrue    0.5         // real beta drift noise
#define phitrue     0.5         // real connectivity strength
#define merr        10.0        // expected measurement error
```

```
#define I0          5.0         // Initial infected individuals

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

#include <Rcpp.h>
using namespace Rcpp;

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


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

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

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

```

```
67      srand(time(NULL));      // Seed PRNG with system time
68
69      double w[NP];           // particle weights
70
71      // initialize particles
72      printf("Initializing particle states\n");
73      Particle * particles = NULL;        // particle estimates for
            current step
74      Particle * particles_old = NULL;    // intermediate particle
            states for resampling
75      initializeParticles(&particles, NP, nloc, N);
76      initializeParticles(&particles_old, NP, nloc, N);
77
78      /*
79      // copy particle test
80      copyParticle(&particles[0], &particles_old[0], nloc);
81
82      // perturb particle test
83      perturbParticles(particles, N, NP, nloc, 1, coolrate);
84
85      // evolution test
86      // reset particle system evolution states
87      for (int n = 0; n < NP; n++) {
88          for (int loc = 0; loc < nloc; loc++) {
89              particles[n].S[loc] = N - particles[n].Iinit[loc];
90              particles[n].I[loc] = particles[n].Iinit[loc];
91              particles[n].R[loc] = 0.0;
92              particles[n].B[loc] = (double) particles[n].R0 *
                    particles[n].r / N;
93          }
94      }
95      printf("Before S:%f | I:%f | R:%f\n", particles[0].S[0],
            particles[0].I[0], particles[0].R[0]);
96      exp_euler_SSIR(1.0/7.0, 0.0, 1.0, N, &particles[0], neinum,
            neibmat, nloc);
97      printf("After S:%f | I:%f | R:%f\n", particles[0].S[0], particles
            [0].I[0], particles[0].R[0]);
98      */
99
100     // START PASSES THROUGH DATA
101
102     printf("Starting filter\n");
103     printf("---------------\n");
104     printf("Pass\n");
105
106
107     for (int pass = 0; pass < nPasses; pass++) {
108
109         printf("...%d / %d\n", pass, nPasses);
110
111         // reset particle system evolution states
```

```
1  112          for (int n = 0; n < NP; n++) {
2  113              for (int loc = 0; loc < nloc; loc++) {
3  114                  particles[n].S[loc] = N - particles[n].Iinit[loc];
4  115                  particles[n].I[loc] = particles[n].Iinit[loc];
5  116                  particles[n].R[loc] = 0.0;
6  117                  particles[n].B[loc] = (double) particles[n].R0 *
7                           particles[n].r / N;
8  118              }
9  119          }
10 120
11 121          if (pass == (nPasses-1)) {
12 122              double means[nloc];
13 123              for (int loc = 0; loc < nloc; loc++) {
14 124                  means[loc] = 0.0;
15 125                  for (int n = 0; n < NP; n++) {
16 126                      means[loc] += particles[n].I[loc] / NP;
17 127                  }
18 128                  infecmeans(loc, 0) = means[loc];
19 129              }
20 130          }
21 131
22 132          for (int t = 1; t < T; t++) {
23 133
24 134              // generate individual predictions and weight
25 135              for (int n = 0; n < NP; n++) {
26 136
27 137                  exp_euler_SSIR(1.0/7.0, 0.0, 1.0, N, &particles[n],
28                          neinum, neibmat, nloc);
29 138
30 139                  double merr_par = particles[n].sigma;
31 140
32 141                  w[n] = 1.0;
33 142                  for (int loc = 0; loc < nloc; loc++) {
34 143                      double y_diff   = data(loc, t) - particles[n].I[
35                              loc];
36 144                      w[n] *= 1.0/(merr_par*sqrt(2.0*M_PI)) * exp( -
37                              y_diff*y_diff / (2.0*merr_par*merr_par) );
38 145                  }
39 146
40 147              }
41 148
42 149              // cumulative sum
43 150              for (int n = 1; n < NP; n++) {
44 151                  w[n] += w[n-1];
45 152              }
46 153
47 154              // save particle states to resample from
48 155              for (int n = 0; n < NP; n++){
49 156                  copyParticle(&particles_old[n], &particles[n], nloc);
50 157              }
51 158
```

```
159             // resampling
160             for (int n = 0; n < NP; n++) {
161 
162                 double w_r = randu() * w[NP-1];
163                 int i = 0;
164                 while (w_r > w[i]) {
165                     i++;
166                 }
167 
168                 // i is now the index to copy state from
169                 copyParticle(&particles[n], &particles_old[i], nloc);
170 
171             }
172 
173             // between-iteration perturbations, not after last time
                    step
174             if (t < (T-1))
175                 perturbParticles(particles, N, NP, nloc, pass,
                        coolrate);
176 
177             if (pass == (nPasses-1)) {
178                 double means[nloc];
179                 for (int loc = 0; loc < nloc; loc++) {
180                     means[loc] = 0.0;
181                     for (int n = 0; n < NP; n++) {
182                         means[loc] += particles[n].I[loc] / NP;
183                     }
184                     infecmeans(loc, t) = means[loc];
185                 }
186             }
187 
188         }
189 
190         // between-pass perturbations, not after last pass
191         if (pass < (nPasses + 1))
192             perturbParticles(particles, N, NP, nloc, pass, coolrate);
193 
194     }
195 
196     // pack parameter data (minus initial conditions)
197     for (int n = 0; n < NP; n++) {
198         paramdata(n, 0) = particles[n].R0;
199         paramdata(n, 1) = particles[n].r;
200         paramdata(n, 2) = particles[n].sigma;
201         paramdata(n, 3) = particles[n].eta;
202         paramdata(n, 4) = particles[n].berr;
203         paramdata(n, 5) = particles[n].phi;
204     }
205 
206     // Pack initial condition data
207     for (int n = 0; n < NP; n++) {
```

```
        for (int loc = 0; loc < nloc; loc++) {
            initInfec(loc, n) = particles[n].Iinit[loc];
        }
    }

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


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



}


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

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

```

```
253     int num_steps = floor( (tn-t0) / h );
254
255     double * S = particle->S;
256     double * I = particle->I;
257     double * R = particle->R;
258     double * B = particle->B;
259
260     // create last state vectors
261     double S_last[nloc];
262     double I_last[nloc];
263     double R_last[nloc];
264     double B_last[nloc];
265
266     double R0   = particle->R0;
267     double r    = particle->r;
268     double B0   = R0 * r / N;
269     double eta  = particle->eta;
270     double berr = particle->berr;
271     double phi  = particle->phi;
272
273     //printf("sphi \t\t| ophi \t\t| BSI \t\t| rI \t\t| dS \t\t| dI \t
            \t| dR \t\t| S \t\t| I \t\t| R |\n");
274
275     for(int t = 0; t < num_steps; t++) {
276
277         for (int loc = 0; loc < nloc; loc++) {
278             S_last[loc] = S[loc];
279             I_last[loc] = I[loc];
280             R_last[loc] = R[loc];
281             B_last[loc] = B[loc];
282         }
283
284         for (int loc = 0; loc < nloc; loc++) {
285
286             B[loc] = exp( log(B_last[loc]) + eta*(log(B0) - log(
                    B_last[loc])) + berr*randn() );
287
288             int n = neinum[loc];
289             double sphi = 1.0 - phi*( (double) n/(n+1.0) );
290             double ophi = phi/(n+1.0);
291
292             double nBIsum = 0.0;
293             for (int j = 0; j < n; j++)
294                 nBIsum += B_last[(int) neibmat(loc, j) - 1] * I_last
                        [(int) neibmat(loc, j) - 1];
295
296             double BSI = S_last[loc]*( sphi*B_last[loc]*I_last[loc] +
                     ophi*nBIsum );
297             double rI  = r*I_last[loc];
298
299             // get derivatives
```

```
300             double dS = - BSI;
301             double dI = BSI - rI;
302             double dR = rI;
303 
304             // step forward by h
305             S[loc] += h*dS;
306             I[loc] += h*dI;
307             R[loc] += h*dR;
308 
309             //if (loc == 1)
310             //  printf("%f\t|%f\t|%f\t|%f\t|%f\t|%f\t|%f\t|%f\t|%f\t
                    |%f\t|\n", sphi, ophi, BSI, rI, dS, dI, dR, S[1], I
                    [1], R[1]);
311 
312         }
313 
314     }
315 
316     /*particle->S = S;
317     particle->I = I;
318     particle->R = R;
319     particle->B = B;*/
320 
321 }
322 
323 /*  Initializes particles
324     */
325 void initializeParticles(Particle ** particles, int NP, int nloc, int
        N) {
326 
327     // allocate space for doubles
328     *particles = (Particle*) malloc (NP*sizeof(Particle));
329 
330     // allocate space for arays inside particles
331     for (int n = 0; n < NP; n++) {
332         (*particles)[n].S = (double*) malloc(nloc*sizeof(double));
333         (*particles)[n].I = (double*) malloc(nloc*sizeof(double));
334         (*particles)[n].R = (double*) malloc(nloc*sizeof(double));
335         (*particles)[n].B = (double*) malloc(nloc*sizeof(double));
336         (*particles)[n].Iinit = (double*) malloc(nloc*sizeof(double))
                ;
337     }
338 
339     // initialize all all parameters
340     for (int n = 0; n < NP; n++) {
341 
342         double R0can, rcan, sigmacan, Iinitcan, etacan, berrcan,
                phican;
343 
344         do {
345             R0can = R0true + R0true*randn();
```

```
346         } while (R0can < 0);
347         (*particles)[n].R0 = R0can;
348 
349         do {
350             rcan = rtrue + rtrue*randn();
351         } while (rcan < 0);
352         (*particles)[n].r = rcan;
353 
354         for (int loc = 0; loc < nloc; loc++)
355             (*particles)[n].B[loc] = (double) R0can * rcan / N;
356 
357         do {
358             sigmacan = merr + merr*randn();
359         } while (sigmacan < 0);
360         (*particles)[n].sigma = sigmacan;
361 
362         do {
363             etacan = etatrue + PSC*etatrue*randn();
364         } while (etacan < 0 || etacan > 1);
365         (*particles)[n].eta = etacan;
366 
367         do {
368             berrcan = berrtrue + PSC*berrtrue*randn();
369         } while (berrcan < 0);
370         (*particles)[n].berr = berrcan;
371 
372         do {
373             phican = phitrue + PSC*phitrue*randn();
374         } while (phican <= 0 || phican >= 1);
375         (*particles)[n].phi = phican;
376 
377         for (int loc = 0; loc < nloc; loc++) {
378             do {
379                 Iinitcan = I0 + I0*randn();
380             } while (Iinitcan < 0 || N < Iinitcan);
381             (*particles)[n].Iinit[loc] = Iinitcan;
382         }
383 
384     }
385 
386 }
387 
388 /*  Particle pertubation function to be run between iterations and
        passes
389 
390     */
391 void perturbParticles(Particle * particles, int N, int NP, int nloc,
        int passnum, double coolrate) {
392 
393     //double coolcoef = exp( - (double) passnum / coolrate );
394     double coolcoef = pow(coolrate, passnum);
```

```
395
396     double spreadR0     = coolcoef * R0true / 10.0;
397     double spreadr      = coolcoef * rtrue / 10.0;
398     double spreadsigma  = coolcoef * merr / 10.0;
399     double spreadIinit  = coolcoef * I0 / 10.0;
400     double spreadeta    = coolcoef * etatrue / 10.0;
401     double spreadberr   = coolcoef * berrtrue / 10.0;
402     double spreadphi    = coolcoef * phitrue / 10.0;
403
404     double R0can, rcan, sigmacan, Iinitcan, etacan, berrcan, phican;
405
406     for (int n = 0; n < NP; n++) {
407
408         do {
409             R0can = particles[n].R0 + spreadR0*randn();
410         } while (R0can < 0);
411         particles[n].R0 = R0can;
412
413         do {
414             rcan = particles[n].r + spreadr*randn();
415         } while (rcan < 0);
416         particles[n].r = rcan;
417
418         do {
419             sigmacan = particles[n].sigma + spreadsigma*randn();
420         } while (sigmacan < 0);
421         particles[n].sigma = sigmacan;
422
423         do {
424             etacan = particles[n].eta + PSC*spreadeta*randn();
425         } while (etacan < 0 || etacan > 1);
426         particles[n].eta = etacan;
427
428         do {
429             berrcan = particles[n].berr + PSC*spreadberr*randn();
430         } while (berrcan < 0);
431         particles[n].berr = berrcan;
432
433         do {
434             phican = particles[n].phi + PSC*spreadphi*randn();
435         } while (phican <= 0 || phican >= 1);
436         particles[n].phi = phican;
437
438         for (int loc = 0; loc < nloc; loc++) {
439             do {
440                 Iinitcan = particles[n].Iinit[loc] + spreadIinit*
                        randn();
441             } while (Iinitcan < 0 || Iinitcan > 500);
442             particles[n].Iinit[loc] = Iinitcan;
443         }
444     }
```

```
445
446 }
447
448 /*  Convinience function for particle resampling process
449     */
450 void copyParticle(Particle * dst, Particle * src, int nloc) {
451
452     dst->R0     = src->R0;
453     dst->r      = src->r;
454     dst->sigma  = src->sigma;
455     dst->eta    = src->eta;
456     dst->berr   = src->berr;
457     dst->phi    = src->phi;
458
459     for (int n = 0; n < nloc; n++) {
460         dst->S[n]       = src->S[n];
461         dst->I[n]       = src->I[n];
462         dst->R[n]       = src->R[n];
463         dst->B[n]       = src->B[n];
464         dst->Iinit[n]   = src->Iinit[n];
465     }
466
467 }
468
469
470
471 double randu() {
472
473     return (double) rand() / (double) RAND_MAX;
474
475 }
476
477 /*
478 void getStateMeans(State * state, Particle* particles, int NP) {
479
480     double Smean = 0, Imean = 0, Rmean = 0;
481
482     for (int n = 0; n < NP; n++) {
483         Smean += particles[n].S;
484         Imean += particles[n].I;
485         Rmean += particles[n].R;
486     }
487
488     state->S = (double) Smean / NP;
489     state->I = (double) Imean / NP;
490     state->R = (double) Rmean / NP;
491
492 }
493 */
494
495 /*  Return a normally distributed random number with mean 0 and
```

```
    standard deviation 1
    Uses the polar form of the Box-Muller transformation
    From http://www.design.caltech.edu/erik/Misc/Gaussian.html
    */
double randn() {

    double x1, x2, w, y1;

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

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

    return y1;

}
```

F.4 CUDA IF2 Spatial Fitting Code

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

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

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

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

    Compile with:

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

    */

```

```
1  22 #include <cuda.h>
2  23 #include <iostream>
3  24 #include <fstream>
4  25 #include <curand.h>
5  26 #include <curand_kernel.h>
6  27 #include <string>
7  28 #include <sstream>
8  29 #include <cmath>
9  30
10 31 #include "timer.h"
11 32 #include "rand.h"
12 33 #include "readdata.h"
13 34
14 35 #define NP          (2*2500)    // number of particles
15 36 #define N           500.0       // population size
16 37 #define R0true      3.0         // infectiousness
17 38 #define rtrue       0.1         // recovery rate
18 39 #define etatrue     0.5         // real drift attraction strength
19 40 #define berrtrue    0.5         // real beta drift noise
20 41 #define phitrue     0.5         // real connectivity strength
21 42 #define merr        10.0        // expected measurement error
22 43 #define I0          5.0         // Initial infected individuals
23 44 #define PSC         0.5         // sensitive parameter perturbation
24       scaling
25 45 #define NLOC        10
26 46
27 47 #define PI      3.141592654f
28 48
29 49 // Wrapper for CUDA calls, from CUDA API
30 50 // Modified to also print the error code and string
31 51 # define CUDA_CALL(x) do { if ((x) != cudaSuccess ) {
32          \
33 52     std::cout << " Error at " << __FILE__ << ":" << __LINE__ << std::
34          endl;      \
35 53     std::cout << " Error was " << x << " " << cudaGetErrorString(x)
36          << std::endl;   \
37 54     return EXIT_FAILURE ;}} while (0)
38           \
39 55
40 56 typedef struct {
41 57     float R0;
42 58     float r;
43 59     float sigma;
44 60     float eta;
45 61     float berr;
46 62     float phi;
47 63     float S[NLOC];
48 64     float I[NLOC];
49 65     float R[NLOC];
50 66     float B[NLOC];
51 67     float Iinit[NLOC];
```

```
    curandState randState;  // PRNG state
} Particle;

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

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


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

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

    if (id < NP) {

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

        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);
```

```
1  113         particles[id].eta = etacan;
2  114
3  115         do {
4  116             berrcan = berrtrue + PSC*berrtrue*curand_normal(&state);
5  117         } while (berrcan < 0);
6  118         particles[id].berr = berrcan;
7  119
8  120         do {
9  121             phican = phitrue + PSC*phitrue*curand_normal(&state);
10 122         } while (phican <= 0 || phican >= 1);
11 123         particles[id].phi = phican;
12 124
13 125         for (int loc = 0; loc < nloc; loc++) {
14 126             do {
15 127                 Iinitcan = I0 + I0*curand_normal(&state);
16 128             } while (Iinitcan < 0 || N < Iinitcan);
17 129             particles[id].Iinit[loc] = Iinitcan;
18 130         }
19 131
20 132         particles[id].randState = state;
21 133
22 134     }
23 135
24 136 }
25 137
26 138 __global__ void resetStates (Particle * particles, int nloc) {
27 139
28 140     int id  = blockIdx.x*blockDim.x + threadIdx.x;  // global thread
29           ID
30 141
31 142     if (id < NP) {
32 143
33 144         for (int loc = 0; loc < nloc; loc++) {
34 145             particles[id].S[loc] = N - particles[id].Iinit[loc];
35 146             particles[id].I[loc] = particles[id].Iinit[loc];
36 147             particles[id].R[loc] = 0.0;
37 148         }
38 149
39 150     }
40 151
41 152 }
42 153
43 154 __global__ void clobberParams (Particle * particles, int nloc) {
44 155
45 156     int id  = blockIdx.x*blockDim.x + threadIdx.x;  // global thread
46           ID
47 157
48 158     if (id < NP) {
49 159
50 160         particles[id].R0 = R0true;
51 161         particles[id].r = rtrue;
```

```
        particles[id].sigma = merr;
        particles[id].eta = etatrue;
        particles[id].berr = berrtrue;
        particles[id].phi = phitrue;

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

    }

}


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

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

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

}

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

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

    if (id < NP) {

        float merr_par = particles[id].sigma;

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

        w[id] = w_local;

    }

```

```
}

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

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

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

}


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

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

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

}


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

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

    if (id < NP) {

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

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

```
256
257     }
258
259 }
260
261 // launch this with probably just nloc threads... block structure/
        size probably not important
262 __global__ void reduceStates (Particle * particles, float *
        countmeans, int t, int T, int nloc) {
263
264     int id  = blockIdx.x*blockDim.x + threadIdx.x;
265
266     if (id < nloc) {
267
268         int loc = id;
269
270         double countmean_local = 0.0;
271         for (int n = 0; n < NP; n++) {
272             countmean_local += particles[n].I[loc] / NP;
273         }
274
275         countmeans[loc*T + t] = (float) countmean_local;
276
277     }
278
279 }
280
281 __global__ void perturbParticles(Particle * particles, int nloc, int
        passnum, double coolrate) {
282
283     //double coolcoef = exp( - (double) passnum / coolrate );
284     double coolcoef = pow(coolrate, passnum);
285
286     double spreadR0     = coolcoef * R0true / 10.0;
287     double spreadr      = coolcoef * rtrue / 10.0;
288     double spreadsigma  = coolcoef * merr / 10.0;
289     double spreadIinit  = coolcoef * I0 / 10.0;
290     double spreadeta    = coolcoef * etatrue / 10.0;
291     double spreadberr   = coolcoef * berrtrue / 10.0;
292     double spreadphi    = coolcoef * phitrue / 10.0;
293
294     double R0can, rcan, sigmacan, Iinitcan, etacan, berrcan, phican;
295
296     int id  = blockIdx.x*blockDim.x + threadIdx.x;
297
298     if (id < NP) {
299
300         do {
301             R0can = particles[id].R0 + spreadR0*curand_normal(&
                    particles[id].randState);
302         } while (R0can < 0);
```

```
303         particles[id].R0 = R0can;
304
305         do {
306             rcan = particles[id].r + spreadr*curand_normal(&particles
                    [id].randState);
307         } while (rcan < 0);
308         particles[id].r = rcan;
309
310         do {
311             sigmacan = particles[id].sigma + spreadsigma*
                    curand_normal(&particles[id].randState);
312         } while (sigmacan < 0);
313         particles[id].sigma = sigmacan;
314
315         do {
316             etacan = particles[id].eta + PSC*spreadeta*curand_normal
                    (&particles[id].randState);
317         } while (etacan < 0 || etacan > 1);
318         particles[id].eta = etacan;
319
320         do {
321             berrcan = particles[id].berr + PSC*spreadberr*
                    curand_normal(&particles[id].randState);
322         } while (berrcan < 0);
323         particles[id].berr = berrcan;
324
325         do {
326             phican = particles[id].phi + PSC*spreadphi*curand_normal
                    (&particles[id].randState);
327         } while (phican <= 0 || phican >= 1);
328         particles[id].phi = phican;
329
330         for (int loc = 0; loc < nloc; loc++) {
331             do {
332                 Iinitcan = particles[id].Iinit[loc] + spreadIinit*
                        curand_normal(&particles[id].randState);
333             } while (Iinitcan < 0 || Iinitcan > 500);
334             particles[id].Iinit[loc] = Iinitcan;
335         }
336
337     }
338
339 }
340
341
342 int main (int argc, char *argv[]) {
343
344
345     int T, nloc;
346
347     double restime;
```

```
348     struct timeval tdr0, tdr1, tdrMaster;
349
350     // Parse arguments ************************************************
351
352     if (argc < 4) {
353         std::cout << "Not enough arguments" << std::endl;
354         return 0;
355     }
356
357     std::string arg1(argv[1]);  // infection counts
358     std::string arg2(argv[2]);  // neighbour counts
359     std::string arg3(argv[3]);  // neighbour indices
360     std::string arg4(argv[4]);  // outfile: params + runtime
361
362     std::cout << "Arguments:" << std::endl;
363     std::cout << "Infection data:    " << arg1 << std::endl;
364     std::cout << "Neighbour counts:  " << arg2 << std::endl;
365     std::cout << "Neighbour indices: " << arg3 << std::endl;
366     std::cout << "Outfile            " << arg4 << std::endl;
367
368     // ************************************************************
369
370
371     // Read count data ************************************************
372
373     std::cout << "Getting count data" << std::endl;
374     float * data = getDataFloat(arg1, &T, &nloc);
375     size_t datasize = nloc*T*sizeof(float);
376
377     // ************************************************************
378
379     // Read neinum matrix data ************************************
380
381     std::cout << "Getting neighbour count data" << std::endl;
382     int * neinum = getDataInt(arg2, NULL, NULL);
383     size_t neinumsize = nloc * sizeof(int);
384
385     // ************************************************************
386
387     // Read neibmat matrix data ************************************
388
389     std::cout << "Getting neighbour count data" << std::endl;
390     int * neibmat = getDataInt(arg3, NULL, NULL);
391     size_t neibmatsize = nloc * nloc * sizeof(int);
392
393     // ************************************************************
394
395     //
        ************************************************************************
50
51  396
```

```
397     // start timing
398     gettimeofday (&tdr0, NULL);
399
400     // CUDA data ************************************************
401
402     std::cout << "Allocating device storage" << std::endl;
403
404     float       * d_data;           // device copy of data
405     Particle    * particles;        // particles
406     Particle    * particles_old;    // intermediate particle states
407     double      * w;                // weights
408     int         * d_neinum;         // device copy of adjacency
            matrix
409     int         * d_neibmat;        // device copy of neighbour
            counts matrix
410     float       * countmeans;       // host copy of reduced infection
             count means from last pass
411     float       * d_countmeans;     // device copy of reduced
            infection count means from last pass
412
413     CUDA_CALL( cudaMalloc( (void**) &d_data         , datasize )
                );
414     CUDA_CALL( cudaMalloc( (void**) &particles      , NP*sizeof(
            Particle))  );
415     CUDA_CALL( cudaMalloc( (void**) &particles_old  , NP*sizeof(
            Particle))  );
416     CUDA_CALL( cudaMalloc( (void**) &w              , NP*sizeof(
            double))    );
417     CUDA_CALL( cudaMalloc( (void**) &d_neinum       , neinumsize)
                );
418     CUDA_CALL( cudaMalloc( (void**) &d_neibmat      , neibmatsize)
                );
419     CUDA_CALL( cudaMalloc( (void**) &d_countmeans   , nloc*T*sizeof(
            float)) );
420
421
422     gettimeofday (&tdr1, NULL);
423     timeval_subtract (&restime, &tdr1, &tdr0);
424
425     std::cout << "\t" << getHRtime(restime) << std::endl;
426
427     size_t avail, total;
428     cudaMemGetInfo( &avail, &total );
429     size_t used = total - avail;
430
431     std::cout << "\t[" << getHRmemsize(used) << "] used of [" <<
            getHRmemsize(total) << "]" <<std::endl;
432
433     std::cout << "Copying data to device" << std::endl;
434
435     gettimeofday (&tdr0, NULL);
```

```
436
437     CUDA_CALL( cudaMemcpy(d_data    , data      , datasize      ,
            cudaMemcpyHostToDevice)   );
438     CUDA_CALL( cudaMemcpy(d_neinum  , neinum    , neinumsize    ,
            cudaMemcpyHostToDevice)   );
439     CUDA_CALL( cudaMemcpy(d_neibmat , neibmat   , neibmatsize   ,
            cudaMemcpyHostToDevice)   );
440
441     gettimeofday (&tdr1, NULL);
442     timeval_subtract (&restime, &tdr1, &tdr0);
443
444     std::cout << "\t" << getHRtime(restime) << std::endl;
445
446     // ************************************************************
447
448
449
450     // Initialize particles ***************************************
451
452     std::cout << "Initializing particles" << std::endl;
453
454     //gettimeofday (&tdr0, NULL);
455
456     int nThreads    = 32;
457     int nBlocks     = ceil( (float) NP / nThreads);
458
459     initializeParticles <<< nBlocks, nThreads >>> (particles, nloc);
460     CUDA_CALL( cudaGetLastError() );
461     CUDA_CALL( cudaDeviceSynchronize() );
462
463     initializeParticles <<< nBlocks, nThreads >>> (particles_old,
            nloc);
464     CUDA_CALL( cudaGetLastError() );
465     CUDA_CALL( cudaDeviceSynchronize() );
466
467     //gettimeofday (&tdr1, NULL);
468     //timeval_subtract (&restime, &tdr1, &tdr0);
469     //std::cout << "\t" << getHRtime(restime) << std::endl;
470
471     cudaMemGetInfo( &avail, &total );
472     used = total - avail;
473     std::cout << "\t[" << getHRmemsize(used) << "] used of [" <<
            getHRmemsize(total) << "]" <<std::endl;
474
475     // ************************************************************
476
477     // Starting filtering *****************************************
478
479     for (int pass = 0; pass < 50; pass++) {
480
481         //std::cout << "pass = " << pass << std::endl;
```

```
1  482
2  483         // ** TEMP **
3  484         //clobberParams <<< nBlocks, nThreads >>> (particles, nloc);
4  485         // ** TEMP **
5  486
6  487         nThreads    = 32;
7  488         nBlocks     = ceil( (float) NP / nThreads);
8  489
9  490         resetStates <<< nBlocks, nThreads >>> (particles, nloc);
10 491         CUDA_CALL( cudaGetLastError() );
11 492         CUDA_CALL( cudaDeviceSynchronize() );
12 493
13 494         //std::cout << "Filtering over [1," << Tlim << "]"<< std::
14                 endl;
15 495
16 496         //gettimeofday (&tdr0, NULL);
17 497
18 498         nThreads = 1;
19 499         nBlocks  = 10;
20 500
21 501         if (pass == 49) {
22 502             reduceStates <<< nBlocks, nThreads >>> (particles,
23                     d_countmeans, 0, T, nloc);
24 503             CUDA_CALL( cudaGetLastError() );
25 504             CUDA_CALL( cudaDeviceSynchronize() );
26 505         }
27 506
28 507         //gettimeofday (&tdr1, NULL);
29 508         //timeval_subtract (&restime, &tdr1, &tdr0);
30 509         //std::cout << "\tReduction " << getHRtime(restime) << std::
31                 endl;
32 510
33 511         int Tlim = T;
34 512
35 513         for (int t = 1; t < Tlim; t++) {
36 514
37 515             // Projection
38                     **********************************************
39 516
40 517             nThreads    = 32;
41 518             nBlocks     = ceil( (float) NP / nThreads);
42 519
43 520             //if (t == 1)
44 521             //  gettimeofday (&tdr0, NULL);
45 522
46 523             project <<< nBlocks, nThreads >>> (particles, d_neinum,
47                     d_neibmat, nloc);
48 524             CUDA_CALL( cudaGetLastError() );
49 525             CUDA_CALL( cudaDeviceSynchronize() );
50 526
51 527             //if (t == 1 && pass == 0) {
```

```
528             //  gettimeofday (&tdr1, NULL);
529             //  timeval_subtract (&restime, &tdr1, &tdr0);
530             //  std::cout << "\tProjection " << getHRtime(restime) <<
                    std::endl;
531             //}
532
533             // Weighting
                   ***********************************************
534
535             nThreads    = 32;
536             nBlocks     = ceil( (float) NP / nThreads);
537
538             weight <<< nBlocks, nThreads >>>(d_data, particles, w, t,
                    T, nloc);
539             CUDA_CALL( cudaGetLastError() );
540             CUDA_CALL( cudaDeviceSynchronize() );
541
542             // Cumulative sum
                   *****************************************
543
544             nThreads    = 1;
545             nBlocks     = 1;
546
547             //if (t == 1)
548             //  gettimeofday (&tdr0, NULL);
549
550             cumsumWeights <<< nBlocks, nThreads >>> (w);
551             CUDA_CALL( cudaGetLastError() );
552             CUDA_CALL( cudaDeviceSynchronize() );
553
554             //if (t == 1) {
555             //  gettimeofday (&tdr1, NULL);
556             //  timeval_subtract (&restime, &tdr1, &tdr0);
557             //  std::cout << "\tCumulative sum   " << getHRtime(
                   restime) << std::endl;
558             //}
559
560             // Save particles for resampling from
                   ************************
561
562             nThreads    = 32;
563             nBlocks     = ceil( (float) NP / nThreads);
564
565             stashParticles <<< nBlocks, nThreads >>> (particles,
                   particles_old, nloc);
566             CUDA_CALL( cudaGetLastError() );
567             CUDA_CALL( cudaDeviceSynchronize() );
568
569
570             // Resampling
                   ***********************************************
```

```
571
572             nThreads    = 32;
573             nBlocks     = ceil( (float) NP/ nThreads);
574
575             //if (t == 1)
576             //  gettimeofday (&tdr0, NULL);
577
578             resample <<< nBlocks, nThreads >>> (particles,
                    particles_old, w, nloc);
579             CUDA_CALL( cudaGetLastError() );
580             CUDA_CALL( cudaDeviceSynchronize() );
581
582             //if (t == 1) {
583             //  gettimeofday (&tdr1, NULL);
584             //  timeval_subtract (&restime, &tdr1, &tdr0);
585             //  std::cout << "\tResampling " << getHRtime(restime) <<
                     std::endl;
586             //}
587
588             // Reduction
                    ************************************************
589
590             //if (t == (Tlim-1)) {
591
592             if (pass == 49) {
593
594                 //if (t == 1)
595                 //  gettimeofday (&tdr0, NULL);
596
597                 nThreads = 1;
598                 nBlocks  = 10;
599
600                 reduceStates <<< nBlocks, nThreads >>> (particles,
                        d_countmeans, t, T, nloc);
601                 CUDA_CALL( cudaGetLastError() );
602                 CUDA_CALL( cudaDeviceSynchronize() );
603
604                 //if (t == 1) {
605                 //  gettimeofday (&tdr1, NULL);
606                 //  timeval_subtract (&restime, &tdr1, &tdr0);
607                 //  std::cout << "Reduction        " << getHRtime(
                        restime) << std::endl;
608                 //}
609
610             }
611
612             // Perturb particles
                    ****************************************
613
614             nThreads    = 32;
615             nBlocks     = ceil( (float) NP/ nThreads);
```

```
616
617             perturbParticles <<< nBlocks, nThreads >>> (particles,
                    nloc, pass, 0.975);
618             CUDA_CALL( cudaGetLastError() );
619             CUDA_CALL( cudaDeviceSynchronize() );
620
621
622         } // end time
623
624     } // end pass
625
626     std::cout.precision(10);
627
628     countmeans = (float*) malloc (nloc*T*sizeof(float));
629     cudaMemcpy(countmeans, d_countmeans, nloc*T*sizeof(float),
                cudaMemcpyDeviceToHost);
630
631     // stop master timer and print
632
633     gettimeofday (&tdrMaster, NULL);
634     timeval_subtract(&restime, &tdrMaster, &tdr0);
635     std::cout << "Time: " << getHRtime(restime) << std::endl;
636     std::cout << "Rawtime: " << restime << std::endl;
637
638     // Write results out
639
640     std::string filename = arg4;
641
642     std::cout << "Writing results to file '" << filename << "' ..."
                << std::endl;
643
644     std::ofstream outfile;
645     outfile.open(filename.c_str());
646
647     for(int loc = 0; loc < nloc; loc++) {
648         for (int t = 0; t < T; t++) {
649             outfile << countmeans[loc*T + t] << " ";
650         }
651         outfile << std::endl;
652     }
653
654     outfile.close();
655
656     //gettimeofday (&tdr1, NULL);
657     //timeval_subtract (&restime, &tdr1, &tdrMaster);
658     //std::cout << "Total PF time (excluding setup) " << getHRtime(
            restime) << std::endl;
659
660     cudaFree(d_data);
661     cudaFree(particles);
662     cudaFree(particles_old);
```

```
663     cudaFree(w);
664     cudaFree(d_neinum);
665     cudaFree(d_neibmat);
666     cudaFree(d_countmeans);
667 
668     exit (EXIT_SUCCESS);
669 
670 }
671 
672 
673 /*  Use the Explicit Euler integration scheme to integrate SIR model
        forward in time
674     float h     - time step size
675     float t0    - start time
676     float tn    - stop time
677     float * y   - current system state; a three-component vector
                      representing [S I R], susceptible-infected-recovered
678     */
679 __device__ void exp_euler_SSIR(float h, float t0, float tn, Particle
        * particle, int * neinum, int * neibmat, int nloc) {
680 
681     int num_steps = floor( (tn-t0) / h );
682 
683     float * S = particle->S;
684     float * I = particle->I;
685     float * R = particle->R;
686     float * B = particle->B;
687 
688     // create last state vectors
689     float * S_last = (float*) malloc (nloc*sizeof(float));
690     float * I_last = (float*) malloc (nloc*sizeof(float));
691     float * R_last = (float*) malloc (nloc*sizeof(float));
692     float * B_last = (float*) malloc (nloc*sizeof(float));
693 
694     float R0    = particle->R0;
695     float r     = particle->r;
696     float B0    = R0 * r / N;
697     float eta   = particle->eta;
698     float berr  = particle->berr;
699     float phi   = particle->phi;
700 
701     for(int t = 0; t < num_steps; t++) {
702 
703         for (int loc = 0; loc < nloc; loc++) {
704             S_last[loc] = S[loc];
705             I_last[loc] = I[loc];
706             R_last[loc] = R[loc];
707             B_last[loc] = B[loc];
708         }
709 
710         for (int loc = 0; loc < nloc; loc++) {
```

```
1  711
2  712             B[loc] = exp( log(B_last[loc]) + eta*(log(B0) - log(
3                      B_last[loc])) + berr*curand_normal(&(particle->
4                      randState)) );
5  713
6  714             int n = neinum[loc];
7  715             float sphi = 1.0 - phi*( (float) n/(n+1.0) );
8  716             float ophi = phi/(n+1.0);
9  717
10 718             float nBIsum = 0.0;
11 719             for (int j = 0; j < n; j++)
12 720                 nBIsum += B_last[neibmat[nloc*loc + j]-1] * I_last[
13                        neibmat[nloc*loc + j]-1];
14 721
15 722             float BSI = S_last[loc]*( sphi*B_last[loc]*I_last[loc] +
16                     ophi*nBIsum );
17 723             float rI  = r*I_last[loc];
18 724
19 725             // get derivatives
20 726             float dS = - BSI;
21 727             float dI = BSI - rI;
22 728             float dR = rI;
23 729
24 730             // step forward by h
25 731             S[loc] += h*dS;
26 732             I[loc] += h*dI;
27 733             R[loc] += h*dR;
28 734
29 735         }
30 736
31 737     }
32 738
33 739     free(S_last);
34 740     free(I_last);
35 741     free(R_last);
36 742     free(B_last);
37 743
38 744 }
39 745
40 746 /*  Convinience function for particle resampling process
41 747     */
42 748 __device__ void copyParticle(Particle * dst, Particle * src, int nloc
43        ) {
44 749
45 750     dst->R0     = src->R0;
46 751     dst->r      = src->r;
47 752     dst->sigma  = src->sigma;
48 753     dst->eta    = src->eta;
49 754     dst->berr   = src->berr;
50 755     dst->phi    = src->phi;
51 756
```

```
1  757     for (int n = 0; n < nloc; n++) {
2  758         dst->S[n]       = src->S[n];
3  759         dst->I[n]       = src->I[n];
4  760         dst->R[n]       = src->R[n];
5  761         dst->B[n]       = src->B[n];
6  762         dst->Iinit[n]   = src->Iinit[n];
7  763     }
8  764
9  765 }
10 766
11 767 /*  Convert memory size in bytes to human-readable format
12 768     */
13 769 std::string getHRmemsize (size_t memsize) {
14 770
15 771     std::stringstream ss;
16 772     std::string valstring;
17 773
18 774     int kb = 1024;
19 775     int mb = kb*1024;
20 776     int gb = mb*1024;
21 777
22 778     if (memsize <= kb)
23 779         ss << memsize << " B";
24 780     else if (memsize > kb && memsize <= mb)
25 781         ss << (float) memsize/ kb << " KB";
26 782     else if (memsize > mb && memsize <= gb)
27 783         ss << (float) memsize/ mb << " MB";
28 784     else
29 785         ss << (float) memsize/ gb << " GB";
30 786
31 787     valstring = ss.str();
32 788
33 789     return valstring;
34 790
35 791 }
36 792
37 793
38 794 /*  Convert time in seconds to human readable format
39 795     */
40 796 std::string getHRtime (float runtime) {
41 797
42 798     std::stringstream ss;
43 799     std::string valstring;
44 800
45 801     int mt = 60;
46 802     int ht = mt*60;
47 803     int dt = ht*24;
48 804
49 805     if (runtime <= mt)
50 806         ss << runtime << " s";
51 807     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 HMCMC.

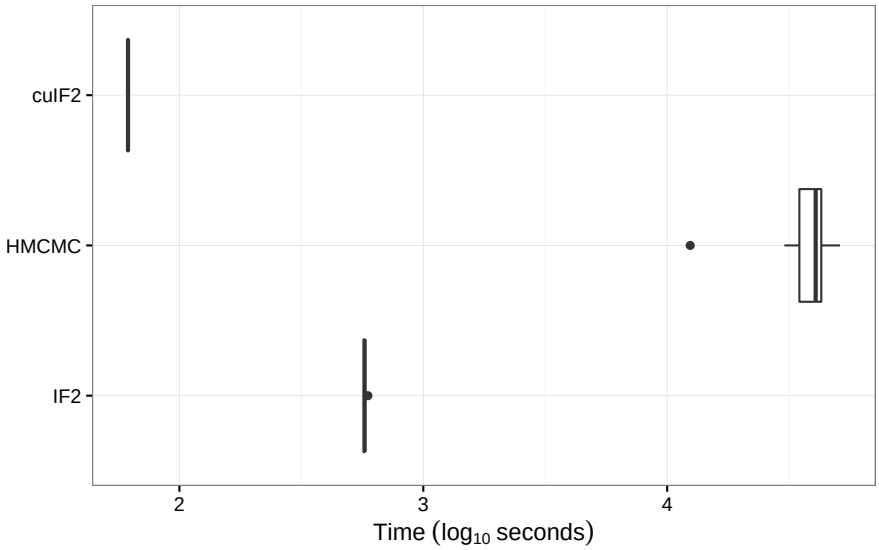

Figure F.1: Running times for fitting the spatial SIR model to data.

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 HMCMC. For `cuIF2` This is a speedup of over 9.33x against IF2 and over 617x against HMCMC.