

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, 147

# Abstract

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

# Acknowledgements

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

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

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

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

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

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Hamiltonian MCMC</b>	<b>5</b>
2.1	Markov Chains . . . . .	5
2.2	Likelihood . . . . .	6
2.3	Prior distribution . . . . .	7
2.4	Proposal distribution . . . . .	7
2.5	Algorithm . . . . .	8
2.6	Burn-in . . . . .	9
2.7	Thinning . . . . .	10
2.8	Hamiltonian Monte Carlo . . . . .	10
2.9	RStan Fitting . . . . .	13
<b>3</b>	<b>Iterated Filtering</b>	<b>18</b>
3.1	Formulation . . . . .	18
3.2	Algorithm . . . . .	19
3.3	Particle Collapse . . . . .	21
3.4	Iterated Filtering and Data Cloning . . . . .	21
3.5	Iterated Filtering 2 (IF2) . . . . .	22
3.6	IF2 Fitting . . . . .	24
<b>4</b>	<b>Parameter Fitting</b>	<b>26</b>
4.1	Fitting Setup . . . . .	26
4.2	Calibrating Samples . . . . .	29
4.3	IF2 Fitting . . . . .	30
4.4	IF2 Convergence . . . . .	32
4.5	IF2 Densities . . . . .	34
4.6	HMC Fitting . . . . .	35
4.7	HMC Densities . . . . .	35
4.8	HMC and Bootstrapping . . . . .	36
4.9	Multi-trajectory Parameter Estimation . . . . .	37

<b>5</b>	<b>Forecasting Frameworks</b>	<b>39</b>
5.1	Data Setup . . . . .	39
5.2	IF2 . . . . .	40
5.2.1	Parametric Bootstrapping . . . . .	41
5.2.2	IF2 Forecasts . . . . .	41
5.3	HMC . . . . .	43
5.4	Truncation vs. Error . . . . .	44
<b>6</b>	<b>S-map and SIRS</b>	<b>46</b>
6.1	S-maps . . . . .	46
6.2	S-map Algorithm . . . . .	47
6.3	SIRS Model . . . . .	49
6.4	SIRS Model Forecasting . . . . .	51
<b>7</b>	<b>Spatial Epidemics</b>	<b>54</b>
7.1	Spatial SIR . . . . .	54
7.2	Dewdrop Regression . . . . .	56
7.3	Spatial Model Forecasting . . . . .	57
<b>8</b>	<b>Discussion and Future Directions</b>	<b>59</b>
8.1	Parallel and Distributed Computing . . . . .	59
8.2	IF2, Bootstrapping, and Forecasting Methodology . . . . .	62
8.3	Fin . . . . .	63
<b>A</b>	<b>Hamiltonian MCMC</b>	<b>69</b>
A.1	Full R code . . . . .	69
A.2	Full Stan code . . . . .	72
<b>B</b>	<b>Iterated Filtering</b>	<b>74</b>
B.1	Full R code . . . . .	74
B.2	Full C++ code . . . . .	76
<b>C</b>	<b>Parameter Fitting</b>	<b>87</b>
C.1	SIR Forward Simulator . . . . .	87
<b>D</b>	<b>Forecasting Frameworks</b>	<b>89</b>
D.1	IF2 Parametric Bootstrapping Function . . . . .	89
D.2	RStan Forward Simulator . . . . .	91
<b>E</b>	<b>S-map and SIRS</b>	<b>93</b>
E.1	SIRS R Function Code . . . . .	93
E.2	SIRS HMC R Function Code . . . . .	94
E.3	SMAP Code . . . . .	96
E.4	SMAP Parameter Optimization Code . . . . .	97

E.5	RStan SIRS Code . . . . .	99
E.6	IF2 SIRS Code . . . . .	101
<b>F</b>	<b>Spatial Epidemics</b>	<b>114</b>
F.1	Spatial SIR R Function Code . . . . .	114
F.2	Spatial SIR HMC R Function Code . . . . .	116
F.3	RStan Spatial SIR Code . . . . .	118
F.4	IF2 Spatial SIR Code . . . . .	120
F.5	CUDA IF2 Spatial Fitting Code . . . . .	130



# List of Figures

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]	6
2.2	True SIR ODE solution infected counts, and with added observation noise.	14
2.3	Traceplot of samples drawn for parameter $R_0$ , excluding burn-in.	15
2.4	Traceplot of samples drawn for parameter $R_0$ , including burn-in.	16
2.5	Kernel density estimates produced by Stan. Dashed lines show true parameter values.	17
3.1	Kernel estimates for four essential system parameters. True values are indicated by dashed lines.	25
4.1	Simulated geometric autoregressive process shown in Equation [4.1].	27
4.2	Density plot of values shown in Figure[4.1].	28
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 $\epsilon_{\text{obs}} = \mathcal{N}(0, 10)$ , and the grey ribbon is centre 95th quantile from 100 random trajectories.	28
4.4	Fitting errors.	31
4.5	True system trajectory (solid line), observed data (dots), and IF2 estimated real state (dashed line).	31
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.	33
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.	33
4.8	The solid grey lines show the true parameter values and the dashed grey lines show the density medians.	35

4.9	As before, the solid grey lines show the true parameter values and the dashed grey lines show the density medians. . . . .	36
4.10	Result from 100 HMC bootstrap trajectories. The solid line shows the true states, the dots show the data, the dotted line shows the average system behaviour, the dashed line shows the bootstrap mean, and the grey ribbon shows the centre 95th quantile of the bootstrap trajectories. . . . .	37
4.11	IF2 point estimate densities are shown in black and HMC point estimate densities are shown in grey. The vertical lines show the true parameter values. . . . .	38
4.12	Fitting times for IF2 and HMC, in seconds. The centre box in each plot shows the centre 50th percent, with the bold centre line showing the median. . . . .	38
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 $\mathcal{N}(0, 10)$ . . . . .	40
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. . . . .	41
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 $\mathcal{N}(0, \sigma)$ . . . . .	42
5.4	Forecast produced by the HMC / bootstrapping framework with $M = 200$ trajectories. The dotted line shows the mean estimate of the forecasts, and the grey ribbon shows the 95% confidence interval on the estimated true states as described in Figure [5.3]. . . . .	44
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. . . . .	45
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. . . . .	50
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 $\theta = 3$ . . . . .	51
6.3	Error as a function of forecast length. . . . .	52

6.4	Runtimes for producing SIRS forecasts. The box shows the middle 50th percent, the bold line is the median, and the dots are outliers. Note that these are not “true” outliers, simply ones outside a threshold based on the interquartile range. . . . .	53
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 $\phi = 0.5$ . . . . .	55
7.2	Evolution of a spatial epidemic as in Figure [7.1], with added observation noise drawn from $\mathcal{N}(0, 10)$ . . . . .	56
7.3	Average SSE (log scale) across each location and all trials as a function of the number of weeks ahead in the forecast. . . . .	58
7.4	Runtimes for producing spatial SIR forecasts. The box shows the middle 50th percent, the bold line is the median, and the dots are outliers. . . . .	58
F.1	Running times for fitting the spatial SIR model to data. . . . .	147

# Chapter 1

## Introduction

Epidemic forecasting is an important tool that can help inform public policy and decision-making in the face of an infectious disease outbreak [9][29][41]. Successful intervention relies on accurate predictions of the number of cases, when they will occur, and where. Without this information it is difficult to efficiently allocate resources, a critical step in curbing the size and duration of an epidemic.

Despite the importance of reliable forecasts, obtaining them remains a challenge from both a theoretical and practical standpoint [29]. 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 [35], thus observational uncertainty must be built into mathematical models of disease spread from the beginning. Models must differentiate between natural variation in the intensity of disease spread (process error) and error in data collection (observation error) in order to accurately determine the dynamics underlying a data set, adding another layer of complexity [24]. With these caveats and concerns acknowledged, we can turn to a discussion of technique.

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

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

1 model. ARIMA assumes a linear underlying process and Gaussian error distributions.  
 2 It uses three parameters representing the degree of autoregression ( $p$ ), integration  
 3 (trend removal) ( $d$ ), and the moving average ( $q$ ), where the orders of the autoregression  
 4 and the moving average are determined through the use of an autocorrelation function  
 5 (ACF) and partial autocorrelation function (PACF), respectively, applied to the data  
 6 *a priori* [42].

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

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

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

26 Combining the phenomenological and mechanistic approaches are data integration  
 27 schemes. These methods use a model to define the expected underlying dynamics  
 28 of the system, but integrate data into the model in order to refine estimates of the  
 29 model parameters and produce more accurate forecasts. Typically the first step in  
 30 implementing such a technique is fitting the desired model to existing data. There are  
 31 many ways to do this, most of which fall into two main categories: Sequential Monte  
 32 Carlo (SMC) methods [3][33][41], and Markov chain Monte Carlo-based (MCMC)  
 33 methods [2][28]. From there data can either be integrated into the model by refitting  
 34 the model to the new longer data set, or in an “on-line” fashion in which data points

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

5 Another, broader, distinction among techniques can be drawn between those that rely  
6 on assumptions of linearity, and those that make no such assumption. As epidemic  
7 dynamics are highly non-linear, it can be questionable as to even consider linear ap-  
8 proaches to epidemic forecasting at all. In particular, stalwart approaches such as  
9 ARIMA and the venerable Kalman filter face a distinct (at least theoretical) disad-  
10 vantage in the face of newer SMC-based methods [38][41]. Extensions of the Kalman  
11 filter, such as the ensemble adjustment Kalman filter are designed to handle non-  
12 linearity, but these methods are very well-studied, and further work showing their  
13 viability would likely prove extraneous in the modern academic landscape.

14 Somewhat frustratingly, there exists no “gold standard” in forecasting [9][29][41]. As  
15 methodology varies widely in theoretical justification, implementation, and operation,  
16 it is difficult to compare the state of the art in forecasting methods from a first-  
17 principles perspective. Further, published work making use of any of these methods  
18 to forecast uses different prediction accuracy metrics, such as the sum of squared  
19 errors of prediction (SSE), peak time/duration/intensity, correlation tests, or root-  
20 mean-square error of prediction (RMSE), among others [9][30]. Thus it is difficult to  
21 select the best tool for the job when faced with a forecasting problem.

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

28 From MCMC-based methods we have selected Hamiltonian MCMC (HMC) [28], a  
29 (slightly) less cutting-edge but nonetheless highly effective technique. We are us-  
30 ing HMC through an implementation in the R package RStan [8], which at its core  
31 uses HMC, but also contains implementations of several other innovative techniques.  
32 Interestingly, the original goal of this package was not to implement a statistical  
33 programming language similar to Just Another Gibbs Sampler (JAGS) or Bayesian  
34 Inference Using Gibbs Sampling (BUGS), but with an HMC backend. In fact the  
35 developers’ original goal was to implement any method that could fit multilevel hi-  
36 erarchical models without halting as they were witnessing with BUGS and JAGS.  
37 Only after experimenting with several options and starting to hear about it more and  
38 more frequently did they attempt to work with HMC. In the end, the scope of the  
39 project grew to include the development and subsequent integration of the No-U-Turn  
40 Sampler (NUTS) [19], and an implementation of automatic differentiation machinery

1 [36].

2 For PF-based methods we have selected Iterated Filtering (IF) 2 [22], a very recently  
3 developed approach that uses multiple particle filtering rounds to generate maximum  
4 likelihood estimates (MLEs). It functions similarly to its predecessor, the Maximum  
5 likelihood via Iterated Filtering (MIF) algorithm [21], but aims to be simpler, faster,  
6 and more accurate. Theoretical justification and synthetic testing indicates that IF2  
7 meets these goals, and as such the authors recommend skipping MIF and jumping  
8 straight to IF2 if an algorithm of their variety is sought. And so, we are doing just  
9 that. We wrote our own IF2 implementation in C++ and integrated it into R using  
10 the Rcpp package [13]. The developers of MIF and IF2 have their own R package that  
11 implements MIF and IF2, Partially Observed Markov Processes (POMP) [23][25], but  
12 it didn't provide some of the diagnostic information we needed, so it was not used  
13 here.

14 Finally, from the phenomenological methods we have selected the sequential locally  
15 weighted global linear maps (S-map) [15][20][37][38], combined with Dewdrop Regres-  
16 sion [15]. These methods stand on their own as a unique take on the forecasting  
17 problem, and bear little resemblance to other methodology. The virtues of these tech-  
18 niques have been long-extolled by their developers, but their efficacy when compared  
19 to competing methods has not been well-studied. This work will mark one of the first  
20 times this has been done.

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

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

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

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



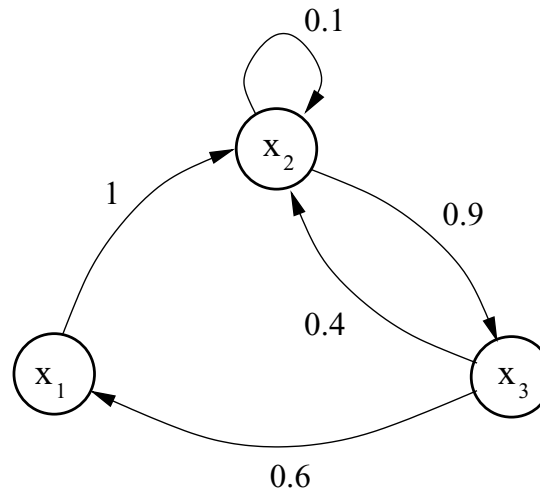


Figure 2.1: A finite state machine. States are shown as graph nodes, and the probability of transitioning from one particular state to another is shown as a weighted graph edge. [2]

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

3 This property holds when the chain satisfies the following conditions

- 4 • *Irreducible* Any state A can be reached from any other state B with non-zero  
 5 probability
- 6 • *Positive Recurrent* The number of steps required for the chain to reach state A  
 7 from state B must be finite
- 8 • *Aperiodic* The chain must be able to explore the parameter space without be-  
 9 coming trapped in a cycle

10 Note that MCMC sampling generates a Markov chain  $(\theta^{(1)}, \theta^{(2)}, \dots, \theta^{(N)})$  that does  
 11 indeed satisfy these conditions, and uses the chain's equilibrium distribution to ap-  
 12 proximate the posterior distribution of the parameter space [2].

## 13 2.2 Likelihood

14 MCMC and similar methods hinge on the idea that the weight or support bestowed  
 15 upon a particular set of parameters  $\theta$  should be proportional to the probability of  
 16 observing the data  $D$  given the model output using that set of parameters  $M(\theta)$ . In  
 17 order to do this we need a way to evaluate whether or not  $M(\theta)$  is a good fit for  $D$ ;

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

$$2 \qquad \qquad \qquad \mathcal{L}(\theta) \propto P(D|\theta). \qquad (2.3)$$

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

## 8 **2.3 Prior distribution**

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

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

## 28 **2.4 Proposal distribution**

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

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

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

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

## 9 **2.5 Algorithm**

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

12 We will denote the previously discussed quantities as

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

17 and then define the acceptance ratio,  $r$ , as

$$18 \quad r = \frac{\mathcal{L}(\theta^*)p(\theta^*)q(\theta^*|\theta)}{\mathcal{L}(\theta)p(\theta)q(\theta|\theta^*)}, \quad (2.4)$$

19 where  $\theta^*$  is the proposed sample to draw from the posterior, and  $\theta$  is the last accepted  
 20 sample. This is known as the Metropolis-Hastings rule.

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

$$23 \quad r = \frac{\mathcal{L}(\theta^*)p(\theta^*)}{\mathcal{L}(\theta)p(\theta)}. \quad (2.5)$$

24 Algorithm [1] shows the Metropolis MCMC algorithm.

25 In this way we are ensuring that steps that lead to better likelihood outcomes are  
 26 likely to be accepted, but steps that do not will not be accepted as frequently. Note  
 27 that these less "advantageous" moves will still occur but that this is by design – it

---

**Algorithm 1:** Metropolis MCMC

---

```

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

```

---

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

## 3 2.6 Burn-in

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

9 Burning in a chain is the act of running the MCMC algorithm normally without saving  
10 first  $M$  samples. As we are seeking a chain of length  $N$ , the total computation will  
11 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  $n^{\text{th}}$  step, which should still give a decent approximate of the posterior (since the chain has time to explore a large portion of the parameter space), but requires less room to store [26].

## 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) [28]. Prior to the advent of HMC, some work was conducted exploring adaptive step-sizing using MCMC-based methods, but found they lack strong theoretical justification, and can lead to some samples being drawn from an incorrect distribution [28]. HMC has in fact existed for nearly the same amount of time as MCMC – both methods having been developed to model molecular dynamics, with MCMC taking a probabilistic approach and HMC taking a more deterministic one – but had not received much attention outside its native discipline until recently.

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

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

Kinetic energy will serve as a way to “nudge” the parameters along a different moment for each component of  $\theta$ . We introduce  $n$  auxiliary variables  $r = (r_1, r_1, \dots, r_n)$ , where  $n$  is the number of components in  $\theta$ . 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

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

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

4 The Hamiltonian of the system is defined as

$$5 \quad H(\theta, r) = U(\theta) + K(r), \quad (2.8)$$

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

$$8 \quad \begin{aligned} \frac{d\theta}{dt} &= M^{-1}r \\ \frac{dr}{dt} &= -\nabla U(\theta). \end{aligned} \quad (2.9)$$

9 It is tempting to try to integrate this system using the standard Euler evolution  
 10 scheme, but in practice this leads to instability as it will not preserve the volume of  
 11 the system. Instead the “Leapfrog” scheme is used. This scheme is very similar to  
 12 Euler scheme, except instead of using a fixed step size  $h$  for all evolutions, a step size  
 13 of  $\epsilon$  is used for most evolutions, with a half step size of  $\epsilon/2$  for evolutions of  $\frac{dr}{dt}$  at  
 14 the first step, and last step  $L$ . In this way the evolution steps “leapfrog” over each  
 15 other while using future values from the other set of steps, leading to the scheme’s  
 16 name.

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

$$20 \quad r = \exp [H(\theta, r) - H(\theta^*, r^*)], \quad (2.10)$$

21 where  $(\theta, r)$  are the last values in the chain. This form of the acceptance ratio comes  
 22 from the definition of the Hamiltonian as an energy function. If we define the distribu-  
 23 tion of the total potential energy in the system (known as the canonical distribution)  
 24 as a function of the Hamiltonian as

$$25 \quad P(\theta, r) = \frac{1}{Z} \exp(-H(\theta, r)) \quad (2.11)$$

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

---

**Algorithm 2:** Hamiltonian MCMC

---

```

/* Select a starting point */
Input : Initialize  $\theta^{(1)}$ 
1 for  $i = 2 : N$  do
    /* Resample moments */
    2 for  $i = 1 : n$  do
    3    $r(i) \leftarrow \mathcal{N}(0, 1)$ 

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

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

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

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

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

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

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

```

---

1 Together, we have Algorithm [2].  
 2 Note that the parameters  $\epsilon$  and  $L$  have to be tuned in order to maintain stability  
 3 and maximize efficiency, a sometimes non-trivial process utilising trial fitting with  
 4 candidate values of  $\epsilon$  and  $L$  [28]. However, some recent algorithms, such as the  
 5 No U-Turn sampler implemented in RStan, and adaptively select appropriate values  
 6 automatically during the sampling process [19].

## 7 **2.9 RStan Fitting**

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

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

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

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

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

23 Figure [2.2] shows the system simulation results.

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

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



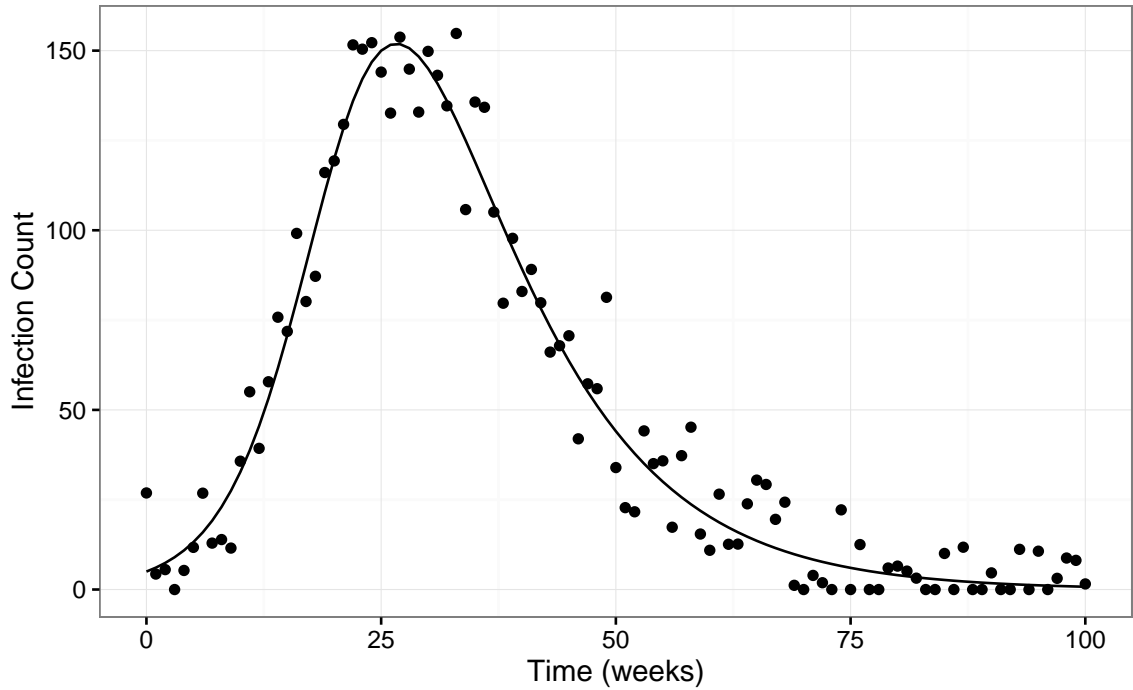


Figure 2.2: True SIR ODE solution infected counts, and with added observation noise.

- 1 In order to use an Explicit Euler-like stepping method in the later Stan model, the
- 2 synthetic observation counts were treated as weekly observations in which the counts
- 3 on the other six days of the week were unobserved.
- 4 Figure [2.3] shows the traceplot for the the post-burn-in chain data returned by the
- 5 RStan fitting. We see that the chains are mixing well and convergence has likely been
- 6 reached.
- 7 Figure [2.4] shows the chain data including the burn-in samples. We can see why it
- 8 is wise to discard these samples (note the scale).
- 9 Figure [2.5] shows the the kernel density estimates for each of the model parameters
- 10 and the initial number of cases. We see that while the estimates are not perfect, they
- 11 are more than satisfactory.

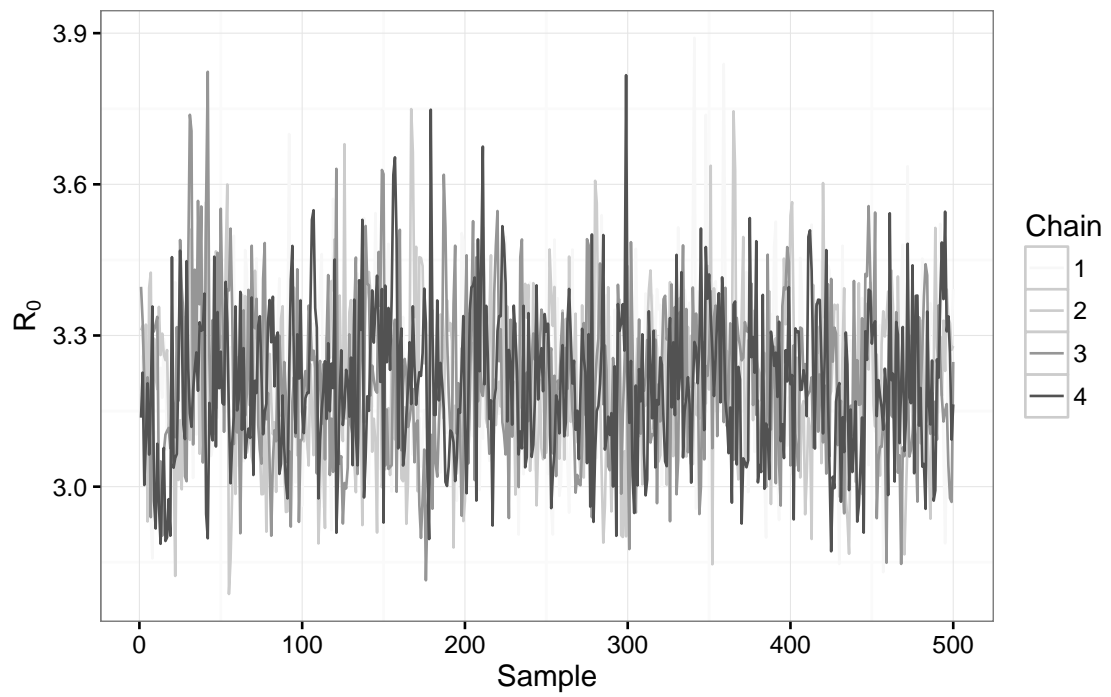


Figure 2.3: Traceplot of samples drawn for parameter  $R_0$ , excluding burn-in.

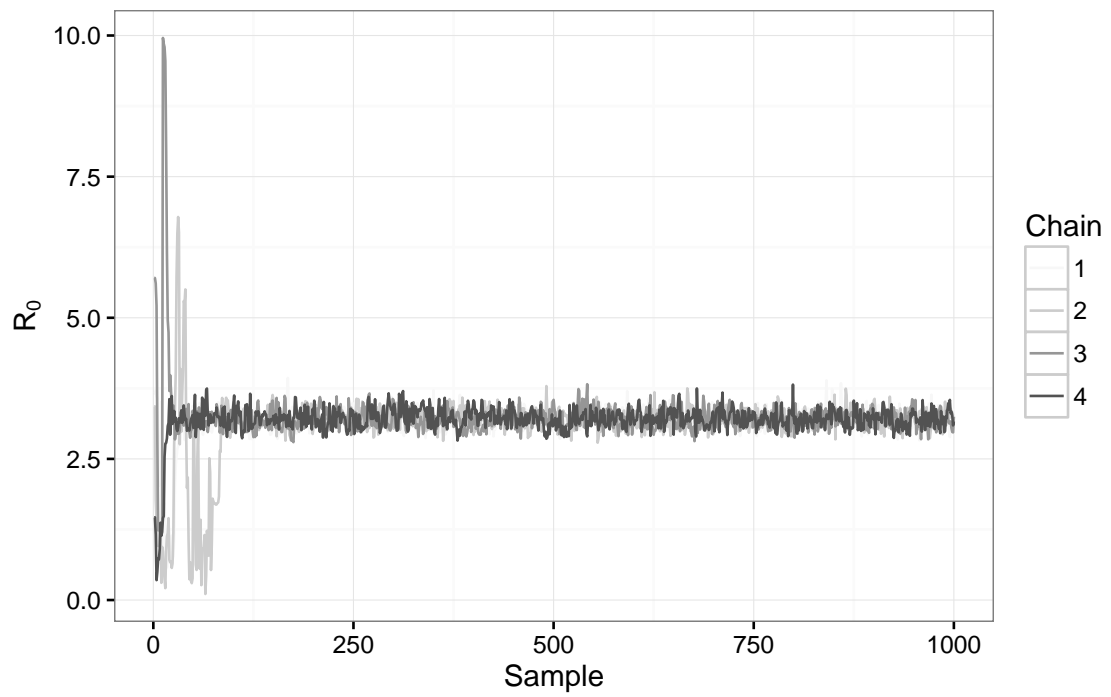


Figure 2.4: Traceplot of samples drawn for parameter  $R_0$ , including burn-in.

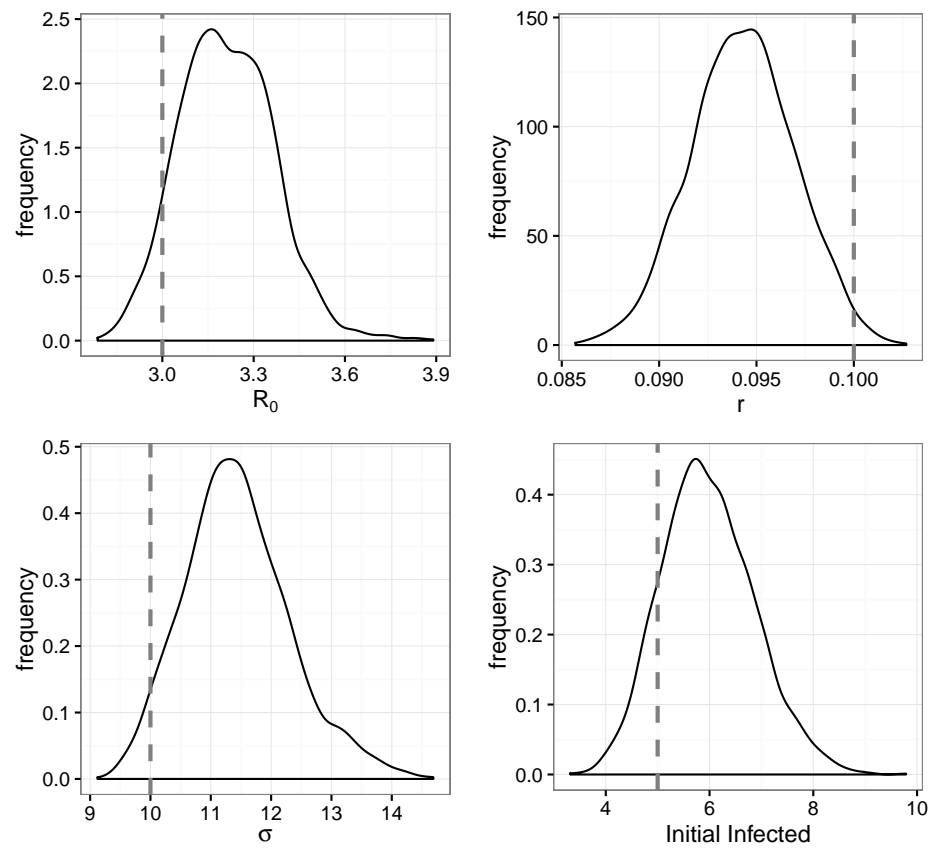


Figure 2.5: Kernel density estimates produced by Stan. Dashed lines show true parameter values.

# Chapter 3

## Iterated Filtering

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

Instead of constructing a Markov chain and approximating its stationary distribution, a cohort of “particles” are used to move through the data in an on-line (sequential) fashion with the cohort being culled of poorly-performing particles at each iteration via importance sampling. If the culled particles are not replenished, this will be a Sequential Importance Sampling (SIS) particle filter. If the culled particles are replenished from surviving particles, in a sense setting up a process analogous to Darwinian selection, then this will be a Sequential Importance Resampling (SIR) particle filter [3].

### 3.1 Formulation

Particle filters, also called Sequential Monte-Carlo (SMC) filters, feature similar core functionality as the venerable Kalman Filter. As the algorithm moves through the data (sequence of observations), a prediction-update cycle is used to simulate the evolution of the model  $M$  with different particular parameter selections, track how closely these predictions approximate the new observed value, and update the current cohort appropriately [3].

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

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

1 And the observation process by

$$2 \qquad Y_t \sim f_2(X_t, \theta). \qquad (3.2)$$

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

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

## 13 **3.2 Algorithm**

14 Now we will formalize the particle filter.

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

18 The algorithm for a Sequential Importance Resampling particle is shown in Algorithm  
 19 [3].

---

**Algorithm 3:** SIR particle filter

---

```

/* Select a starting point */
Input : Observations  $D = y_1, y_2, \dots, y_T$ , initial particle distribution  $P_0$  of size
         $J$ 

/* Setup */
1 Initialize particle cohort by sampling  $(p^{(1)}, p^{(2)}, \dots, p^{(J)})$  from  $P_0$ 
2 for  $t = 1 : T$  do
    /* Evolve */
    3 for  $j = 1 : J$  do
    4    $X_t^{(j)} \leftarrow f_1(X_{t-1}^{(j)}, \theta^{(j)})$ 

    /* Weight */
    5 for  $j = 1 : J$  do
    6    $w^{(j)} \leftarrow P(y_t | X_t^{(j)}, \theta^{(j)}) = f_2(X_t^{(j)}, \theta^{(j)})$ 

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

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

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

```

---

### 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,  $N_{\text{eff}}$ , which represents the number of particles that are acceptably dissimilar. This is estimated by evaluating

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

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.



1 Rigorous proofs have been developed [21][22] that show that by treating the param-  
2 eters as stochastic processes instead of fixed values, the multiple passes through the  
3 data will indeed force convergence of the process mean toward maximum likelihood,  
4 and the process variance toward 0.

## 5 **3.5 Iterated Filtering 2 (IF2)**

6 The successor to Iterated Filtering 1 [21], Iterated Filtering 2 [22] is simpler, faster,  
7 and demonstrated better convergence toward maximum likelihood. The core concept  
8 involves a two-pronged approach. First, a data cloning-like procedure is used to  
9 allow more time for the parameter stochastic process means to converge to maximum  
10 likelihood, and frequent cooled perturbation of the particle parameters allow better  
11 exploration of the parameter space while still allowing convergence to good point  
12 estimates.

13 IF2 is not designed to estimate the full posterior distribution, instead to produce  
14 a Maximum Likelihood (ML) point estimate. Further, IF2 thwarts the problem of  
15 particle collapse by keeping at least some perturbation in the system at all times. It  
16 is important to note that while true particle collapse will not occur, there is still risk  
17 of a pseudo-collapse in which all particles will be extremely close to one another so as  
18 to be virtually indistinguishable. However this will only occur with the use of overly-  
19 aggressive cooling strategies or by specifying an excessive number of passes through  
20 the data.

21 An important new quantity is the particle perturbation density denoted  $h(\theta|\sigma)$ . Typ-  
22 ically this is multivariate Normal with  $\sigma$  being a vector of variances proportional to  
23 the expected values of  $\theta$ . In practice the proportionality can be derived from current  
24 means or specified ahead of time. Further, these intensities must decrease over time.  
25 This can be done via exponential or geometric cooling, a decreasing step function, a  
26 combination of these, or through some other similar scheme.

27 The algorithm for IF2 can be seen in Algorithm [4].

28

---

**Algorithm 4:** IF2

---

```

/* Select a starting point */
Input : Observations  $D = y_1, y_2, \dots, y_T$ , initial particle distribution  $P_0$  of size
         $J$ , decreasing sequence of perturbation intensity vectors  $\sigma_1, \sigma_2, \dots, \sigma_M$ 

/* Setup */
1 Initialize particle cohort by sampling  $(p^{(1)}, p^{(2)}, \dots, p^{(J)})$  from  $P_0$ 

/* Particle seeding distribution */
2  $\Theta \leftarrow P_0$ 
3 for  $m = 1 : M$  do
    /* Pass perturbation */
    4 for  $j = 1 : J$  do
    5      $p^{(j)} \sim h(\Theta^{(j)}, \sigma_m)$ 
    6 for  $t = 1 : T$  do
    7     for  $j = 1 : J$  do
    8         /* Iteration perturbation */
    9          $p^{(j)} \sim h(p^{(j)}, \sigma_m)$ 
    10        /* Evolve */
    11         $X_t^{(j)} \leftarrow f_1(X_{t-1}^{(j)}, \theta^{(j)})$ 
    12        /* Weight */
    13         $w^{(j)} \leftarrow P(y_t | X_t^{(j)}, \theta^{(j)}) = f_2(X_t^{(j)}, \theta^{(j)})$ 
    14        /* Normalize */
    15        for  $j = 1 : J$  do
    16             $w^{(j)} \leftarrow w^{(j)} / \sum_1^J w^{(j)}$ 
    17        /* Resample */
    18         $p^{(1:J)} \leftarrow \text{sample}(p^{(1:J)}, \text{prob} = w, \text{replace} = \text{true})$ 
    19        /* Collect particles for next pass */
    20        for  $j = 1 : J$  do
    21             $\Theta^{(j)} \leftarrow p^{(j)}$ 

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

```

---

## 1 3.6 IF2 Fitting

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

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

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

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

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

25 Figure [3.1] shows the final kernel estimates for four of the key parameters. As with  
26 HMC, the distributions are not perfect, but are promising. Unlike with HMC, these  
27 distributions are not meant to consist of samples from the true posterior distribution,  
28 but rather serve a diagnostic role.

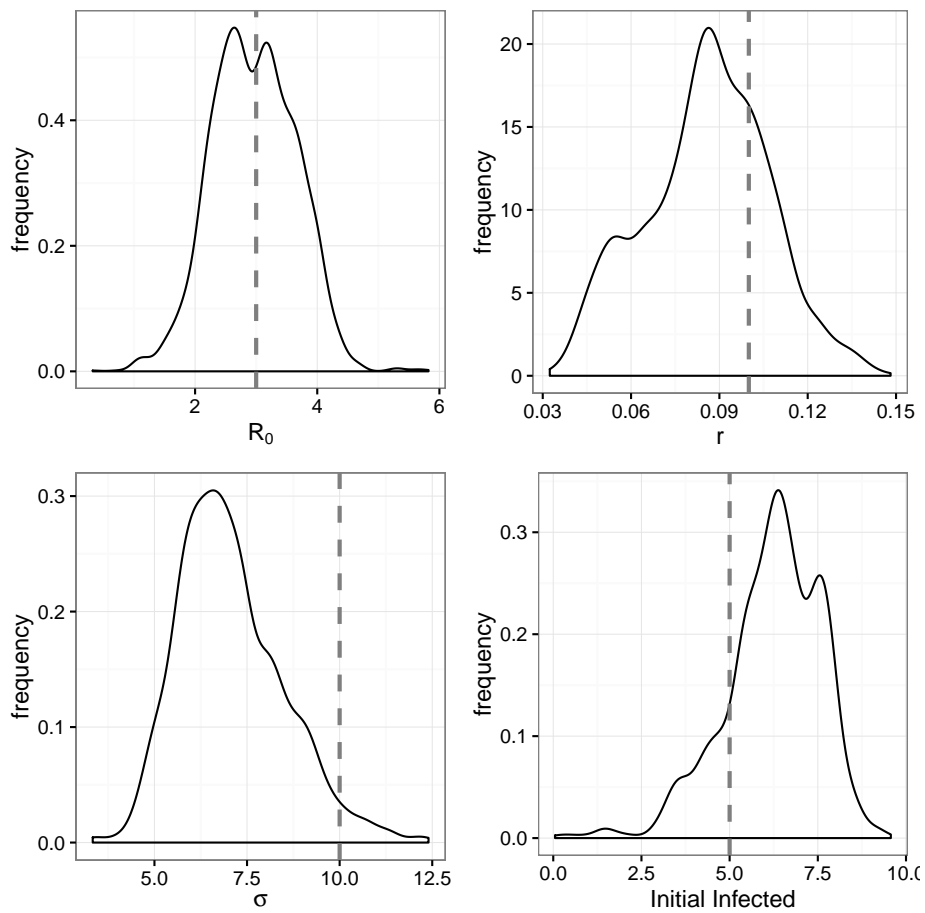


Figure 3.1: Kernel estimates for four essential system parameters. True values are indicated by dashed lines.

# 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  $\beta$  to follow a geometric random walk given by

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

We will take  $\epsilon_t$  to be normally distributed with standard deviation  $\rho^2$  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  $\beta_t$  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

$$X_{t+1} = c + \rho X_t + \epsilon_t, \quad (4.2)$$

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

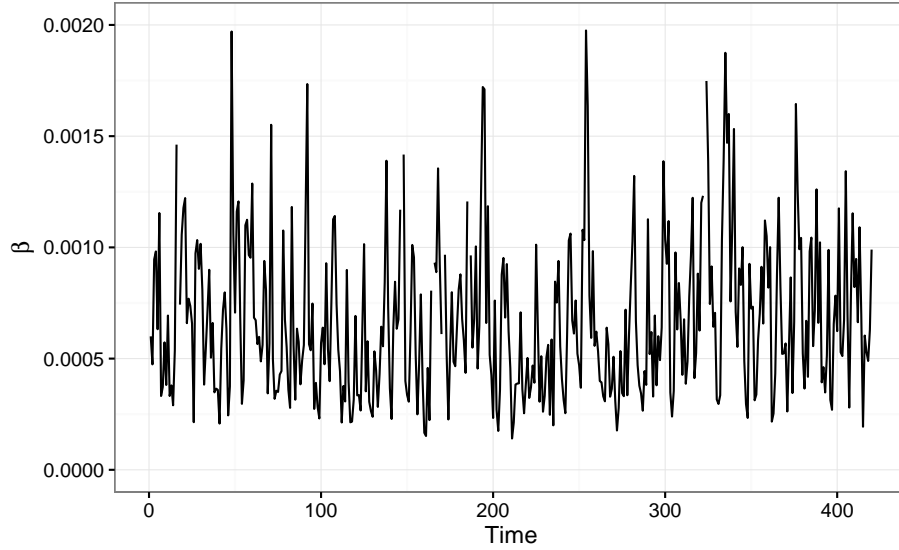


Figure 4.1: Simulated geometric autoregressive process shown in Equation [4.1].

- 1 If we choose  $\eta = 0.5$ , the resulting log-normal distribution has a mean of  $6.80 \times 10^{-4}$
- 2 and standard deviation of  $4.46 \times 10^{-4}$ .
- 3 Figure [4.1] shows the result of simulating the process in Equation [4.1] with  $\eta =$
- 4 0.5.
- 5 Figure [4.2] shows the density plot corresponding to the values in Figure [4.1].
- 6 We see a density plot similar in shape to the desired density, and the geometric random
- 7 walk displays dependence on previous values. Further the mean of this distribution
- 8 was calculated to be  $6.92 \times 10^{-4}$  and standard the deviation to be  $3.99 \times 10^{-4}$ , which
- 9 are very close to the theoretical values.
- 10 If we take the full stochastic SIR system and evolve it using an Euler stepping scheme
- 11 with a step size of  $h = 1/7$ , for 1 step per day, we obtain the plot in Figure [4.3].

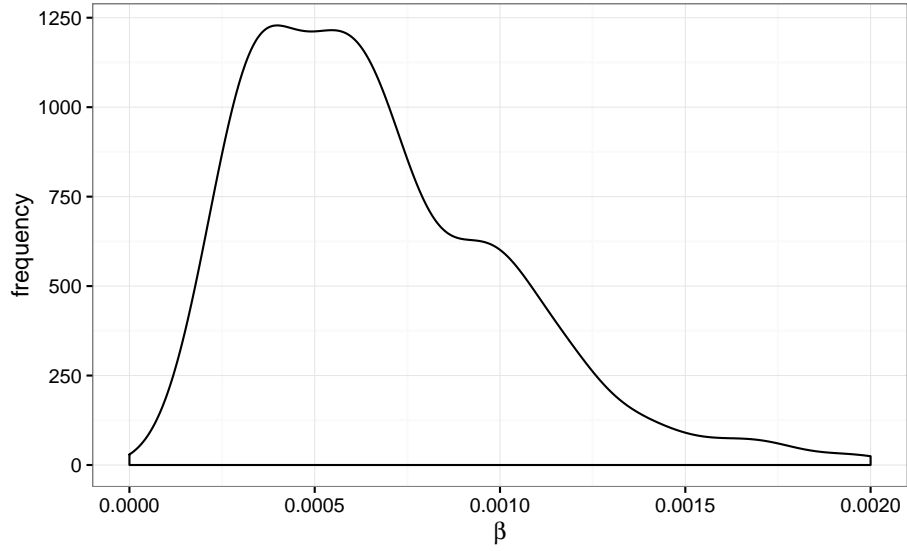


Figure 4.2: Density plot of values shown in Figure 4.1.

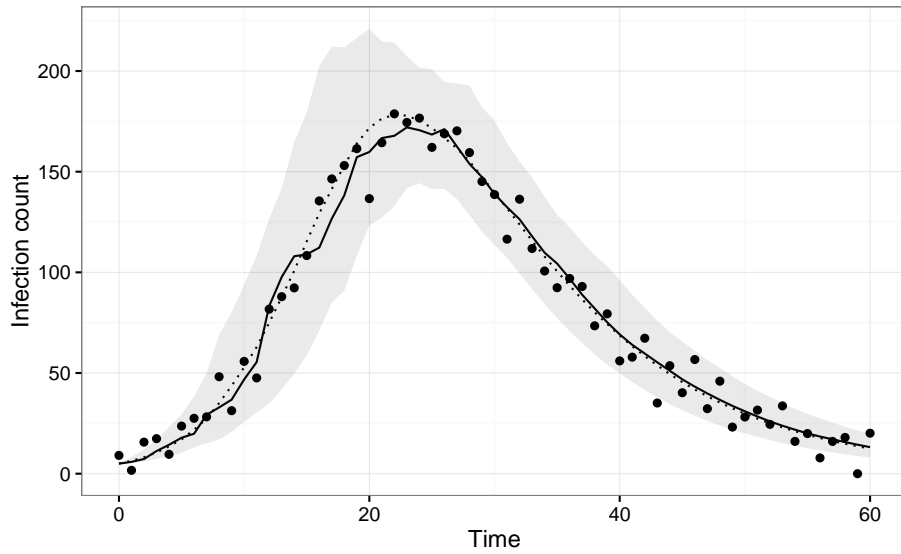


Figure 4.3: Stochastic SIR model simulated using an explicit Euler stepping scheme. The solid line is a single random trajectory, the dots show the data points obtained by adding in observation error defined as  $\epsilon_{\text{obs}} = \mathcal{N}(0, 10)$ , and the grey ribbon is centre 95th quantile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

## 1 4.2 Calibrating Samples

2 In order to compare HMC and IF2 we need to set up a fair and theoretically justified  
 3 way to select the number of samples to draw for the HMC iterations and the number  
 4 of particles to use for IF2. As we wish to compare, among other things, approximate  
 5 computational cost using runtimes, we need to determine how many sample draws for  
 6 each method are required to obtain a certain accuracy. Sample draws are typically  
 7 not comparable in terms of quality when considering multiple methods. For example,  
 8 vanilla MCMC draws are computationally cheap compared to those from HMC, but  
 9 HMC produces draws that more efficiently cover the sampling space. Thus we cannot  
 10 just set the number of HMC draws equal to the number of particles used in IF2 – we  
 11 must calibrate both quantities based on a desired target error. We assume that we  
 12 are working with a problem that has an unknown real solution, so we use the Monte  
 13 Carlo Standard Error (MCSE) [17].

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

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

$$21 \quad \hat{\sigma}_n = \sqrt{\frac{1}{n} \sum_{i=1}^n (X_i - \hat{\mu})^2}, \quad (4.3)$$

22 which is known as the Monte Carlo Standard Error.

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

$$25 \quad \Theta^* V (\Theta^*)^T \quad (4.4)$$

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

$$31 \quad \hat{\sigma}_n = \sqrt{\frac{1}{n} \frac{1}{P} \Theta^* V (\Theta^*)^T} \quad (4.5)$$



1 where  $P$  is the number of particles.

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

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

### 11 **4.3 IF2 Fitting**

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

17 The code used here is a mix of R and C++ implemented using Rcpp. The fitting  
18 was undertaken using 2500 particles with 50 IF2 passes and a cooling schedule given  
19 by a reduction in particle spread determined by  $0.975^p$ , where  $p$  is the pass number  
20 starting with 0. This geometric cooling scheme is standard for use with IF2 [23][25],  
21 with the cooling rate chosen to neatly scale the perturbation factor from 1 to 0.02  
22 (almost 0) over 50 passes.

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

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

Name	True	IF2		HMC	
		Fit	Error	Fit	Error
$R_0$	3.0	3.27	$9.08 \times 10^{-2}$	3.12	$1.05 \times 10^{-1}$
$r$	$10^{-1}$	$1.04 \times 10^{-1}$	$3.61 \times 10^{-2}$	$9.99 \times 10^{-2}$	$-7.56 \times 10^{-4}$
Initial Infected	5	7.90	$5.80 \times 10^{-1}$	6.64	$3.28 \times 10^{-1}$
$\sigma$	10	8.84	$-1.15 \times 10^{-1}$	8.5	$-1.50 \times 10^{-1}$
$\eta$	$5 \times 10^{-1}$	$5.87 \times 10^{-1}$	$1.73 \times 10^{-1}$	$4.57 \times 10^{-1}$	$-8.27 \times 10^{-2}$
$\varepsilon_{err}$	$5 \times 10^{-1}$	$1.63 \times 10^{-1}$	$-6.73 \times 10^{-1}$	$1.60 \times 10^{-1}$	$-6.80 \times 10^{-1}$

Figure 4.4: Fitting errors.

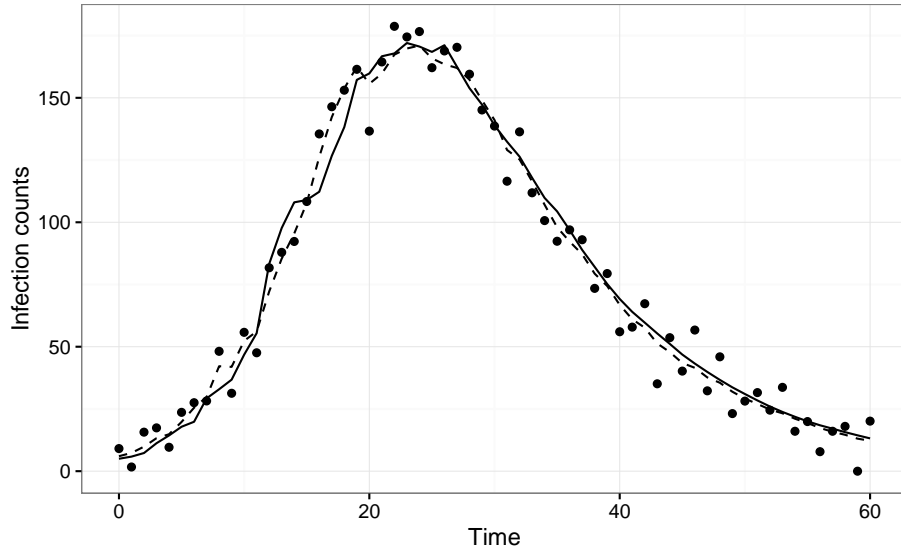


Figure 4.5: True system trajectory (solid line), observed data (dots), and IF2 estimated real state (dashed line).

## 1 4.4 IF2 Convergence

2 Since IF2 is an iterative algorithm where each pass through the data is expected to  
3 push the parameter estimates towards the MLE, we can see the evolution of these es-  
4 timates as a function of the pass number. We expect near-convergence in the param-  
5 eter estimates as IF2 nears the maximum number of passes specified. Unconvincing  
6 convergence plots may signal suboptimal algorithm parameters. If sensible algorithm  
7 parameters have been chosen, we should see the convergence plots display “flattening”  
8 over time.

9 Figure [4.6] shows evolution of the mean estimates for the six most critical paramete-  
10 rs.

11 Figure [4.7] shows the evolution of the standard deviations of the parameter estimates  
12 from the particle swarm as a function of the pass number. We should expect to see  
13 asymptotic convergence to zero if the filter is converging.

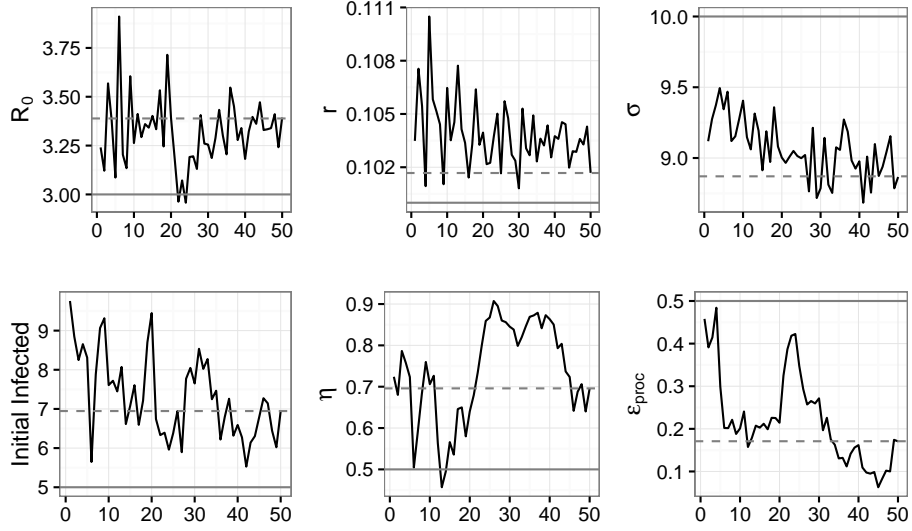


Figure 4.6: The horizontal axis shows the IF2 pass number. The solid black lines show the evolution of the ML estimates, the solid grey lines show the true value, and the dashed grey lines show the mean parameter estimates from the particle swarm after the final pass.

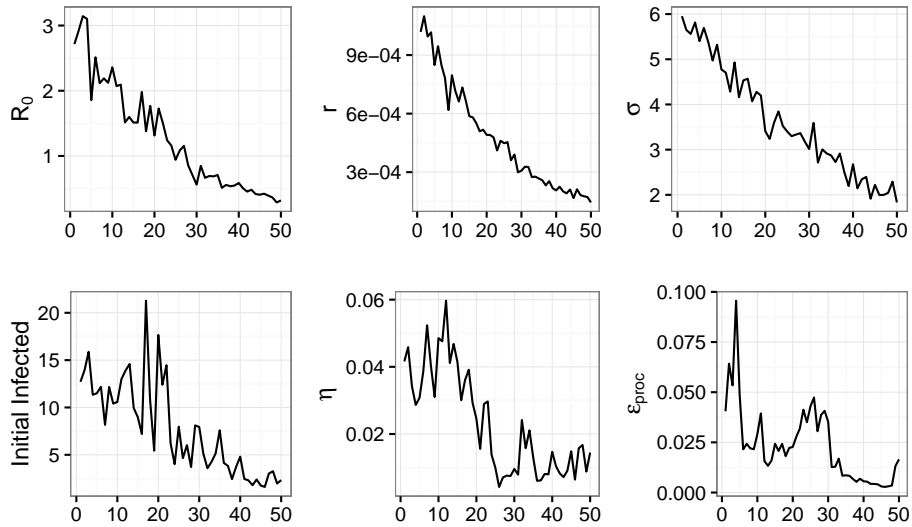


Figure 4.7: The horizontal axis shows the IF2 pass number and the solid black lines show the evolution of the standard deviations of the particle swarm values.

## 1 4.5 IF2 Densities

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

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

7 The IF2 parameters chosen were in part chosen so as to not artificially narrow these  
8 densities; a more aggressive cooling schedule and/or an increased number of passes  
9 would have resulted in much narrower densities, and indeed have the potential to  
10 collapse them to point estimates. This is undesirable as it may indicate instability –  
11 the particles may have become “trapped” in a region of the sampling space.

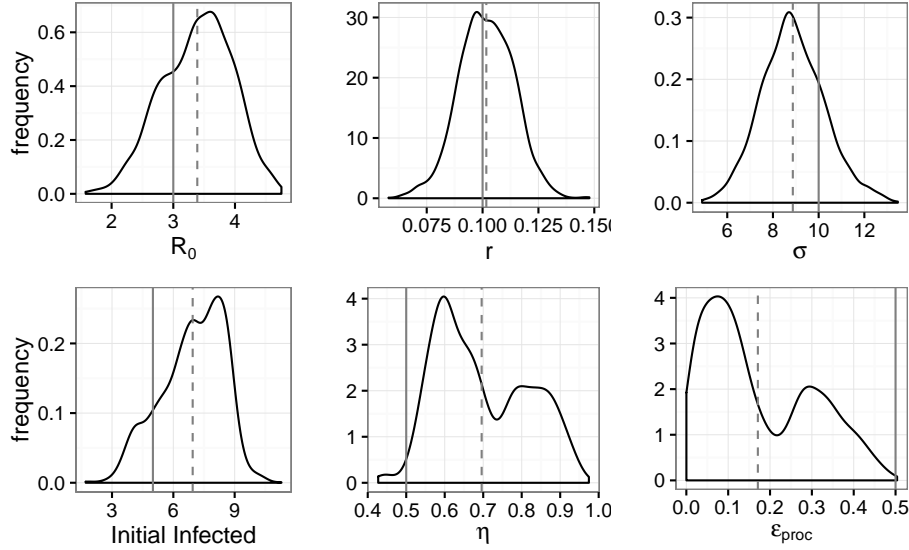


Figure 4.8: The solid grey lines show the true parameter values and the dashed grey lines show the density medians.

## 4.6 HMC Fitting

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

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

## 4.7 HMC Densities

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

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

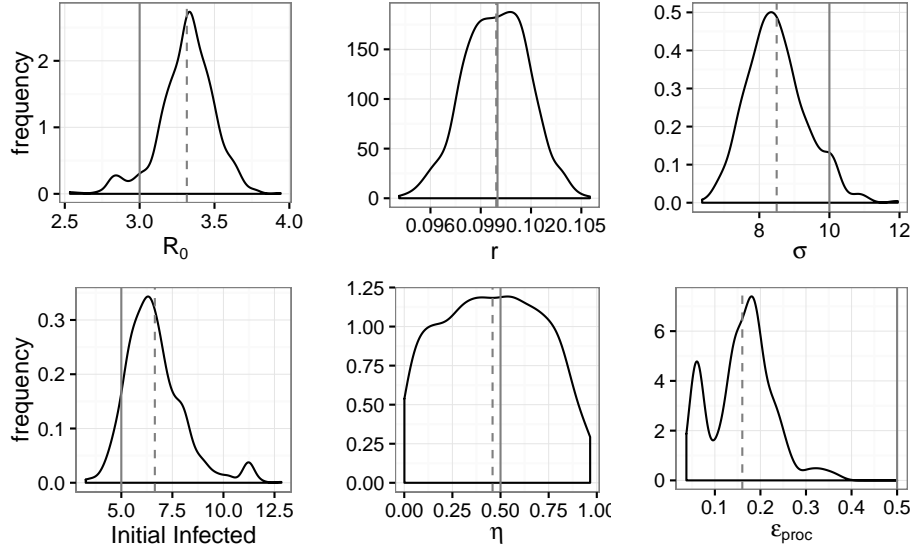


Figure 4.9: As before, the solid grey lines show the true parameter values and the dashed grey lines show the density medians.

## 1 4.8 HMC and Bootstrapping

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

7 Figure [4.10] shows the results of 100 bootstrap trajectories generated from the RStan  
 8 HMC samples.

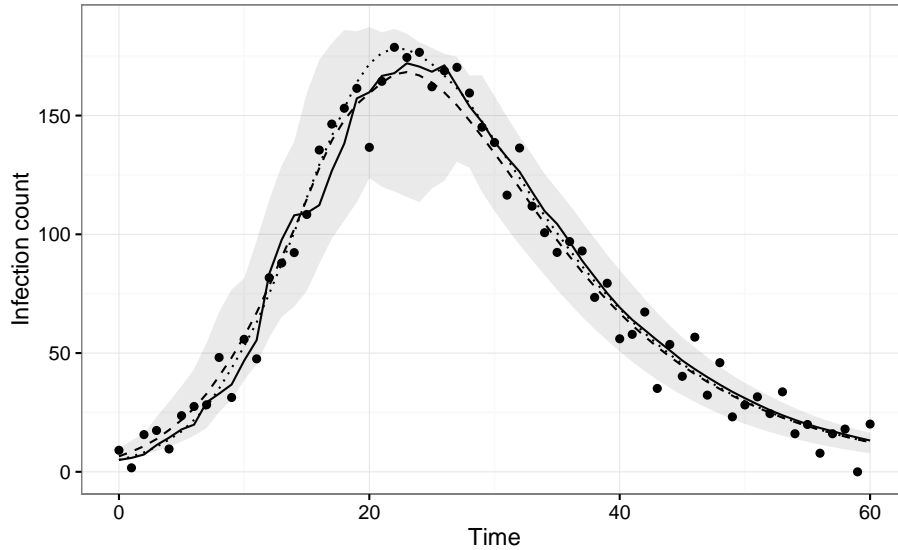


Figure 4.10: Result from 100 HMC bootstrap trajectories. The solid line shows the true states, the dots show the data, the dotted line shows the average system behaviour, the dashed line shows the bootstrap mean, and the grey ribbon shows the centre 95th quantile of the bootstrap trajectories.

## 1 4.9 Multi-trajectory Parameter Estimation

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

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

6 The densities by and large display similar coverage, with the IF2 densities for  $r$  and  
 7  $\varepsilon_{proc}$  showing slightly wider coverage than the HMC densities for the same param-  
 8 eters.

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

10 The average running times were approximately 45.5 seconds and 257.4 seconds for IF2  
 11 and HMC respectively, representing a 5.7x speedup for IF2 over HMC. While IF2 may  
 12 be able to fit the model to data faster than HMC, we are obtaining less information;  
 13 this will become important in the next section. Further, the results in Figure [4.12]  
 14 show that while the running time for IF2 is relatively fixed, the times for HMC are  
 15 anything but, showing a wide spread of potential times.



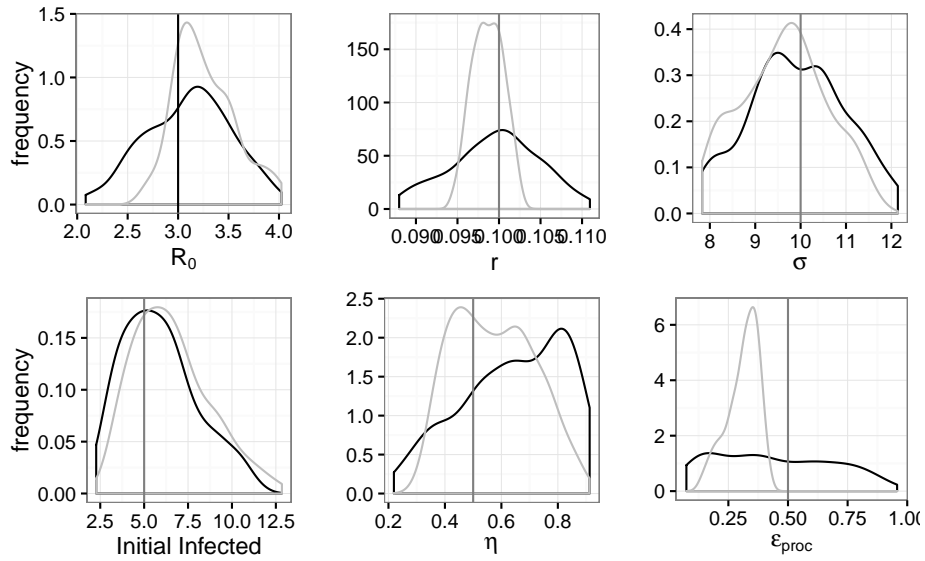


Figure 4.11: IF2 point estimate densities are shown in black and HMC point estimate densities are shown in grey. The vertical lines show the true parameter values.

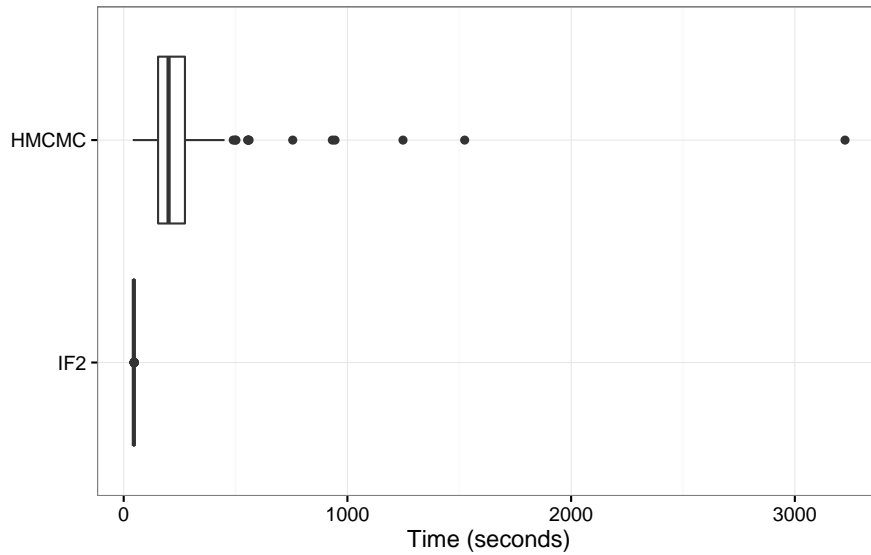


Figure 4.12: Fitting times for IF2 and HMC, in seconds. The centre box in each plot shows the centre 50th percent, with the bold centre line showing the median.

# **Chapter 5**

## **Forecasting Frameworks**

### **5.1 Data Setup**

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

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

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

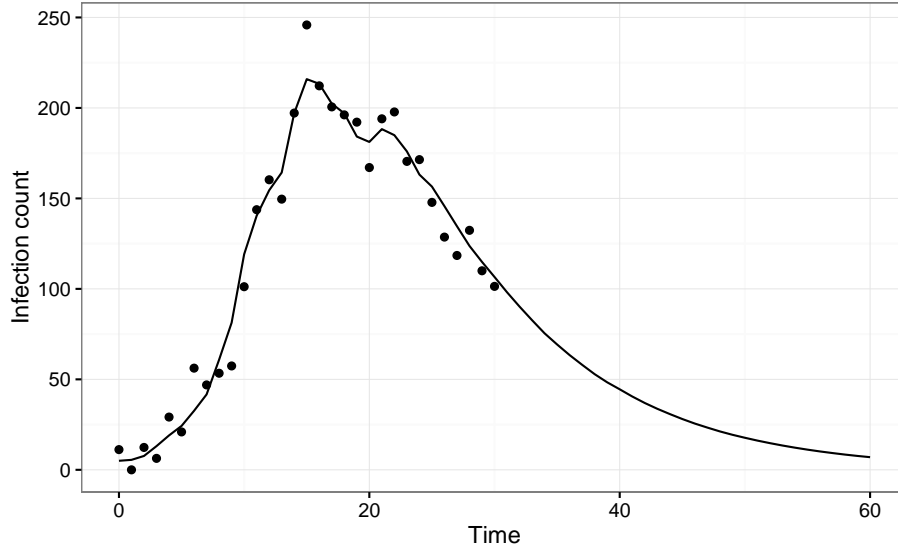


Figure 5.1: Infection count data truncated at  $T = 30$ . The solid line shows the true underlying system states, and the dots show those states with added observation noise. Parameters used were  $R_0 = 3.0$ ,  $r = 0.1$ ,  $\eta = .05$ ,  $\sigma_{proc} = 0.5$ , and additive observation noise was drawn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

## 1 5.2 IF2

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

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

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

13 Recall that IF2 is not trying to generate posterior probability densities, but rather  
 14 produce a point estimate. Since we wish to determine the approximate distribution of  
 15 each of the parameters in addition to the point estimate, we must add another layer  
 16 atop the IF2 machinery, parametric bootstrapping.

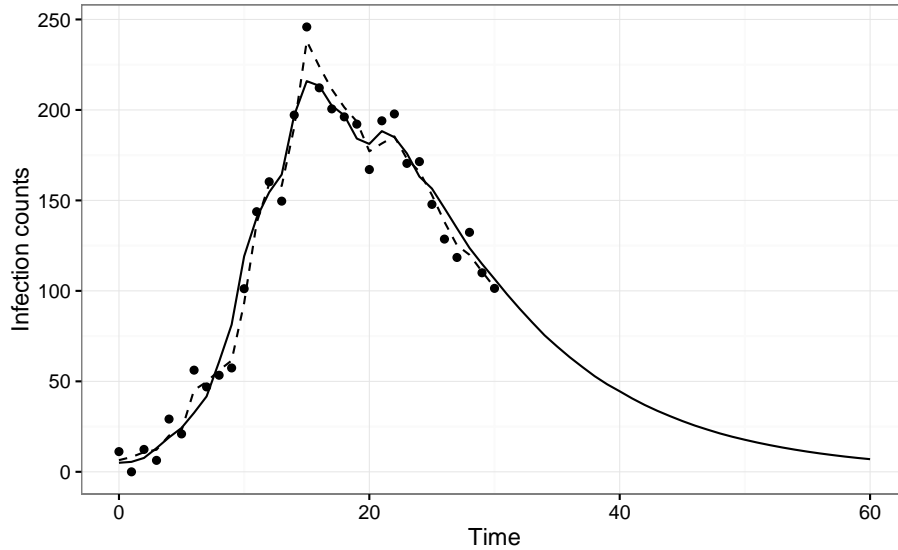


Figure 5.2: Infection count data truncated at  $T = 30$  from Figure [5.1]. The dashed line shows IF2's attempt to reconstruct the true underlying state from the observed data points.

### 1 5.2.1 Parametric Bootstrapping

2 The goal of the parametric bootstrap is use an initial density sample  $\theta^*$  to generate  
 3 further samples  $\theta_1, \theta_2, \dots, \theta_M$  from the sampling distribution of  $\theta$ . It works by using  $\theta$  to  
 4 generate artificial data sets  $D_1, D_2, \dots, D_M$  to which we can refit our model of interest  
 5 and generate new parameter sets. The literature suggests the most straightforward  
 6 way of doing this is to fit the model to obtain  $\theta^*$ , then use the model's forward  
 7 simulator to generate new data sets, in essence treating our original estimate  $\theta^*$  as  
 8 the “truth” set [[14]].

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

### 11 5.2.2 IF2 Forecasts

12 Using the parameter sets  $\theta_1, \theta_2, \dots, \theta_M$  and the point estimate of the state provided by  
 13 the initial IF2 fit, we can use forward simulations from the last estimated state  
 14 to produce estimates of the future state.

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

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

**Algorithm 5:** Parametric Bootstrap

---

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

```

/* Initial fit */
1  $\theta^* \leftarrow IF2(D)$ 
/* Generate artificial data sets */
2 for  $i = 1 : M$  do
3    $D_i \leftarrow S(\theta^*)$ 
/* Fit to new data sets */
4 for  $i = 1 : M$  do
5    $\theta_i \leftarrow IF2(D_i)$ 

```

---

**Output:** Distribution samples  $\theta_1, \theta_2, \dots, \theta_M$

---

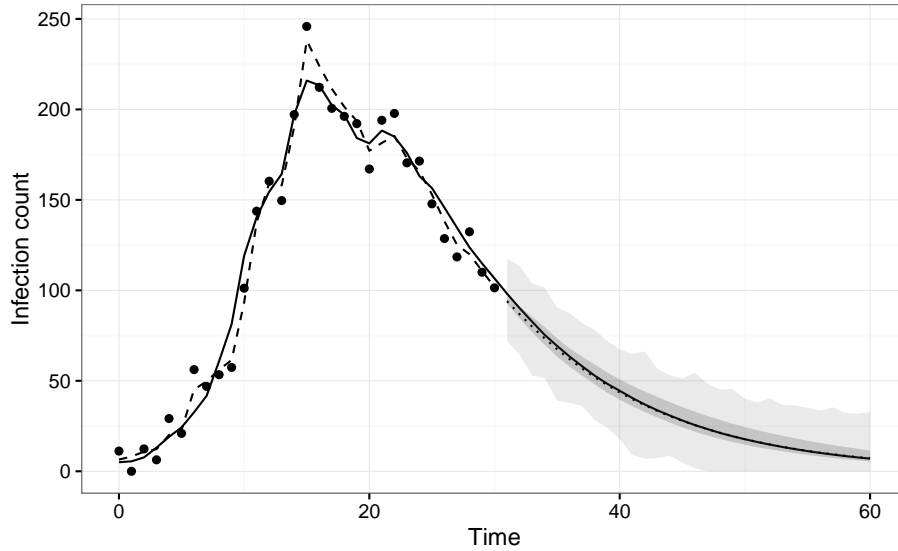


Figure 5.3: Forecast produced by the IF2 / parametric bootstrapping framework. The dotted line shows the mean estimate of the forecasts, the dark grey ribbon shows the 95% confidence interval based on the 0.025 and 0.975 quantiles on the true state estimates, and the lighter grey ribbon shows the same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

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

## 5 **5.3 HMC**

6 For HMC we can use a simpler approach to approach forecasting. We do not get state  
7 estimates directly from the RStan fitting due to the way we implemented the model,  
8 but we can construct them using the process noise latent variables as described in  
9 Chapter 2. Once we've done this we can forward simulate the system from the state  
10 estimate into the future.

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

13 And as before we can evaluate the SSE of the forecast for the data shown, giving  
14 approximately  $SSE = 608$ .

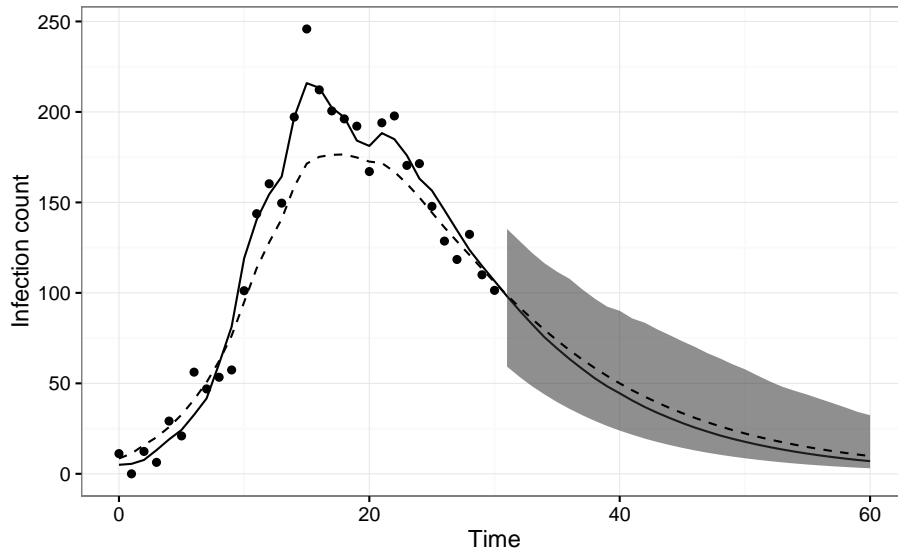


Figure 5.4: Forecast produced by the HMC / bootstrapping framework with  $M = 200$  trajectories. The dotted line shows the mean estimate of the forecasts, and the grey ribbon shows the 95% confidence interval on the estimated true states as described in Figure [5.3].

## 1 5.4 Truncation vs. Error

2 Of course the above mini-comparison only shows one truncation value for one trajec-  
 3 tory. Really, we need to know how each method performs on average given different  
 4 trajectories and truncation amounts. In effect we wish to “starve” each method of data  
 5 and see how poor the estimates become with each successive data point loss.

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

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

12 Since the parametric bootstrapping approach used by IF2 requires a significant num-  
 13 ber of additional fits, its computational cost is significantly higher than the simpler  
 14 bootstrapping approach used by the HMC framework, about 35.5x as expensive. How-  
 15 ever the now much longer running time can somewhat alleviated by parallelizing the  
 16 parametric bootstrapping process; as each of the parametric bootstrap fittings in en-  
 17 tirely independent, this can be done without a great deal of additional effort. The code

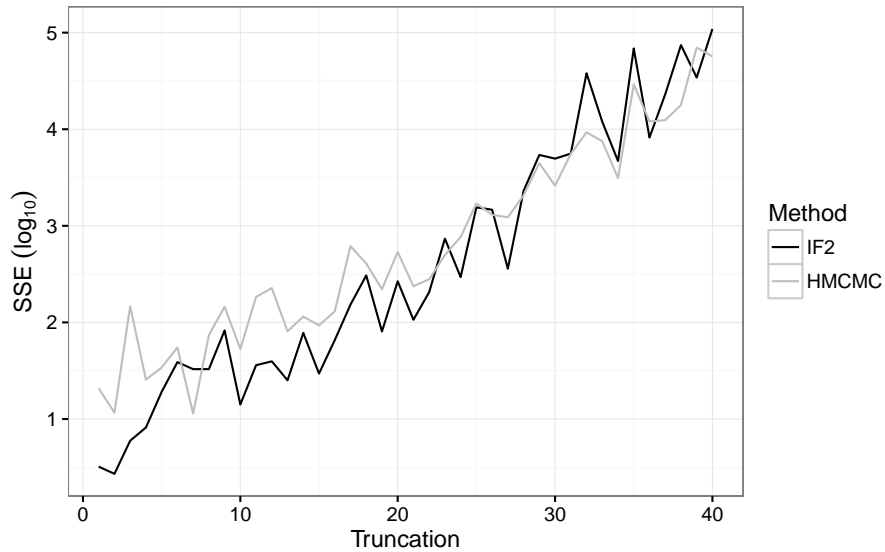


Figure 5.5: Error growth as a function of data truncation amount. Both methods used 200 bootstrap trajectories. Note that the y-axis shows the natural log of the averaged SSE, not the total SSE.

- 1 used here has this capability, but it was not utilised in the comparison so as to accu-
- 2 rately represent total computational cost, rather than potential running time.



# Chapter 6

## S-map and SIRS

### 6.1 S-maps

A family of forecasting methods that shy away from the mechanistic model-based approaches outlined in the previous sections have been developed by George Sugihara and collaborators [37][38][20][15] over the last several decades. As these methods do not include a mechanistic model in their forecasting process, they also do not attempt to perform parameter estimation or inference. Instead they attempt to reconstruct the underlying dynamical process as a weighted linear model from a time series.

One such method, the sequential locally weighted global linear maps (S-map), builds a global linear map model and uses it to produce forecasts directly. Despite relying on a linear mapping, the S-map does not assume the time series on which it is operating is the product of linear system dynamics, and in fact was developed to accommodate non-linear dynamics. The linear component of the method only comes into play when combining forecast components together to produce a single estimate

The S-map works by first constructing a time series embedding of length  $E$ , known as the library and denoted  $\{\mathbf{x}_i\}$ . Consider a time series of length  $T$  denoted  $x_1, x_2, \dots, x_T$ . Each element in the time series with indices in the range  $E, E + 1, \dots, T$  will have a corresponding entry in the library such that a given element  $x_t$  will correspond to a library vector of the form  $\mathbf{x}_i = (x_t, x_{t-1}, \dots, x_{t-E+1})$ . 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:

1 First construct a matrix  $A$  and vector  $b$  defined as

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

3 where  $\|\cdot\|$  is the Euclidean norm,  $i$  ranges over 1 to the length of the library, and  $j$   
4 ranges over  $[0, E]$ . In the above equations and the ones that follow, we set  $x_t(0) \equiv 1$   
5 to account for the linear term in the map.

6 The weighting function  $w$  is defined as

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

8 where  $d$  is the euclidean distance between the predictor vector and library vectors in  
9 Equation [6.1] and  $\bar{d}$  is the average of these distances. We can then see that  $\theta$  serves  
10 as a way to specify the appropriate level of penalization applied to poorly-matching  
11 library vectors – if  $\theta$  is 0 all weights are the same (no penalization), and increasing  $\theta$   
12 increases the level of penalization.

13 Now we solve the system  $Ac = b$  to obtain the linear weightings used to generate the  
14 forecast according to

$$\hat{y}_t = \sum_{j=0}^E c_t(j)\mathbf{x}_t(j). \tag{6.3}$$

16 In this way we have produced a forecast value for a single time. This process can  
17 be repeated for a sequence of times  $T + 1, T + 2, \dots$  to project a time series into the  
18 future.

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

## 24 **6.2 S-map Algorithm**

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

---

**Algorithm 6:** S-map

---

```

/* Select a starting point */
Input : Time series  $x_1, x_2, \dots, x_T$ , embedding dimension  $E$ , distance
        penalization  $\theta$ , forecast length  $L$ , predictor vector  $\mathbf{x}_t$ 

/* Construct library  $\{\mathbf{x}_i\}$  */
1 for  $i = E : T$  do
2    $\mathbf{x}_i = (x_i, x_{i-1}, \dots, x_{i-E+1})$ 

/* Construct mapping from library vectors to predictions */
3 for  $i = 1 : (T_E + 1)$  do
4   for  $j = 1 : E$  do
5      $A(i, j) = w(\|\mathbf{x}_i - \mathbf{x}_t\|)\mathbf{x}_i(j)$ 
6 for  $i = 1 : (T_E + 1)$  do
7    $b(i) = w(\|\mathbf{x}_i - \mathbf{x}_t\|)y_i$ 

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

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

/* Forecasted value in time series */
Output: Forecast  $\hat{y}_t$ 

```

---

## 6.3 SIRS Model

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

Given, the S-map's data requirements, we need to specify a modified version of the SIR model. As IF2 and HMC in principle should be able operate on any reasonably well-specified model, the easiest way to compare the efficacy of S-mapping to IF2 or HMC is to generate data from a SIRS model with a seasonal component, and have all methods operate on the resulting time series.

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

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

There are two new features here. We have a rate of waning immunity  $\eta$  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,  $\beta$  is allowed to walk restricted by a geometric mean, described by

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

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

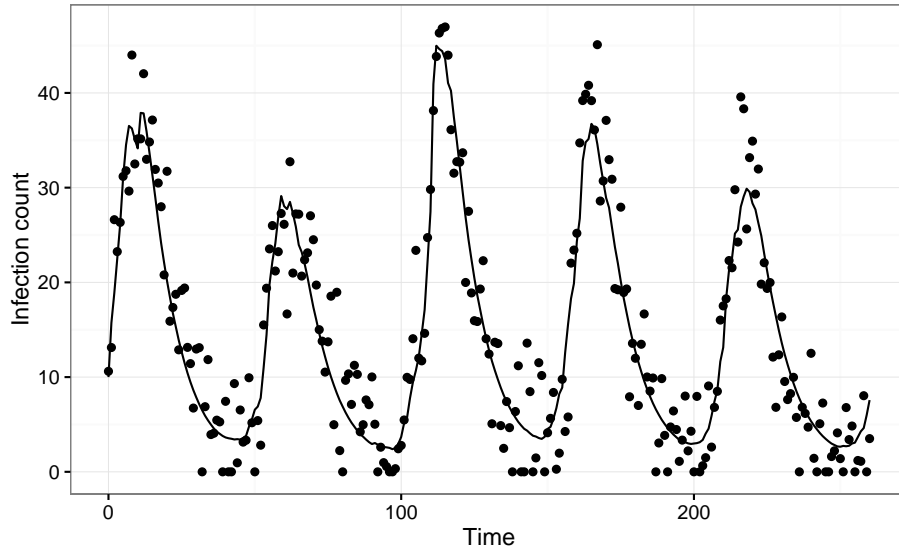


Figure 6.1: Five cycles generated by the SIRS function. The solid line the the true number of cases, dots show case counts with added observation noise. The parameter values were  $R_0 = 3.0$ ,  $\gamma = 0.1$ ,  $\eta = 1$ ,  $\sigma = 5$ , and 10 initial cases.

- 1 Figure [6.2] shows how the S-map can reconstruct the next cycle in the time se-
- 2 ries.

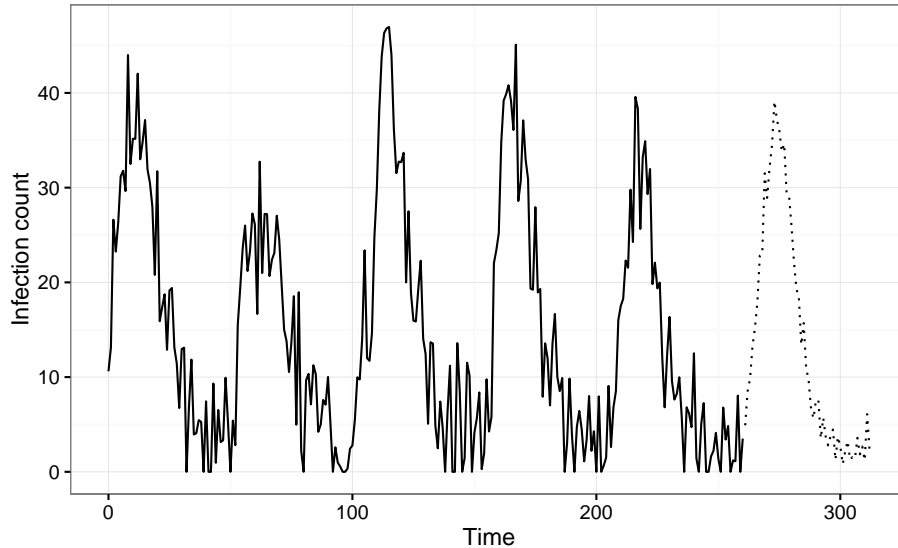


Figure 6.2: S-map applied to the data from the previous figure. The solid line shows the infection counts with observation noise from the previous plot, and the dotted line is the S-map forecast. Parameters chosen were  $E = 14$  and  $\theta = 3$ .

- 1 The parameters used in the S-map algorithm to obtain the forecast used in Figure
- 2 [6.2] were obtained using a grid search of potential parameters outlined in [15]. The
- 3 script to perform this optimisation procedure is included in the appendices.

## 4 6.4 SIRS Model Forecasting

5 Naturally we wish to compare the efficacy of this comparatively simple technique  
 6 against the more complex and more computationally taxing frameworks we have es-  
 7 tablished to perform forecasting using IF2 and HMC.

8 To do this we generated a series of artificial time series of length 260 meant to represent  
 9 5 years of weekly incidence counts and used each method to forecast up to 2 years into  
 10 the future. Our goal here was to determine how forecast error changed with forecast  
 11 length.

12 Figure [6.3] shows the results of the simulation.

13 Interestingly, all methods produce roughly the same result, which is to say the spikes  
 14 in each outbreak cycle are difficult to accurately predict. IF2 produces better results  
 15 than either HMC and the S-map for the majority of forecast lengths, with the S-map

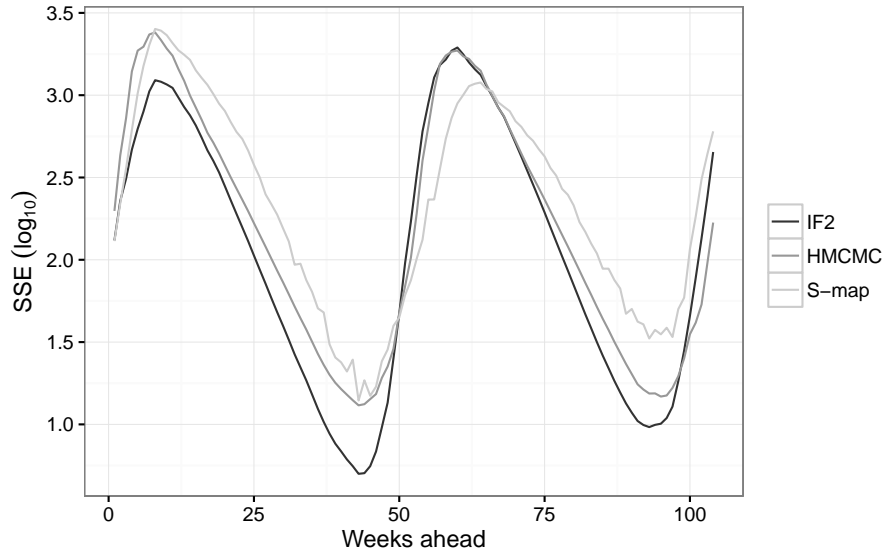


Figure 6.3: Error as a function of forecast length.

- 1 producing the poorest results with the exception of the second rise in infection rates
- 2 where it outperforms the other methods.
- 3 While the S-map may not provide the same fidelity or forecast as IF2 or HMC, it
- 4 shines when it comes to running time. Figure [6.4] shows the running times over 20
- 5 simulations.
- 6 It is clear from Figure [6.4] that the S-map running times are minute compared to the
- 7 other methods, but to emphasize the degree: The average running time for the S-map
- 8 is about 0.15 seconds, for IF2 it is about 47,000, and for HMC it is about 9,200.
- 9 This is a speed-up of over 316,000x compared to IF2 and over 61,800x compared to
- 10 HMC.

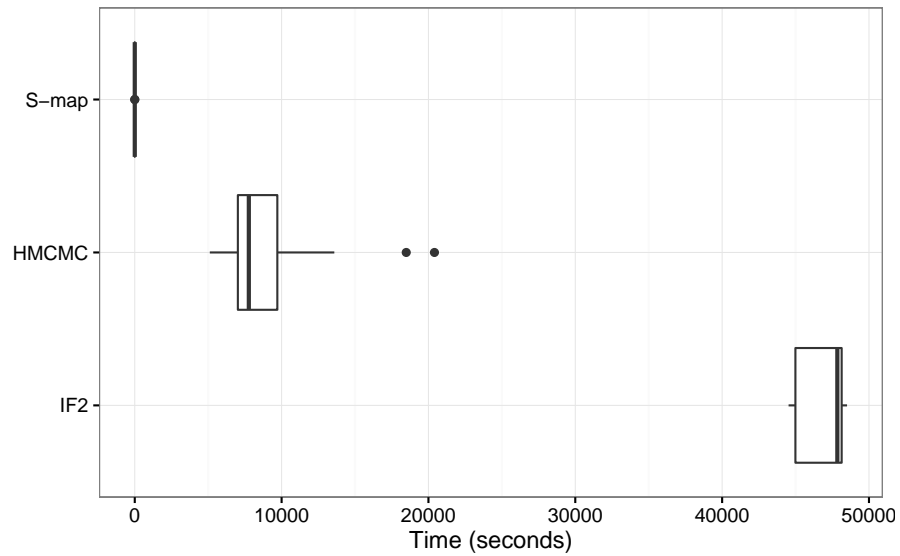


Figure 6.4: Runtimes for producing SIRS forecasts. The box shows the middle 50th percent, the bold line is the median, and the dots are outliers. Note that these are not “true” outliers, simply ones outside a threshold based on the interquartile range.



# Chapter 7

## Spatial Epidemics

### 7.1 Spatial SIR

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

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

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

Neighbours for a particular location are numbered  $j = 1, \dots, N_i$ . 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  $\beta$  embark on a geometric random walk defined as

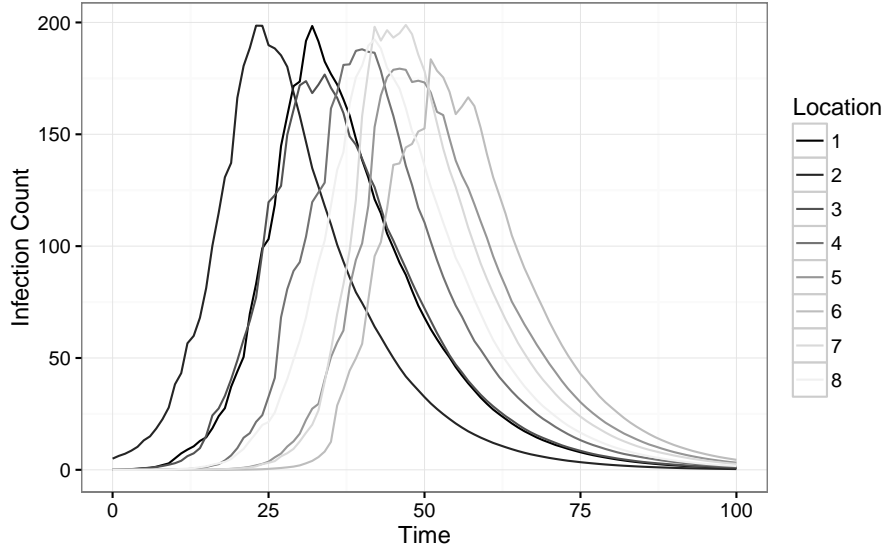


Figure 7.1: Evolution of a spatial epidemic in a ring topology. The outbreak was started with 5 cases in Location 2. Parameters were  $R_0 = 3.0$ ,  $\gamma = 0.1$ ,  $\eta = 0.5$ ,  $\sigma_{err} = 0.5$ , and  $\phi = 0.5$ .

- 1 
$$\beta_{i,t+1} = \exp \left( \log(\beta_{i,t}) + \eta(\log(\bar{\beta}) - \log(\beta_{i,t})) + \epsilon_t \right). \quad (7.2)$$
- 2 Note that as  $\beta$  is a state variable, each location has its own stochastic process driving
- 3 the evolution of its  $\beta$  state.
- 4 If we imagine a circular topology in which each of 8 locations is connected to exactly
- 5 two neighbours (i.e. location 1 is connected to locations 8 and 2, location 2 is
- 6 connected to locations 1 and 3, etc.), and we start each location with completely
- 7 susceptible populations except for a handful of infected individuals in one of the
- 8 locations, we obtain a plot of the outbreak progression in Figure [7.1].
- 9 If we add noise to the data from Figure [7.1], we obtain Figure [7.2].

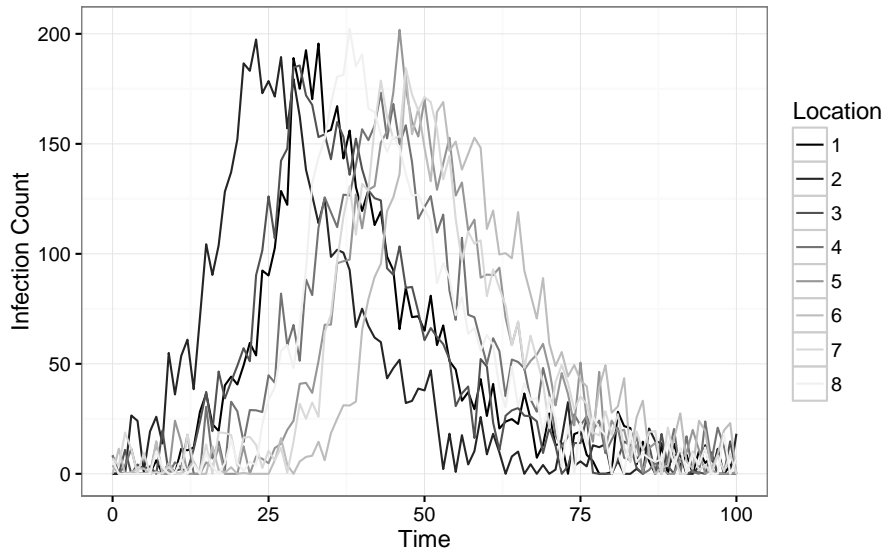


Figure 7.2: Evolution of a spatial epidemic as in Figure [7.1], with added observation noise drawn from  $\mathcal{N}(0, 10)$ .

## 7.2 Dewdrop Regression

Dewdrop regression [20] 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 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.

1 This means the rescaling process in Dewdrop Regression is not necessary and can  
2 be skipped. Further, the overall variation between the epidemic curves in each loca-  
3 tion is on the smaller side, meaning the S-map will have a high-quality library from  
4 which to build forecasts.

## 5 **7.3 Spatial Model Forecasting**

6 In order to compare the forecasting efficacy of Dewdrop Regression with S-mapping  
7 against IF2 and HMC, we generated 20 independent spatial data sets up to time  
8  $T = 50$  weeks in each of  $L = 10$  locations and forecasted 10 weeks into the future.  
9 Forecasts were compared to that of the true model evolution, and the average  $SSE$   
10 for each week ahead in the forecast were computed. The number of bootstrapping  
11 trajectories used by IF2 and HMC was reduced from 200 to 50 to curtail running  
12 times.

13 The results are shown in Figure [7.3].

14 The results show a clear delineation in forecast fidelity between methods. IF2 main-  
15 tains an advantage regardless of how long the forecast produced. Interestingly, Dew-  
16 drop Regression with S-mapping performs almost as well as IF2, and outperforms  
17 HMC. HMC lags behind both methods by a healthy margin.

18 If we examine the runtimes for each forecast framework, we obtain the data in Figure  
19 [7.4].

20 As before, the S-map with Dewdrop Regression runs faster than the other two methods  
21 with a huge margin. It is again hard to see exactly how large the margin is from the  
22 figure due to the scale, but we can examine the average values: the average running  
23 time for S-mapping with Dewdrop Regression was about 249 seconds, whereas the  
24 average times for IF2 and HMC were about 29,000 seconds and 38,800 seconds,  
25 respectively. This is a speed-up of just over 116x over IF2 and 156x over HMC.

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

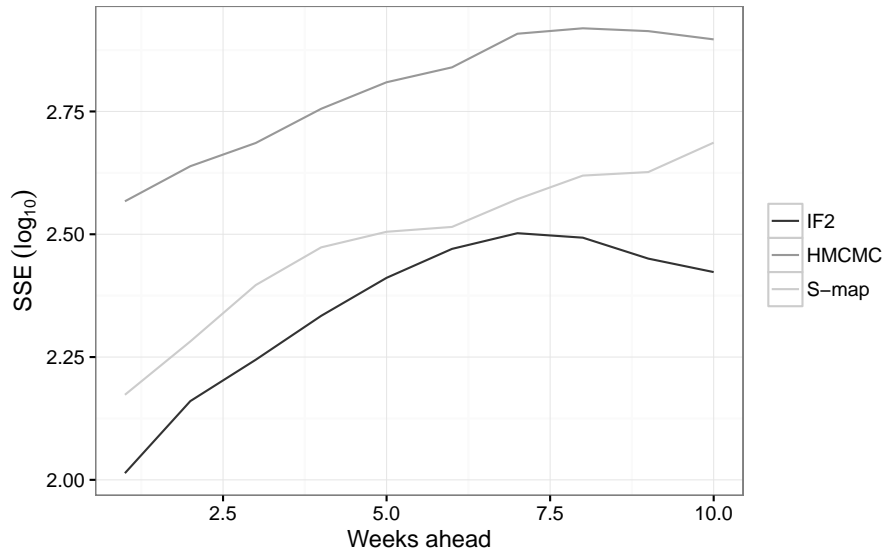


Figure 7.3: Average SSE (log scale) across each location and all trials as a function of the number of weeks ahead in the forecast.

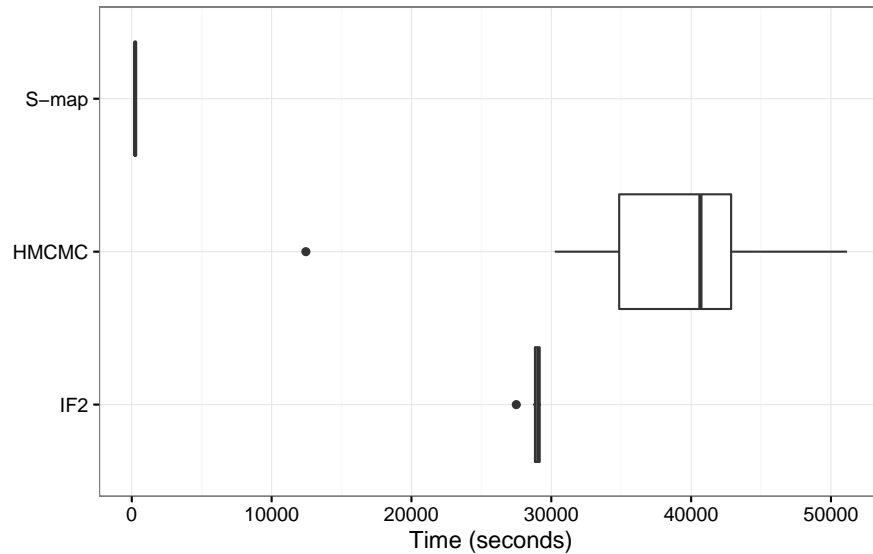


Figure 7.4: Runtimes for producing spatial SIR forecasts. The box shows the middle 50th percent, the bold line is the median, and the dots are outliers.

# Chapter 8

## Discussion and Future Directions

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

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

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

### 8.1 Parallel and Distributed Computing

Whenever running times are discussed, we must consider the current computing landscape and hardware boundaries. In 1965, Intel co-founder Gordon E. Moore published a paper in which he observed that the number of transistors per unit area in integrated

1 circuits double roughly every year. The consequence of this growth is the approximate  
2 year-over-year doubling of clock speeds (maximum number of sequential calculations  
3 performed per second), equivalent to raw performance of the chip. This forecast was  
4 updated in 1975 to double every 2 years and has held steady until the very recent  
5 past [40].

6 Recently, transistor count growth has begun to falter. This is due to several physical  
7 factors preventing tighter “packing” of transistors into a single processor. To com-  
8 pensate for these limitations, chip manufacturers have instead redesigned the internal  
9 chip structures to consist of smaller “cores” within a single CPU die. The resulting  
10 processing power per processor then stays on track with Moore’s Law, but keeps the  
11 clock speeds of each individual core under control.

12 Of course this raises many problems on the software and algorithm side of computing.  
13 Using several smaller cores instead of a single large one has the distinct disadvantage  
14 of lack of cohesion – the cores must execute instructions completely decoupled from  
15 each other. This means algorithms have to be redesigned, or at least rewritten at the  
16 software level to consist of multiple independent pieces that can be run in parallel.  
17 This practice is known as parallelization, and has become critical in taking full ad-  
18 vantage of machines of all scales – from mobile phones which overwhelmingly favour  
19 multi-core CPU architectures, to large clusters and supercomputers which rely on  
20 distributed computing “nodes”.

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

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

1 With regards to the bootstrapping process we used with HMC, it should be clear that  
 2 each bootstrap trajectory is completely independent, and thus this component of the  
 3 forecasting framework can be considered “embarrassingly” parallel or distributed. Un-  
 4 fortunately, however, this is the least computationally demanding part of the process  
 5 by several orders of magnitude, and so working to parallelize it would provide little  
 6 advantage.

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

- 9     • Forward evolution of the particles’ internal system state using their parameter  
 10       state
- 11     • Weighting those state estimates against the data point using the observation  
 12       function
- 13     • Particle weight normalizations
- 14     • Resampling from the particle weight distribution
- 15     • Particle parameter perturbations

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

24 Further, the full parametric bootstrapping process is highly computationally demand-  
 25 ing, and also completely parallelizable. Each trajectory requires a fair bit of time to  
 26 generate, on the order of of the original fitting time, and can be computed completely  
 27 independently. Hence, IF2 is a very good candidate for a good parallel implementa-  
 28 tion.

29 A future offshoot of this project would be a good parallel implementation of both the  
 30 IF2 fitting process and the parametric bootstrapping framework. An ideal platform for  
 31 this work would be NVIDIA’s Compute Unified Device Architecture (CUDA) Graph-  
 32 ics Processing Unit (GPU) computing framework. While a CUDA implementation of  
 33 a spatial epidemic IF2 parameter fitting algorithm was implemented, it lacked a good  
 34 front-end implementation, R integration, and a parametric bootstrapping framework  
 35 and so was not included in the main results of this paper. However, the code and  
 36 some preliminary results are included in the appendices.

37 S-mapping, like the other two methods, is parallelizable to a degree. However, the



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

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

## 15 8.2 IF2, Bootstrapping, and Forecasting Method- 16 ology

17 The parametric bootstrapping approach used to generate additional parameter pos-  
18 terior samples and produce forecasts has proven effective, but not necessarily compu-  
19 tationally efficient.

20 A recent paper utilising IF2 for forecasting [24] generated trajectories using IF2,  
21 parameter likelihood profiles, weighted quantiles, and the basic particle filter. The  
22 parameter profiles were used to construct a bounding box to search for good parameter  
23 sets, within which combinations of parameters to generate forecasts were selected  
24 using a Sobol sequence. Finally the forecasts were combined using a weighted quantile,  
25 taking into account the likelihood of the parameter sets used. Whether this approach  
26 would result in higher quality forecasts or lower running times is of interest, and could  
27 serve as a future research direction.

28 Expanding on this, there are other bootstrapping approaches that could be used to  
29 produce forecasts. A paper focusing solely on using IF2 with varied bootstrapping  
30 approaches and determining a forecast accuracy versus computational time trade-off  
31 curve of sorts would be useful, and would be another step towards establishing which  
32 tools are best for which jobs.

## 1 **8.3 Fin**

2 The overarching theme in this paper, from the theoretical considerations to the results  
3 to the discussion, is that there still exists no “silver bullet” for forecasting problems.  
4 Largely you can decide, as the user, how accurate you need your results to be, how  
5 much computer time you have at your disposal, and how fast you need your results,  
6 and select the method that best satisfies your needs. If speed is the priority, then you  
7 can use S-mapping to get very quick and relatively accurate results. If you require  
8 accuracy above all else, you must turn to heavier methods such as IF2, HMC, and  
9 parametric bootstrapping in order to produce the cleanest forecast possible. And  
10 this represents only three data points in a larger picture. There are a wide variety  
11 of methods that are similar but not identical to methods explored here, each with  
12 their own positive and negative attributes, their own advantages and disadvantages,  
13 and that are ultimately likely to fill out our spectrum of methods more completely.  
14 Thus future work should focus on attempting further direct comparison across a wider  
15 swath of techniques, and implementing those techniques in a parallel fashion to take  
16 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%26%7Dttool=pmcentrez%7B%5C%26%7Drendertype=abstract>.
- [5] M Babyak. “What You See May Not Be What You Get: A Brief, Nontechnical Introduction to Overfitting in Regression Type Models”. In: *J. Bio. Med.* 66.3 (2004), pp. 411–421. ISSN: 0033-3174. DOI: 10.1097/01.psy.0000127692.23278.a9.
- [6] Thomas Bengtsson, Peter Bickel, and Bo Li. “Curse-of-dimensionality revisited: Collapse of the particle filter in very large scale systems”. In: *Probability and Statistics* 2 (2008), pp. 316–334. DOI: 10.1214/193940307000000518. arXiv: 0805.3034. URL: <http://arxiv.org/abs/0805.3034>.
- [7] a. Camacho et al. “Explaining rapid reinfections in multiple-wave influenza outbreaks: Tristan da Cunha 1971 epidemic as a case study”. In: *Proceedings of the Royal Society B: Biological Sciences* 278 (2011), pp. 3635–3643. ISSN: 0962-8452. DOI: 10.1098/rspb.2011.0300.

- 1 [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>.
- 2
- 3
- 4 [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>.
- 5
- 6
- 7
- 8
- 9 [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.
- 10
- 11
- 12 [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>.
- 13
- 14
- 15
- 16 [12] Jonathan Dushoff et al. “Dynamical resonance can account for seasonality of influenza epidemics.” In: *Proceedings of the National Academy of Sciences of the United States of America* 101.48 (2004), pp. 16915–16916. ISSN: 0027-8424. DOI: 10.1073/pnas.0407293101. URL: <http://www.pnas.org/content/101/48/16915.full>.
- 17
- 18
- 19
- 20
- 21 [13] Dirk Eddelbuettel and Romain Fran. “Rcpp : Seamless R and C ++ Integration”. In: *Journal Of Statistical Software* 40.8 (2011), pp. 1–18. ISSN: 15487660. DOI: 10.1007/978-1-4614-6868-4. arXiv: arXiv:1011.1669v3. URL: <http://www.jstatsoft.org/v40/i08/>.
- 22
- 23
- 24
- 25 [14] Christian Genest and Bruno Remillard. “Validity of the parametric bootstrap for goodness-of-fit testing in semiparametric models”. In: *Annales de l’institut Henri Poincare (B) Probability and Statistics* 44.6 (2008), pp. 1096–1127. ISSN: 02460203. DOI: 10.1214/07-AIHP148.
- 26
- 27
- 28
- 29 [15] Sarah M. Glaser, Hao Ye, and George Sugihara. “A nonlinear, low data requirement model for producing spatially explicit fishery forecasts”. In: *Fisheries Oceanography* 23 (2014), pp. 45–53. ISSN: 10546006. DOI: 10.1111/fog.12042.
- 30
- 31
- 32 [16] Andrea L Graham et al. “Explaining rapid reinfections in multiple- wave influenza outbreaks : Tristan da Cunha 1971 epidemic as a case study”. In: *Proc. R. Soc. B* (2016). DOI: 10.1098/rspb.2011.0300.
- 33
- 34
- 35 [17] Bradley Harding. “Standard errors : A review and evaluation of standard error estimators using Monte Carlo simulations”. In: (2014), pp. 107–123.
- 36
- 37 [18] 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.
- 38
- 39

- 1 [19] Matthew D. Hoffman and Andrew Gelman. “The No-U-Turn Sampler: Adap-  
2 tively Setting Path Lengths in Hamiltonian Monte Carlo”. In: *Journal of Ma-*  
3 *chine Learning Research* 15.April (2014), pp. 1593–1623. arXiv: 1111.4246.  
4 URL: <http://mc-stan.org/>.
- 5 [20] Chih-hao Hsieh, Christian Anderson, and George Sugihara. “Extending nonlin-  
6 ear analysis to short ecological time series.” In: *The American naturalist* 171.1  
7 (2008), pp. 71–80. ISSN: 0003-0147. DOI: 10.1086/524202.
- 8 [21] E L Ionides, C Bret, and a a King. “Inference for nonlinear dynamical systems.”  
9 In: *Proceedings of the National Academy of Sciences of the United States of*  
10 *America* 103.49 (Dec. 2006), pp. 18438–43. ISSN: 0027-8424. DOI: 10.1073/pnas.  
11 0603181103. URL: [http://www.pubmedcentral.nih.gov/articlerender.fcgi?](http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=30201387B%5C&%7Dtool=pmcentrez%7B%5C&%7Drendertype=abstract)  
12 [artid=30201387B%5C&%7Dtool=pmcentrez%7B%5C&%7Drendertype=abstract](http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=30201387B%5C&%7Dtool=pmcentrez%7B%5C&%7Drendertype=abstract).
- 13 [22] Edward L. Ionides et al. “Inference for dynamic and latent variable models  
14 via iterated, perturbed Bayes maps”. In: *Proceedings of the National Academy*  
15 *of Sciences* 112.3 (2015), pp. 719–724. ISSN: 0027-8424. DOI: 10.1073/pnas.  
16 1410597112. URL: [http://www.pnas.org/lookup/doi/10.1073/pnas.](http://www.pnas.org/lookup/doi/10.1073/pnas.1410597112)  
17 [1410597112](http://www.pnas.org/lookup/doi/10.1073/pnas.1410597112).
- 18 [23] Aaron A King, Dao Nguyen, and Edward L. Ionides. “Statistical Inference for  
19 Partially Observed Markov Processes via the R Package pomp”. In: *Journal of*  
20 *Statistical Software* 59.10 (2015). arXiv: arXiv:1509.00503v1.
- 21 [24] Aaron A King et al. “Avoidable errors in the modelling of outbreaks of emerg-  
22 ing pathogens, with special reference to Ebola”. In: *Proceedings of the Royal*  
23 *Society B: Biological Sciences* 282.1806 (2015), pp. 20150347–20150347. ISSN:  
24 0962-8452. DOI: 10.1098/rspb.2015.0347. arXiv: 1412.0968. URL: [http:](http://rspb.royalsocietypublishing.org/cgi/doi/10.1098/rspb.2015.0347)  
25 [//rspb.royalsocietypublishing.org/cgi/doi/10.1098/rspb.2015.0347](http://rspb.royalsocietypublishing.org/cgi/doi/10.1098/rspb.2015.0347).
- 26 [25] Aaron A King et al. *pomp: Statistical inference for partially observed Markov*  
27 *processes*. 2016. URL: <http://kingaa.github.io/pomp>.
- 28 [26] William A. Link and Mitchell J. Eaton. “On thinning of chains in MCMC”.  
29 In: *Methods in Ecology and Evolution* 3.1 (2012), pp. 112–115. ISSN: 2041210X.  
30 DOI: 10.1111/j.2041-210X.2011.00131.x.
- 31 [27] Kanti V. Mardia et al. “The Kriged Kalman filter”. In: *Test* 7.2 (1998), pp. 217–  
32 282. ISSN: 11330686. DOI: 10.1007/BF02565111.
- 33 [28] Radford M Neal. “Handbook of Markov Chain Monte Carlo”. In: *Handbook of*  
34 *Markov Chain Monte Carlo* 20116022 (2011), pp. 113–162. ISSN: jnullj. DOI:  
35 10.1201/b10905. arXiv: arXiv:1206.1901v1. URL: [http://www.crcnetbase.](http://www.crcnetbase.com/doi/book/10.1201/b10905)  
36 [com/doi/book/10.1201/b10905](http://www.crcnetbase.com/doi/book/10.1201/b10905).

- 1 [29] Elaine O Nsoesie et al. “A systematic review of studies on forecasting the dy-  
2 namics of influenza outbreaks.” In: *Influenza and other respiratory viruses* 8.3  
3 (May 2014), pp. 309–16. ISSN: 1750-2659. DOI: 10.1111/irv.12226. URL: [http:  
4 //www.pubmedcentral.nih.gov/articlerender.fcgi?artid=4181479%7B%5C%  
5 %7Dtool=pmcentrez%7B%5C%7Drendertype=abstract](http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=4181479%7B%5C%7Dtool=pmcentrez%7B%5C%7Drendertype=abstract).
- 6 [30] Elaine Nsoesie, Madhav Marathe, and John Brownstein. “Forecasting peaks of  
7 seasonal influenza epidemics.” In: *PLoS currents* 5 (Jan. 2013), pp. 1–14. ISSN:  
8 2157-3999. DOI: 10.1371/currents.outbreaks.bb1e879a23137022ea79a8c508b030bc.  
9 URL: [http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=  
10 3712489%7B%5C%7Dtool=pmcentrez%7B%5C%7Drendertype=abstract](http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3712489%7B%5C%7Dtool=pmcentrez%7B%5C%7Drendertype=abstract).
- 11 [31] Robert C Reiner et al. “Highly localized sensitivity to climate forcing drives  
12 endemic cholera in a megacity.” In: *Proceedings of the National Academy of  
13 Sciences of the United States of America* 109.6 (Feb. 2012), pp. 2033–6. ISSN:  
14 1091-6490. DOI: 10.1073/pnas.1108438109. URL: [http://www.pubmedcentral.  
15 nih.gov/articlerender.fcgi?artid=3277579%7B%5C%7Dtool=pmcentrez%  
16 7B%5C%7Drendertype=abstract](http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3277579%7B%5C%7Dtool=pmcentrez%7B%5C%7Drendertype=abstract).
- 17 [32] Sujit K Sahu. “A Bayesian Kriged-Kalman model for short-term fore- casting of  
18 air pollution levels”. In: *Appl. Statist* 54 (2005), pp. 223–244. ISSN: 0035-9254.  
19 DOI: 10.1111/j.1467-9876.2005.00480.x.
- 20 [33] Jacques Sau et al. “Particle filter-based real-time estimation and prediction of  
21 traffic conditions Modeling framework”. In: *Information Systems* (1918), pp. 1–  
22 8.
- 23 [34] Jeffrey Shaman and Alicia Karspeck. “Forecasting seasonal outbreaks of in-  
24 fluenza.” In: *Proceedings of the National Academy of Sciences of the United  
25 States of America* 109.50 (Dec. 2012), pp. 20425–30. ISSN: 1091-6490. DOI:  
26 10.1073/pnas.1208772109. URL: [http://www.pubmedcentral.nih.gov/  
27 articlerender.fcgi?artid=3528592%7B%5C%7Dtool=pmcentrez%7B%5C%  
28 %7Drendertype=abstract](http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3528592%7B%5C%7Dtool=pmcentrez%7B%5C%7Drendertype=abstract).
- 29 [35] Jeffrey Shaman, Wan Yang, and Sasikiran Kandula. “Inference and Forecast of  
30 the Current West African Ebola Outbreak in Guinea, Sierra Leone and Liberia”.  
31 In: *PLoS Currents* 6 (2014), ecurrents.outbreaks.3408774290b1a0f2dd7cae877c8b8f.  
32 ISSN: 2157-3999. DOI: 10.1371/currents.outbreaks.3408774290b1a0f2dd7cae877c8b8ff6.
- 33 [36] Stan Development Team. “Stan Modeling Language User’s Guide and Reference  
34 Manual”. In: 2.9.0 (2015).
- 35 [37] G Sugihara and R M May. *Nonlinear forecasting as a way of distinguishing  
36 chaos from measurement error in time series*. 1990. DOI: 10.1038/344734a0.
- 37 [38] George Sugihara. “Nonlinear Forecasting for the Classification of Natural Time  
38 Series”. In: *Philosophical Transactions of the Royal Society A: Mathematical,  
39 Physical and Engineering Sciences* 348 (1994), pp. 477–495. ISSN: 1364-503X.  
40 DOI: 10.1098/rsta.1994.0106.

- 1 [39] Carmen L Vidal Rodeiro and Andrew B Lawson. “Online updating of space-  
2 time disease surveillance models via particle filters.” In: *Statistical methods in*  
3 *medical research* 15.5 (2006), pp. 423–444. ISSN: 0962-2802. DOI: 10.1177 /  
4 0962280206071640.
- 5 [40] Mitchell Waldrop. “More then Moore”. In: *Nature* 530.11. February (2016),  
6 p. 145.
- 7 [41] Wan Yang, Alicia Karspeck, and Jeffrey Shaman. “Comparison of filtering meth-  
8 ods for the modeling and retrospective forecasting of influenza epidemics.” In:  
9 *PLoS computational biology* 10.4 (Apr. 2014), e1003583. ISSN: 1553-7358. DOI:  
10 10.1371/journal.pcbi.1003583. URL: [http://www.pubmedcentral.nih.gov/  
11 articlerender.fcgi?artid=3998879%7B%5C%7Dtool=pmcentrez%7B%5C%  
12 %7Drendertype=abstract](http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3998879%7B%5C%7Dtool=pmcentrez%7B%5C%7Drendertype=abstract).
- 13 [42] Xingyu Zhang et al. “Comparative study of four time series methods in fore-  
14 casting typhoid fever incidence in China.” In: *PloS one* 8.5 (Jan. 2013), e63116.  
15 ISSN: 1932-6203. DOI: 10.1371/journal.pone.0063116. URL: [http://www.  
16 pubmedcentral.nih.gov/articlerender.fcgi?artid=3641111%7B%5C%  
17 %7Dtool=pmcentrez%7B%5C%7Drendertype=abstract](http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3641111%7B%5C%7Dtool=pmcentrez%7B%5C%7Drendertype=abstract).

# 1 Appendix A

## 2 Hamiltonian MCMC

### 3 A.1 Full R code

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

```
5  
6 1 ## Dexter Barrows  
7 2 ## dbarrows.github.io  
8 3 ## McMaster University  
9 4 ## 2016  
10 5  
11 6 library(deSolve)  
12 7 library(rstan)  
13 8 library(shinystan)  
14 9 library(ggplot2)  
15 10 library(RColorBrewer)  
16 11 library(reshape2)  
17 12  
18 13 SIR ← function(Time, State, Pars) {  
19 14  
20 15     with(as.list(c(State, Pars)), {  
21 16  
22 17         B ← R0*r/N  
23 18         BSI ← B*S*I  
24 19         rI ← r*I  
25 20  
26 21         dS = -BSI  
27 22         dI = BSI - rI  
28 23         dR = rI  
29 24  
30 25         return(list(c(dS, dI, dR)))  
31 26  
32 27     })  
33 28  
34 29 }
```



```

1 30
2 31 pars <- c(R0 <- 3.0,      # average number of new infected individuals
3      per infectious person
4 32          r <- 0.1,      # recovery rate
5 33          N <- 500)      # population size
6 34
7 35 T <- 100
8 36 y_ini <- c(S = 495, I = 5, R = 0)
9 37 times <- seq(0, T, by = 1)
10 38
11 39 odeout <- ode(y_ini, times, SIR, pars)
12 40
13 41 set.seed(1001)
14 42 sigma <- 10
15 43 infec_counts_raw <- odeout[,3] + rnorm(T+1, 0, sigma)
16 44 infec_counts <- ifelse(infec_counts_raw < 0, 0, infec_counts_raw)
17 45
18 46 g <- qplot(0:T, odeout[,3], geom = "line", xlab = "Time (weeks)", ylab
19      = "Infection Count") +
20      geom_point(aes(y = infec_counts)) +
21      theme_bw()
22 49
23 50 print(g)
24 51 ggsave(g, filename="dataplot.pdf", height=4, width=6.5)
25 52
26 53 sPw <- 7
27 54 datlen <- (T-1)*7 + 1
28 55
29 56 data <- matrix(data = -1, nrow = T+1, ncol = sPw)
30 57 data[,1] <- infec_counts
31 58 standata <- as.vector(t(data))[1:datlen]
32 59
33 60 sir_data <- list( T = datlen,      # simulation time
34      y = standata, # infection count data
35      N = 500,      # population size
36      h = 1/sPw )   # step size per day
37 64
38 65 rstan_options(auto_write = TRUE)
39 66 options(mc.cores = parallel::detectCores())
40 67 stan_options <- list( chains = 4,      # number of chains
41      iter = 2000, # iterations per chain
42      warmup = 1000, # warmup iterations
43      thin = 2)     # thinning number
44 71 fit <- stan(file = "d_sirole_euler.stan",
45      data = sir_data,
46      chains = stan_options$chains,
47      iter = stan_options$iter,
48      warmup = stan_options$warmup,
49      thin = stan_options$thin )
50 77
51 78 exfit <- extract(fit, permuted = TRUE, inc_warmup = FALSE)

```

```

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

```

```

1 120         scale_color_brewer(palette="Greys") +
2 121         theme_bw()
3 122
4 123 print(tracefitR0)
5 124 ggsave(tracefitR0, filename="traceplotR0.pdf", height=4, width=6.5)
6 125
7 126 exfit <- extract(fit, permuted = FALSE, inc_warmup = TRUE)
8 127 plotdata <- melt(exfit[,,"R0"])
9 128 tracefitR0 <- ggplot() +
10 129     geom_line(data = plotdata,
11 130               aes(x = iterations,
12 131                   y = value,
13 132                   color = factor(chains, labels = 1:stan_
14 133                                 options$chains))) +
15 134     labs(x = "Sample", y = expression(R[0]), color = "Chain
16 135           ") +
17 136     scale_color_brewer(palette="Greys") +
18 137     theme_bw()
19 138
20 139 print(tracefitR0)
21 140 ggsave(tracefitR0, filename="traceplotR0_inc.pdf", height=4, width
22 141         =6.5)
23 142
24 143 sso <- as.shinystan(fit)
25 144 sso <- launch_shinystan(sso)

```

## 27 A.2 Full Stan code

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

```

29
30 1 ## Dexter Barrows
31 2 ## dbarrows.github.io
32 3 ## McMaster University
33 4 ## 2016
34 5
35 6 data {
36 7
37 8     int      <lower=1>    T;      // total integration steps
38 9     real      y[T];      // observed number of cases
39 10    int      <lower=1>    N;      // population size
40 11    real      h;          // step size
41 12
42 13 }
43 14
44 15 parameters {
45 16
46 17     real <lower=0, upper=10> R0;    // R0
47 18     real <lower=0, upper=10> r;    // recovery rate

```

```

1 19      real <lower=0, upper=20>    sigma; // observation error
2 20      real <lower=0, upper=500>  y0[3]; // initial conditions
3 21
4 22 }
5 23
6 24 model {
7 25
8 26     real S[T];
9 27     real I[T];
10 28     real R[T];
11 29
12 30     S[1] <- y0[1];
13 31     I[1] <- y0[2];
14 32     R[1] <- y0[3];
15 33
16 34     y[1] ~ normal(y0[2], sigma);
17 35
18 36     for (t in 2:T) {
19 37
20 38         S[t] <- S[t-1] + h*( - S[t-1]*I[t-1]*R0*r/N );
21 39         I[t] <- I[t-1] + h*( S[t-1]*I[t-1]*R0*r/N - I[t-1]*r );
22 40         R[t] <- R[t-1] + h*( I[t-1]*r );
23 41
24 42         if (y[t] > 0) {
25 43             y[t] ~ normal( I[t], sigma );
26 44         }
27 45
28 46     }
29 47
30 48     y0[1] ~ normal(N - y[1], sigma);
31 49     y0[2] ~ normal(y[1], sigma);
32 50
33 51     R0      ~ lognormal(1,1);
34 52     r        ~ lognormal(1,1);
35 53     sigma    ~ lognormal(1,1);
36 54
37 55 }
38

```

# 1 Appendix B

## 2 Iterated Filtering

### 3 B.1 Full R code

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

```
5  
6 1 ## Dexter Barrows  
7 2 ## dbarrows.github.io  
8 3 ## McMaster University  
9 4 ## 2016  
10 5  
11 6 library(deSolve)  
12 7 library(ggplot2)  
13 8 library(reshape2)  
14 9 library(gridExtra)  
15 10 library(Rcpp)  
16 11  
17 12 SIR ← function(Time, State, Pars) {  
18 13  
19 14     with(as.list(c(State, Pars)), {  
20 15  
21 16         B ← R0*r/N  
22 17         BSI ← B*S*I  
23 18         rI ← r*I  
24 19  
25 20         dS = -BSI  
26 21         dI = BSI - rI  
27 22         dR = rI  
28 23  
29 24         return(list(c(dS, dI, dR)))  
30 25  
31 26     })  
32 27  
33 28 }  
34 29
```

```

1 30 T      ← 100
2 31 N      ← 500
3 32 sigma  ← 10
4 33 i_infec ← 5
5 34
6 35 ## Generate true trajecory and synthetic data
7 36 ##
8 37
9 38 true_init_cond ← c(S = N - i_infec,
10 39                  I = i_infec,
11 40                  R = 0)
12 41
13 42 true_pars ← c(R0 = 3.0,
14 43              r = 0.1,
15 44              N = 500.0)
16 45
17 46 odeout ← ode(true_init_cond, 0:T, SIR, true_pars)
18 47 trueTraj ← odeout[,3]
19 48
20 49 set.seed(1001)
21 50
22 51 infec_counts_raw ← odeout[,3] + rnorm(T+1, 0, sigma)
23 52 infec_counts     ← ifelse(infec_counts_raw < 0, 0, infec_counts_raw)
24 53
25 54 g ← qplot(0:T, odeout[,3], geom = "line", xlab = "Time (weeks)", ylab
26 55         = "Infection Count") +
27 56     geom_point(aes(y = infec_counts)) +
28 57     theme_bw()
29 58
30 58 print(g)
31 59 ggsave(g, filename="dataplot.pdf", height=4, width=6.5)
32 60
33 61 ## Rcpp stuff
34 62 ##
35 63
36 64 sourceCpp(paste(getwd(),"d_if2.cpp",sep="/"))
37 65
38 66 paramdata ← data.frame(if2(infec_counts, T+1, N))
39 67 colnames(paramdata) ← c("R0", "r", "sigma", "Sinit", "Iinit", "Rinit"
40 68                        )
41 68
42 69 ## Parameter density kernels
43 70 ##
44 71
45 72 R0points ← paramdata$R0
46 73 R0kernel ← qplot(R0points, geom = "density", xlab = expression(R[0]),
47 74                ylab = "frequency") +
48 75                geom_vline(aes(xintercept=true_pars[["R0"]]), linetype="
49 76                dashed", size=1, color="grey50") +
50 77                theme_bw()
51 76

```

```

1 77 print(R0kernel)
2 78 ggsave(R0kernel, filename="kernelR0.pdf", height=3, width=3.25)
3 79
4 80 rpoints <- paramdata$r
5 81 rkernel <- qplot(rpoints, geom = "density", xlab = "r", ylab = "
6      frequency") +
7 82     geom_vline(aes(xintercept=true_pars[["r"]]), linetype="dashed
8      ", size=1, color="grey50") +
9 83     theme_bw()
10 84
11 85 print(rkernel)
12 86 ggsave(rkernel, filename="kernelr.pdf", height=3, width=3.25)
13 87
14 88 sigmapoints <- paramdata$sigma
15 89 sigmakernel <- qplot(sigmapoints, geom = "density", xlab = expression(
16     sigma), ylab = "frequency") +
17 90     geom_vline(aes(xintercept=sigma), linetype="dashed", size=1,
18     color="grey50") +
19 91     theme_bw()
20 92
21 93 print(sigmakernel)
22 94 ggsave(sigmakernel, filename="kernelsigma.pdf", height=3, width=3.25)
23 95
24 96 infecpoints <- paramdata$Iinit
25 97 infeckernel <- qplot(infecpoints, geom = "density", xlab = "Initial
26     Infected", ylab = "frequency") +
27 98     geom_vline(aes(xintercept=true_init_cond[["I"]]), linetype="
28     dashed", size=1, color="grey50") +
29 99     theme_bw()
30 100
31 101 print(infeckernel)
32 102 ggsave(infeckernel, filename="kernelinfec.pdf", height=3, width=3.25)
33 103
34 104 # show grid
35 105 grid.arrange(R0kernel, rkernel, sigmakernel, infeckernel, ncol = 2,
36     nrow = 2)
37 106
38 107 pdf("if2kernels.pdf", height = 6.5, width = 6.5)
39 108 grid.arrange(R0kernel, rkernel, sigmakernel, infeckernel, ncol = 2,
40     nrow = 2)
41 109 dev.off()
42

```

## 43 B.2 Full C++ code

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

```

45 1 /* Dexter Barrows
46 2 dbarrows.github.io
47

```

```

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

```



```

1      particle);
2 53 void copyParticle(Particle * dst, Particle * src);
3 54 void perturbParticles(Particle * particles, int N, int NP, int
4      passnum, double coolrate);
5 55 bool isCollapsed(Particle * particles, int NP);
6 56 void particleDiagnostics(ParticleInfo * partInfo, Particle *
7      particles, int NP);
8 57 NumericMatrix if2(NumericVector * data, int T, int N);
9 58 double randu();
10 59 double randn();
11 60
12 61 // [[Rcpp::export]]
13 62 NumericMatrix if2(NumericVector data, int T, int N) {
14 63
15 64     int      NP          = 2500;
16 65     int      nPasses     = 50;
17 66     double   coolrate    = 0.975;
18 67
19 68     int      i_infec     = I0;
20 69
21 70     NumericMatrix paramdata(NP, 6);
22 71
23 72     srand(time(NULL));    // Seed PRNG with system time
24 73
25 74     double w[NP];        // particle weights
26 75
27 76     Particle particles[NP]; // particle estimates for current
28     step
29 77     Particle particles_old[NP]; // intermediate particle states for
30     resampling
31 78
32 79     printf("Initializing particle states\n");
33 80
34 81     // initialize particle parameter states (seeding)
35 82     for (int n = 0; n < NP; n++) {
36 83
37 84         double R0can, rcan, sigmacan, Iinitcan;
38 85
39 86         do {
40 87             R0can = R0true + R0true*randn();
41 88         } while (R0can < 0);
42 89         particles[n].R0 = R0can;
43 90
44 91         do {
45 92             rcan = rtrue + rtrue*randn();
46 93         } while (rcan < 0);
47 94         particles[n].r = rcan;
48 95
49 96         do {
50 97             sigmacan = merr + merr*randn();
51 98         } while (sigmacan < 0);

```

```

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

```

```

1 149     }
2 150
3 151     // cumulative sum
4 152     for (int n = 1; n < NP; n++) {
5 153         w[n] += w[n-1];
6 154     }
7 155
8 156     // save particle states to resample from
9 157     for (int n = 0; n < NP; n++){
10 158         copyParticle(&particles_old[n], &particles[n]);
11 159     }
12 160
13 161     // resampling
14 162     for (int n = 0; n < NP; n++) {
15 163
16 164         double w_r = randu() * w[NP-1];
17 165         int i = 0;
18 166         while (w_r > w[i]) {
19 167             i++;
20 168         }
21 169
22 170         // i is now the index to copy state from
23 171         copyParticle(&particles[n], &particles_old[i]);
24 172
25 173     }
26 174
27 175 }
28 176
29 177 }
30 178
31 179 ParticleInfo pInfo;
32 180 particleDiagnostics(&pInfo, particles, NP);
33 181
34 182 printf("Parameter results (mean | sd)\n");
35 183 printf("-----\n");
36 184 printf("R0      %f %f\n", pInfo.R0mean, pInfo.R0sd);
37 185 printf("r      %f %f\n", pInfo.rmean, pInfo.rsd);
38 186 printf("sigma   %f %f\n", pInfo.sigamean, pInfo.sigmasd);
39 187 printf("S_init  %f %f\n", pInfo.Sinitmean, pInfo.Sinit_sd);
40 188 printf("I_init  %f %f\n", pInfo.Iinitmean, pInfo.Iinit_sd);
41 189 printf("R_init  %f %f\n", pInfo.Rinitmean, pInfo.Rinit_sd);
42 190
43 191 printf("\n");
44 192
45 193
46 194
47 195 // Get particle results to pass back to R
48 196
49 197 for (int n = 0; n < NP; n++) {
50 198
51 199     paramdata(n, 0) = particles[n].R0;

```

```

1 200         paramdata(n, 1) = particles[n].r;
2 201         paramdata(n, 2) = particles[n].sigma;
3 202         paramdata(n, 3) = particles[n].Sinit;
4 203         paramdata(n, 4) = particles[n].Iinit;
5 204         paramdata(n, 5) = particles[n].Rinit;
6 205
7 206     }
8 207
9 208     return paramdata;
10 209
11 210 }
12 211
13 212
14 213 /* Use the Explicit Euler integration scheme to integrate SIR model
15 214    forward in time
16 214    double h      - time step size
17 215    double t0     - start time
18 216    double tn     - stop time
19 217    double * y    - current system state; a three-component vector
20 218                  representing [S I R], susceptible-infected-recovered
21 218
22 219    */
23 220 void exp_euler_SIR(double h, double t0, double tn, int N, Particle *
24 221    particle) {
25 221
26 222     int num_steps = floor( (tn-t0) / h );
27 223
28 224     double S = particle->S;
29 225     double I = particle->I;
30 226     double R = particle->R;
31 227
32 228     double R0   = particle->R0;
33 229     double r    = particle->r;
34 230     double B    = R0 * r / N;
35 231
36 232     for(int i = 0; i < num_steps; i++) {
37 233         // get derivatives
38 234         double dS = - B*S*I;
39 235         double dI = B*S*I - r*I;
40 236         double dR = r*I;
41 237         // step forward by h
42 238         S += h*dS;
43 239         I += h*dI;
44 240         R += h*dR;
45 241     }
46 242
47 243     particle->S = S;
48 244     particle->I = I;
49 245     particle->R = R;
50 246
51 247 }

```

```

1 248
2 249
3 250 /* Particle pertubation function to be run between iterations and
4      passes
5 251
6 252 */
7 253 void perturbParticles(Particle * particles, int N, int NP, int
8      passnum, double coolrate) {
9 254
10 255     double coolcoef = pow(coolrate, passnum);
11 256
12 257     double spreadR0      = coolcoef * R0true / 10.0;
13 258     double spreadr       = coolcoef * rtrue / 10.0;
14 259     double spreadsigma   = coolcoef * merr / 10.0;
15 260     double spreadIinit   = coolcoef * I0 / 10.0;
16 261
17 262     double R0can, rcan, sigmacan, Iinitcan;
18 263
19 264     for (int n = 0; n < NP; n++) {
20 265
21 266         do {
22 267             R0can = particles[n].R0 + spreadR0*randn();
23 268         } while (R0can < 0);
24 269         particles[n].R0 = R0can;
25 270
26 271         do {
27 272             rcan = particles[n].r + spreadr*randn();
28 273         } while (rcan < 0);
29 274         particles[n].r = rcan;
30 275
31 276         do {
32 277             sigmacan = particles[n].sigma + spreadsigma*randn();
33 278         } while (sigmacan < 0);
34 279         particles[n].sigma = sigmacan;
35 280
36 281         do {
37 282             Iinitcan = particles[n].Iinit + spreadIinit*randn();
38 283         } while (Iinitcan < 0 || Iinitcan > 500);
39 284         particles[n].Iinit = Iinitcan;
40 285         particles[n].Sinit = N - Iinitcan;
41 286
42 287     }
43 288
44 289 }
45 290
46 291
47 292 /* Convenience function for particle resampling process
48 293
49 294 */
50 295 void copyParticle(Particle * dst, Particle * src) {
51 296

```

```

1 297     dst->R0      = src->R0;
2 298     dst->r       = src->r;
3 299     dst->sigma    = src->sigma;
4 300     dst->S        = src->S;
5 301     dst->I        = src->I;
6 302     dst->R        = src->R;
7 303     dst->Sinit     = src->Sinit;
8 304     dst->Iinit     = src->Iinit;
9 305     dst->Rinit     = src->Rinit;
10 306
11 307 }
12 308
13 309
14 310 /* Checks to see if particles are collapsed
15 311    This is done by checking if the standard deviations between the
16 312    particles' parameter
17 312    values are significantly close to one another. Spread threshold
18 312    may need to be tuned.
19 313
20 314    */
21 315 bool isCollapsed(Particle * particles, int NP) {
22 316
23 317     bool retVal;
24 318
25 319     double R0mean = 0, rmean = 0, sigmamean = 0, Sinitmean = 0,
26 319     Iinitmean = 0, Rinitmean = 0;
27 320
28 321     // means
29 322
30 323     for (int n = 0; n < NP; n++) {
31 324
32 325         R0mean      += particles[n].R0;
33 326         rmean       += particles[n].r;
34 327         sigmamean   += particles[n].sigma;
35 328         Sinitmean   += particles[n].Sinit;
36 329         Iinitmean   += particles[n].Iinit;
37 330         Rinitmean   += particles[n].Rinit;
38 331
39 332     }
40 333
41 334     R0mean      /= NP;
42 335     rmean       /= NP;
43 336     sigmamean   /= NP;
44 337     Sinitmean   /= NP;
45 338     Iinitmean   /= NP;
46 339     Rinitmean   /= NP;
47 340
48 341     double R0sd = 0, rsd = 0, sigmasd = 0, Sinitsd = 0, Iinitsd = 0,
49 341     Rinitsd = 0;
50 342
51 343     for (int n = 0; n < NP; n++) {

```

```

1 344
2 345     R0sd      += ( particles[n].R0 - R0mean ) * ( particles[n].R0 -
3          R0mean );
4 346     rsd      += ( particles[n].r - rmean ) * ( particles[n].r -
5          rmean );
6 347     sigmasd += ( particles[n].sigma - sigmamean ) * ( particles[n]
7          ].sigma - sigmamean );
8 348     Sinitdsd += ( particles[n].Sinit - Sinitmean ) * ( particles[n]
9          ].Sinit - Sinitmean );
10 349     Iinitdsd += ( particles[n].Iinit - Iinitmean ) * ( particles[n]
11          ].Iinit - Iinitmean );
12 350     Rinitdsd += ( particles[n].Rinit - Rinitmean ) * ( particles[n]
13          ].Rinit - Rinitmean );
14 351
15 352 }
16 353
17 354     R0sd      /= NP;
18 355     rsd       /= NP;
19 356     sigmasd   /= NP;
20 357     Sinitdsd  /= NP;
21 358     Iinitdsd  /= NP;
22 359     Rinitdsd  /= NP;
23 360
24 361     if ( (R0sd + rsd + sigmasd) < 1e-5)
25 362         retVal = true;
26 363     else
27 364         retVal = false;
28 365
29 366     return retVal;
30 367
31 368 }
32 369
33 370 void particleDiagnostics(ParticleInfo * partInfo, Particle *
34     particles, int NP) {
35 371
36 372     double    R0mean      = 0.0,
37 373             rmean        = 0.0,
38 374             sigmamean    = 0.0,
39 375             Sinitmean    = 0.0,
40 376             Iinitmean    = 0.0,
41 377             Rinitmean    = 0.0;
42 378
43 379     // means
44 380
45 381     for (int n = 0; n < NP; n++) {
46 382
47 383         R0mean      += particles[n].R0;
48 384         rmean       += particles[n].r;
49 385         sigmamean   += particles[n].sigma;
50 386         Sinitmean   += particles[n].Sinit;
51 387         Iinitmean   += particles[n].Iinit;

```

```

1 388         Rinitmean    += particles[n].Rinit;
2 389
3 390     }
4 391
5 392     R0mean        /= NP;
6 393     rmean         /= NP;
7 394     sigmamean     /= NP;
8 395     Sinitmean     /= NP;
9 396     Iinitmean     /= NP;
10 397     Rinitmean     /= NP;
11 398
12 399     // standard deviations
13 400
14 401     double  R0sd    = 0.0,
15 402             rsd     = 0.0,
16 403             sigmasd = 0.0,
17 404             Sinitsd = 0.0,
18 405             Iinitds = 0.0,
19 406             Rinitds = 0.0;
20 407
21 408     for (int n = 0; n < NP; n++) {
22 409
23 410         R0sd    += ( particles[n].R0 - R0mean ) * ( particles[n].R0 -
24 411                 R0mean );
25 412         rsd     += ( particles[n].r - rmean ) * ( particles[n].r -
26 413                 rmean );
27 414         sigmasd += ( particles[n].sigma - sigmamean ) * ( particles[n]
28 415                 ].sigma - sigmamean );
29 416         Sinitsd += ( particles[n].Sinit - Sinitmean ) * ( particles[n]
30 417                 ].Sinit - Sinitmean );
31 418         Iinitds += ( particles[n].Iinit - Iinitmean ) * ( particles[n]
32 419                 ].Iinit - Iinitmean );
33 420         Rinitds += ( particles[n].Rinit - Rinitmean ) * ( particles[n]
34 421                 ].Rinit - Rinitmean );
35 422
36 423     }
37 424
38 425     R0sd        /= NP;
39 426     rsd         /= NP;
40 427     sigmasd     /= NP;
41 428     Sinitsd     /= NP;
42 429     Iinitds     /= NP;
43 430     Rinitds     /= NP;
44 431
45 432     partInfo->R0mean    = R0mean;
46 433     partInfo->R0sd      = R0sd;
47 434     partInfo->sigmamean = sigmamean;
48 435     partInfo->sigmasd   = sigmasd;
49 436     partInfo->rmean     = rmean;
50 437     partInfo->rsd       = rsd;
51 438     partInfo->Sinitmean = Sinitmean;

```



```
1 433     partInfo->Sinitd    = Sinitd;
2 434     partInfo->Iinitmean = Iinitmean;
3 435     partInfo->Iinitd    = Iinitd;
4 436     partInfo->Rinitmean = Rinitmean;
5 437     partInfo->Rinitd    = Rinitd;
6 438
7 439 }
8 440
9 441 double randu() {
10 442
11 443     return (double) rand() / (double) RAND_MAX;
12 444
13 445 }
14 446
15 447
16 448 /* Return a normally distributed random number with mean 0 and
17 449    standard deviation 1
18 449    Uses the polar form of the Box-Muller transformation
19 450    From http://www.design.caltech.edu/erik/Misc/Gaussian.html
20 451    */
21 452 double randn() {
22 453
23 454     double x1, x2, w, y1;
24 455
25 456     do {
26 457         x1 = 2.0 * randu() - 1.0;
27 458         x2 = 2.0 * randu() - 1.0;
28 459         w = x1 * x1 + x2 * x2;
29 460     } while ( w >= 1.0 );
30 461
31 462     w = sqrt( (-2.0 * log( w ) ) / w );
32 463     y1 = x1 * w;
33 464
34 465     return y1;
35 466
36 467 }
```

# 1 Appendix C

## 2 Parameter Fitting

### 3 C.1 SIR Forward Simulator

4 The basic Stochastic SIR model simulation function.

```
5 1 ## Dexter Barrows
6 2 ## dbarrows.github.com
7 3 ## McMaster University
8 4 ## 2016
9 5
10 6 StocSIR ← function(y, pars, T, steps) {
11 7
12 8     out ← matrix(NA, nrow = (T+1), ncol = 4)
13 9
14 10    R0 ← pars[['R0']]
15 11    r ← pars[['r']]
16 12    N ← pars[['N']]
17 13    eta ← pars[['eta']]
18 14    berr ← pars[['berr']]
19 15
20 16    S ← y[['S']]
21 17    I ← y[['I']]
22 18    R ← y[['R']]
23 19
24 20    B0 ← R0 * r / N
25 21    B ← B0
26 22
27 23    out[1,] ← c(S,I,R,B)
28 24
29 25    h ← 1 / steps
30 26
31 27    for ( i in 1:(T*steps) ) {
32 28
```

```

1 29      B ← exp( log(B) + eta*(log(B0) - log(B)) + rnorm(1, 0, berr)
2      )
3 30
4 31      BSI ← B*S*I
5 32      rI ← r*I
6 33
7 34      dS ← -BSI
8 35      dI ← BSI - rI
9 36      dR ← rI
10 37
11 38      S ← S + h*dS
12 39      I ← I + h*dI
13 40      R ← R + h*dR
14 41
15 42      if (i %% steps == 0)
16 43          out[i/steps+1,] ← c(S,I,R,B)
17 44
18 45  }
19 46
20 47  return(out)
21 48
22 49 }
23 50
24 51 ### Suggested parameters
25 52 #
26 53 # T          ← 60
27 54 # i_infec ← 5
28 55 # steps    ← 7
29 56 # N          ← 500
30 57 # sigma     ← 10
31 58 #
32 59 # pars ← c(R0 = 3.0,      # new infected people per infected person
33 60 #           r = 0.1,      # recovery rate
34 61 #           N = 500,      # population size
35 62 #           eta = 0.5,    # geometric random walk
36 63 #           berr = 0.5)   # Beta geometric walk noise

```

# 1 Appendix D

## 2 Forecasting Frameworks

### 3 D.1 IF2 Parametric Bootstrapping Function

4 The parametric bootstrapping machinery used to produce forecasts.

```
5 1 # Dexter Barrows
6 2 # dbarrows.github.io
7 3 # McMaster University
8 4 # 2016
9 5 #
10 6 # IF2 parametric bootstrapping function
11 7
12 8 library(foreach)
13 9 library(parallel)
14 10 library(doParallel)
15 11 library(Rcpp)
16 12
17 13 if2_paraboot ← function(if2data_parent, T, Tlim, steps, N, nTrials,
18 14 if2file, if2_s_file, stoc_sir_file, NP, nPasses, coolrate) {
19 15
20 16   source(stoc_sir_file)
21 17
22 18   if (nTrials < 2)
23 19     ntrials ← 2
24 20
25 21   # unpack if2 first fit data
26 22   # ...parameters
27 23   paramdata_parent ← data.frame( if2data_parent$paramdata )
28 24   names(paramdata_parent) ← c("R0", "r", "sigma", "eta", "berr", "
29 25     Sinit", "Iinit", "Rinit")
30 26   parmeans_parent ← colMeans(paramdata_parent)
31 27   names(parmmeans_parent) ← c("R0", "r", "sigma", "eta", "berr", "
32 28     Sinit", "Iinit", "Rinit")
33 29   # ...states
34 30 }
```

```

1 27 statedata_parent <- data.frame( if2data_parent$statedata )
2 28 names(statedata_parent) <- c("S", "I", "R", "B")
3 29 statemeans_parent <- colMeans(statedata_parent)
4 30 names(statemeans_parent) <- c("S", "I", "R", "B")
5 31
6 32
7 33 ## use parametric bootstrapping to generate forecasts
8 34 ##
9 35 trajectories <- foreach( i = 1:nTrials, .combine = rbind, .packages
10   = "Rcpp") %dopar% {
11 36
12 37   source(stoc_sir_file)
13 38
14 39   ## draw new data
15 40   ##
16 41
17 42   pars <- with( as.list(parmmeans_parent),
18 43               c(R0 = R0,
19 44                 r = r,
20 45                 N = N,
21 46                 eta = eta,
22 47                 berr = berr) )
23 48
24 49   init_cond <- with( as.list(parmmeans_parent),
25 50                   c(S = Sinit,
26 51                     I = Iinit,
27 52                     R = Rinit) )
28 53
29 54   # generate trajectory
30 55   sdeout <- StocSIR(init_cond, pars, Tlim + 1, steps)
31 56   colnames(sdeout) <- c('S', 'I', 'R', 'B')
32 57
33 58   # add noise
34 59   counts_raw <- sdeout[, 'I'] + rnorm(dim(sdeout)[1], 0, parmeans_
35   parent[['sigma']])
36 60   counts <- ifelse(counts_raw < 0, 0, counts_raw)
37 61
38 62   ## refit using new data
39 63   ##
40 64
41 65   rm(if2) # because stupid things get done in packages
42 66   sourceCpp(if2file)
43 67   if2time <- system.time( if2data <- if2(counts, Tlim+1, N, NP,
44   nPasses, coolrate) )
45 68
46 69   paramdata <- data.frame( if2data$paramdata )
47 70   names(paramdata) <- c("R0", "r", "sigma", "eta", "berr", "Sinit",
48   "Iinit", "Rinit")
49 71   parmeans <- colMeans(paramdata)
50 72   names(parmeans) <- c("R0", "r", "sigma", "eta", "berr", "Sinit", "
51   Iinit", "Rinit")

```

```

1  73
2  74   ## generate the rest of the trajectory
3  75   ##
4  76
5  77   # pack new parameter estimates
6  78   pars ← with( as.list(parmmeans),
7  79               c(R0 = R0,
8  80                 r = r,
9  81                 N = N,
10 82                 eta = eta,
11 83                 berr = berr) )
12 84   init_cond ← c(S = statemeans_parent[['S']],
13 85               I = statemeans_parent[['I']],
14 86               R = statemeans_parent[['R']])
15 87
16 88   # generate remaining trajectory part
17 89   sdeout_future ← StocSIR(init_cond, pars, T-Tlim, steps)
18 90   colnames(sdeout_future) ← c('S','I','R','B')
19 91
20 92   return ( c( counts = unname(sdeout_future[, 'I']),
21 93             parmeans,
22 94             time = if2time[['user.self']]) )
23 95
24 96
25 97 }
26 98
27 99 return(trajectories)
28 100
29 101 }

```

## 31 D.2 RStan Forward Simulator

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

```

34 1  ## Dexter Barrows
35 2  ## dbarrows.github.io
36 3  ## McMaster University
37 4  ## 2016
38 5
39 6 StocSIRstan ← function(y, pars, T, steps, berrvec, bveclim) {
40 7
41 8   out ← matrix(NA, nrow = (T+1), ncol = 4)
42 9
43 10  R0 ← pars[['R0']]
44 11  r ← pars[['r']]
45 12  N ← pars[['N']]
46 13  eta ← pars[['eta']]

```

```

1 14 berr ← pars[['berr']]
2 15
3 16 S ← y[['S']]
4 17 I ← y[['I']]
5 18 R ← y[['R']]
6 19
7 20 B0 ← R0 * r / N
8 21 B ← B0
9 22
10 23 out[1,] ← c(S,I,R,B)
11 24
12 25 h ← 1 / steps
13 26
14 27 for ( i in 1:(T*steps) ) {
15 28
16 29     if (i <= bveclim) {
17 30         B ← exp( log(B) + eta*(log(B0) - log(B)) + berrvec[i])
18 31     } else {
19 32         B ← exp( log(B) + eta*(log(B0) - log(B)) + rnorm(1, 0, berr
20 33         ))
21 34     }
22 35
23 36 BSI ← B*S*I
24 37 rI ← r*I
25 38
26 39 dS ← -BSI
27 40 dI ← BSI - rI
28 41 dR ← rI
29 42
30 43 S ← S + h*dS
31 44 I ← I + h*dI
32 45 R ← R + h*dR
33 46
34 47 if (i %% steps == 0)
35 48     out[i/steps+1,] ← c(S,I,R,B)
36 49
37 50 }
38 51
39 52 return(out)
40 53
41 54 }
42 55

```

# 1 Appendix E

## 2 S-map and SIRS

### 3 E.1 SIRS R Function Code

4 R code to simulate the outlined SIRS function.

```
5  
6 1 ## Dexter Barrows  
7 2 ## dbarrows.github.io  
8 3 ## McMaster University  
9 4 ## 2016  
10 5  
11 6 StocSIRS ← function(y, pars, T, steps) {  
12 7  
13 8   out ← matrix(NA, nrow = (T+1), ncol = 4)  
14 9  
15 10  R0 ← pars[['R0']]  
16 11  r ← pars[['r']]  
17 12  N ← pars[['N']]  
18 13  eta ← pars[['eta']]  
19 14  berr ← pars[['berr']]  
20 15  re ← pars[['re']]  
21 16  
22 17  S ← y[['S']]  
23 18  I ← y[['I']]  
24 19  R ← y[['R']]  
25 20  
26 21  B0 ← R0 * r / N  
27 22  B ← B0  
28 23  
29 24  out[1,] ← c(S,I,R,B)  
30 25  
31 26  h ← 1 / steps  
32 27  
33 28  for ( i in 1:(T*steps) ) {  
34 29
```



```

1 30      #Bfac ← 1/2 - cos((2*pi/365)*i)/2
2 31      Bfac ← exp(2*cos((2*pi/365)*i) - 2)
3 32
4 33      B ← exp( log(B) + eta*(log(B0) - log(B)) + rnorm(1, 0, berr) )
5 34
6 35      BSI ← Bfac*B*S*I
7 36      rI ← r*I
8 37      reR ← re*R
9 38
10 39      dS ← -BSI + reR
11 40      dI ← BSI - rI
12 41      dR ← rI - reR
13 42
14 43      S ← S + h*dS #newInf
15 44      I ← I + h*dI #newInf - h*dR
16 45      R ← R + h*dR #h*dR
17 46
18 47      if (i %% steps == 0)
19 48          out[i/steps+1,] ← c(S,I,R,B)
20 49
21 50  }
22 51
23 52  colnames(out) ← c("S","I","R","B")
24 53  return(out)
25 54
26 55 }
27 56
28 57 ### suggested parameters
29 58 #
30 59 # T      ← 200
31 60 # i_infec ← 10
32 61 # steps  ← 7
33 62 # N      ← 500
34 63 # sigma  ← 5
35 64 #
36 65 # pars ← c(R0 = 3.0, # new infected people per infected person
37 66 #           r = 0.1, # recovery rate
38 67 #           N = 500, # population size
39 68 #           eta = 0.5, # geometric random walk
40 69 #           berr = 0.5, # Beta geometric walk noise
41 70 #           re = 1) # resuceptibility rate
42

```

## 43 E.2 SIRS HMC R Function Code

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

```

45 1 ## Dexter Barrows
46 2 ## dbarrows.github.io
47

```

```

1  3 ## McMaster University
2  4 ## 2016
3  5
4  6 StocSIRSstan ← function(y, pars, T, steps, berrvec, bveclim) {
5  7
6  8   out ← matrix(NA, nrow = (T+1), ncol = 4)
7  9
8  10  R0 ← pars[['R0']]
9  11  r ← pars[['r']]
10 12  N ← pars[['N']]
11 13  eta ← pars[['eta']]
12 14  berr ← pars[['berr']]
13 15  re ← pars[['re']]
14 16
15 17  S ← y[['S']]
16 18  I ← y[['I']]
17 19  R ← y[['R']]
18 20
19 21  B0 ← R0 * r / N
20 22  B ← B0
21 23
22 24  out[1,] ← c(S,I,R,B)
23 25
24 26  h ← 1 / steps
25 27
26 28  for ( i in 1:(T*steps) ) {
27 29
28 30    Bfac ← exp(2*cos((2*pi/365)*i) - 2)
29 31
30 32    if (i <= bveclim) {
31 33      B ← exp( log(B) + eta*(log(B0) - log(B)) + berrvec[i])
32 34    } else {
33 35      B ← exp( log(B) + eta*(log(B0) - log(B)) + rnorm(1, 0, berr
34 36      ))
35 37    }
36 38
37 39    BSI ← Bfac*B*S*I
38 40    rI ← r*I
39 41    reR ← re*R
40 42
41 43    dS ← -BSI + reR
42 44    dI ← BSI - rI
43 45    dR ← rI -reR
44 46
45 47    S ← S + h*dS #newInf
46 48    I ← I + h*dI #newInf - h*dR
47 49    R ← R + h*dR #h*dR
48 50
49 51    if (i %% steps == 0)
50 52      out[i/steps+1,] ← c(S,I,R,B)
51 53

```

```

1 53 }
2 54
3 55 return(out)
4 56
5 57 }
6 58
7 59 ### suggested parameters
8 60 #
9 61 # T      ← 200
10 62 # i_infec ← 5
11 63 # steps  ← 7
12 64 # N      ← 500
13 65 # sigma  ← 5
14 66 #
15 67 # pars ← c(R0 = 3.0, # new infected people per infected person
16 68 #          r = 0.1, # recovery rate
17 69 #          gam = 2, # new infected shock intensity
18 70 #          N = 500, # population size
19 71 #          eta = 0.5, # geometric random walk
20 72 #          berr = 0.5, # Beta geometric walk noise
21 73 #          re = 2) # resuceptibility rate
22

```

### 23 E.3 SMAP Code

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

```

25
26 1 ## Dexter Barrows
27 2 ## dbarrows.github.io
28 3 ## McMaster University
29 4 ## 2016
30 5
31 6 library(pracma) # needed for tiling function
32 7
33 8 smap ← function(data, E, theta, stepsAhead) {
34 9
35 10 # construct library
36 11 tseries ← as.vector(data)
37 12 liblen ← length(tseries) - E + 1 - stepsAhead
38 13 lib ← matrix(NA, liblen, E)
39 14
40 15 for (i in 1:E) {
41 16 lib[,i] ← tseries[(E-i+1):(liblen+E-i)]
42 17 }
43 18
44 19 # predict from the last index
45 20 tslen ← length(tseries)
46 21 predictee ← rev(t(as.matrix(tseries[(tslen-E+1):tslen])))
47 22 predictions ← numeric(stepsAhead)

```

```

1 23
2 24   # for each prediction index (number of steps ahead)
3 25   for(i in 1:stepsAhead) {
4 26
5 27       # set up weight calculation
6 28       predmat ← repmat(predictee, liblen, 1)
7 29       distances ← sqrt( rowSums( abs(lib - predmat)^2 ) )
8 30       meanDist ← mean(distances)
9 31
10 32       # calculate weights
11 33       weights ← exp( - (theta * distances) / meanDist )
12 34
13 35       # construct A, B
14 36
15 37       preds ← tseries[(E+i):(liblen+E+i-1)]
16 38
17 39       A ← cbind( rep(1.0, liblen), lib ) * repmat(as.matrix(weights
18 40           ), 1, E+1)
19 41       B ← as.matrix(preds * weights)
20 42
21 43       # solve system for C
22 44
23 45       Asvd ← svd(A)
24 46       C ← Asvd$v %*% diag(1/Asvd$d) %*% t(Asvd$u) %*% B
25 47
26 48       # get prediction
27 49
28 50       predsum ← sum(C * c(1,predictee))
29 51
30 52       # save
31 53
32 54       predictions[i] ← predsum
33 55   }
34 56
35 57   return(predictions)
36 58
37 59 }
38 59

```

## 40 E.4 SMAP Parameter Optimization Code

41 This code determines the optimal parameter values to be used by the S-map algo-  
42 rithm.

```

43 1 ## Dexter Barrows
44 2 ## dbarrows.github.io
45 3 ## McMaster University
46 4 ## 2016

```

```

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

```

```

1 55      E ← Elist[i]
2 56      theta ← thetalist[j]
3 57
4 58      ## get true trajectory
5 59      ##
6 60      sdeout ← StocSIRS(true_init_cond, true_pars, T, steps)
7 61
8 62      ## perturb to get data
9 63      ##
10 64      infec_counts_raw ← sdeout[1:(Tlim+1), 'I'] + rnorm(Tlim+1, 0,
11      sigma)
12 65      infec_counts ← ifelse(infec_counts_raw < 0, 0, infec_counts_
13      raw)
14 66
15 67      predictions ← smap(infec_counts, E, theta, 52)
16 68
17 69      err ← sdeout[(Tlim+2):dim(sdeout)[1], 'I'] - predictions
18 70      sse ← sum(err^2)
19 71
20 72      ssemean ← ssemean + (sse / nTrials)
21 73
22 74  }
23 75
24 76      ssemat[i,j] ← ssemean
25 77
26 78
27 79  }
28 80 }
29 81
30 82 quartz()
31 83 image(-ssemat)
32 84 quartz()
33 85 filled.contour(-ssemat)
34 86
35 87 mininds ← which(ssemat==min(ssemat), arr.ind=TRUE)
36 88
37 89 Emin ← Elist[mininds[, 'row']]
38 90 thetamin ← thetalist[mininds[, 'col']]
39 91
40 92 print(Emin)
41 93 print(thetamin)
42

```

## 43 E.5 RStan SIRS Code

44 This code implements a periodic SIRS model in Rstan.

```

45 1 ## Dexter Barrows
46 2 ## dbarrows.github.io
47

```

```

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

```

```

1  53      B[t] ← exp( log(B0) + eta * ( log(B[t-1]) - log(B0) ) +
2          Bnoise[t] );
3  54
4  55      S[t] ← S[t-1] + h*( - Bfac*B[t]*S[t-1]*I[t-1] + re*R[t-1] );
5  56      I[t] ← I[t-1] + h*( Bfac*B[t]*S[t-1]*I[t-1] - I[t-1]*r );
6  57      R[t] ← R[t-1] + h*( I[t-1]*r - re*R[t-1] );
7  58
8  59      if (y[t] > 0) {
9  60          y[t] ~ normal( I[t], sigma );
10 61      }
11 62
12 63  }
13 64
14 65      R0      ~ lognormal(1,1);
15 66      r        ~ lognormal(1,1);
16 67      sigma    ~ lognormal(1,1);
17 68      re       ~ lognormal(1,1);
18 69      Iinit    ~ normal(y[1], sigma);
19 70
20 71 }

```

## 22 E.6 IF2 SIRS Code

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

```

24
25 1  /* Dexter Barrows
26 2     dbarrows.github.io
27 3     McMaster University
28 4     2016
29 5
30 6     */
31 7
32 8 #include <stdio.h>
33 9 #include <math.h>
34 10 #include <sys/time.h>
35 11 #include <time.h>
36 12 #include <stdlib.h>
37 13 #include <string>
38 14 #include <cmath>
39 15 #include <cstdlib>
40 16 #include <fstream>
41 17
42 18 #define Treal      100          // time to simulate over
43 19 #define R0true     3.0          // infectiousness
44 20 #define rtrue      0.1          // recovery rate
45 21 #define retrue     0.05         // resusceptibility rate
46 22 #define Nreal      500.0        // population size
47 23 #define etatrue     0.5          // real drift attraction strength

```



```

1 24 #define berrtrue    0.5          // real beta drift noise
2 25 #define merr      5.0          // expected measurement error
3 26 #define I0         5.0          // Initial infected individuals
4 27
5 28 #define PSC         0.5          // scale factor for more sensitive
6    parameters
7 29
8 30 #include <Rcpp.h>
9 31 using namespace Rcpp;
10 32
11 33 struct State {
12 34     double S;
13 35     double I;
14 36     double R;
15 37 };
16 38
17 39 struct Particle {
18 40     double R0;
19 41     double r;
20 42     double re;
21 43     double sigma;
22 44     double eta;
23 45     double berr;
24 46     double B;
25 47     double S;
26 48     double I;
27 49     double R;
28 50     double Sinit;
29 51     double Iinit;
30 52     double Rinit;
31 53 };
32 54
33 55 struct ParticleInfo {
34 56     double R0mean;    double R0sd;
35 57     double rmean;     double rsd;
36 58     double remean;    double resd;
37 59     double sigmamean; double sigmasd;
38 60     double etamean;   double etasd;
39 61     double berrmean;  double berrsd;
40 62     double Sinitmean; double Sinitsd;
41 63     double Iinitmean; double Iinitsd;
42 64     double Rinitmean; double Rinitsd;
43 65 };
44 66
45 67
46 68 int timeval_subtract (double *result, struct timeval *x, struct
47    timeval *y);
48 69 int check_double(double x,double y);
49 70 void exp_euler_SIRS(double h, double t0, double tn, int N, Particle *
50    particle);
51 71 void copyParticle(Particle * dst, Particle * src);

```

```

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

```

```

1 117         particles[n].re = recan;
2 118
3 119         particles[n].B = (double) R0can * rcan / N;
4 120
5 121         do {
6 122             sigmacan = merr + merr*randn();
7 123         } while (sigmacan < 0);
8 124         particles[n].sigma = sigmacan;
9 125
10 126        do {
11 127            etacan = etatrue + PSC*etatrue*randn();
12 128        } while (etacan < 0 || etacan > 1);
13 129        particles[n].eta = etacan;
14 130
15 131        do {
16 132            berrcan = berrtrue + PSC*berrtrue*randn();
17 133        } while (berrcan < 0);
18 134        particles[n].berr = berrcan;
19 135
20 136        do {
21 137            Iinitcan = I0 + I0*randn();
22 138        } while (Iinitcan < 0 || N < Iinitcan);
23 139        particles[n].Sinit = N - Iinitcan;
24 140        particles[n].Iinit = Iinitcan;
25 141        particles[n].Rinit = 0.0;
26 142
27 143    }
28 144
29 145    // START PASSES THROUGH DATA
30 146
31 147    printf("Starting filter\n");
32 148    printf("-----\n");
33 149    printf("Pass\n");
34 150
35 151
36 152    for (int pass = 0; pass < nPasses; pass++) {
37 153
38 154        printf("...%d / %d\n", pass, nPasses);
39 155
40 156        // reset particle system evolution states
41 157        for (int n = 0; n < NP; n++) {
42 158
43 159            particles[n].S = particles[n].Sinit;
44 160            particles[n].I = particles[n].Iinit;
45 161            particles[n].R = particles[n].Rinit;
46 162            particles[n].B = (double) particles[n].R0 * particles[n].
47            r / N;
48 163
49 164        }
50 165
51 166        if (pass == (nPasses-1)) {

```

```

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

```

```

1 215
2 216         if (pass == (nPasses-1)) {
3 217             State sMeans;
4 218             getStateMeans(&sMeans, particles, NP);
5 219             statemeans(t,0) = sMeans.S;
6 220             statemeans(t,1) = sMeans.I;
7 221             statemeans(t,2) = sMeans.R;
8 222         }
9 223
10 224     }
11 225
12 226     ParticleInfo pInfo;
13 227     particleDiagnostics(&pInfo, particles, NP);
14 228
15 229     means(pass, 0) = pInfo.R0mean;
16 230     means(pass, 1) = pInfo.rmean;
17 231     means(pass, 2) = pInfo.remean;
18 232     means(pass, 3) = pInfo.sigamean;
19 233     means(pass, 4) = pInfo.etamean;
20 234     means(pass, 5) = pInfo.berrmean;
21 235     means(pass, 6) = pInfo.Sinitmean;
22 236     means(pass, 7) = pInfo.Iinitmean;
23 237     means(pass, 8) = pInfo.Rinitmean;
24 238
25 239     sds(pass, 0) = pInfo.R0sd;
26 240     sds(pass, 1) = pInfo.rsd;
27 241     sds(pass, 2) = pInfo.resd;
28 242     sds(pass, 3) = pInfo.sigmasd;
29 243     sds(pass, 4) = pInfo.etasd;
30 244     sds(pass, 5) = pInfo.berrsd;
31 245     sds(pass, 6) = pInfo.Sinitsd;
32 246     sds(pass, 7) = pInfo.Iinitsd;
33 247     sds(pass, 8) = pInfo.Rinitsd;
34 248
35 249     // between-pass perturbations, not after last pass
36 250     if (pass < (nPasses + 1))
37 251         perturbParticles(particles, N, NP, pass, coolrate);
38 252
39 253 }
40 254
41 255 ParticleInfo pInfo;
42 256 particleDiagnostics(&pInfo, particles, NP);
43 257
44 258 printf("Parameter results (mean | sd)\n");
45 259 printf("-----\n");
46 260 printf("R0          %f %f\n", pInfo.R0mean, pInfo.R0sd);
47 261 printf("r          %f %f\n", pInfo.rmean, pInfo.rsd);
48 262 printf("re          %f %f\n", pInfo.remean, pInfo.resd);
49 263 printf("sigma        %f %f\n", pInfo.sigamean, pInfo.sigmasd);
50 264 printf("eta          %f %f\n", pInfo.etamean, pInfo.etasd);
51 265 printf("berr         %f %f\n", pInfo.berrmean, pInfo.berrsd);

```

```

1 266     printf("S_init   %f %f\n", pInfo.Sinitmean, pInfo.Sinitstd);
2 267     printf("I_init   %f %f\n", pInfo.Iinitmean, pInfo.Iinitstd);
3 268     printf("R_init   %f %f\n", pInfo.Rinitmean, pInfo.Rinitstd);
4 269
5 270     printf("\n");
6 271
7 272     // Get particle results to pass back to R
8 273
9 274     for (int n = 0; n < NP; n++) {
10 275
11 276         paramdata(n, 0) = particles[n].R0;
12 277         paramdata(n, 1) = particles[n].r;
13 278         paramdata(n, 2) = particles[n].re;
14 279         paramdata(n, 3) = particles[n].sigma;
15 280         paramdata(n, 4) = particles[n].eta;
16 281         paramdata(n, 5) = particles[n].berr;
17 282         paramdata(n, 6) = particles[n].Sinit;
18 283         paramdata(n, 7) = particles[n].Iinit;
19 284         paramdata(n, 8) = particles[n].Rinit;
20 285
21 286     }
22 287
23 288     for (int n = 0; n < NP; n++) {
24 289
25 290         statedata(n, 0) = particles[n].S;
26 291         statedata(n, 1) = particles[n].I;
27 292         statedata(n, 2) = particles[n].R;
28 293         statedata(n, 3) = particles[n].B;
29 294
30 295     }
31 296
32 297     return Rcpp::List::create( Rcpp::Named("paramdata") = paramdata,
33 298                               Rcpp::Named("means") = means,
34 299                               Rcpp::Named("statemeans") =
35                                     statemeans,
36 300                               Rcpp::Named("statedata") = statedata,
37 301                               Rcpp::Named("sds") = sds);
38 302
39 303 }
40 304
41 305
42 306 /* Use the Explicit Euler integration scheme to integrate SIR model
43     forward in time
44 307     double h      - time step size
45 308     double t0     - start time
46 309     double tn     - stop time
47 310     double * y    - current system state; a three-component vector
48                     representing [S I R], susceptible-infected-recovered
49 311
50 312 */
51 313 void exp_euler_SIRS(double h, double t0, double tn, int N, Particle *

```

```

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

```

```

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

```



```

1 412         do {
2 413             Iinitcan = particles[n].Iinit + spreadIinit*randn();
3 414         } while (Iinitcan < 0 || Iinitcan > 500);
4 415         particles[n].Iinit = Iinitcan;
5 416         particles[n].Sinit = N - Iinitcan;
6 417     }
7 418 }
8 419
9 420 }
10 421
11 422
12 423 /*  Convenience function for particle resampling process
13 424
14 425  */
15 426 void copyParticle(Particle * dst, Particle * src) {
16 427
17 428     dst->R0      = src->R0;
18 429     dst->r        = src->r;
19 430     dst->re       = src->re;
20 431     dst->sigma    = src->sigma;
21 432     dst->eta      = src->eta;
22 433     dst->berr     = src->berr;
23 434     dst->B        = src->B;
24 435     dst->S        = src->S;
25 436     dst->I        = src->I;
26 437     dst->R        = src->R;
27 438     dst->Sinit    = src->Sinit;
28 439     dst->Iinit    = src->Iinit;
29 440     dst->Rinit    = src->Rinit;
30 441
31 442 }
32 443
33 444 void particleDiagnostics(ParticleInfo * partInfo, Particle *
34 445     particles, int NP) {
35 446
36 447     double  R0mean      = 0.0,
37 448            rmean       = 0.0,
38 449            remean      = 0.0,
39 450            sigmamean   = 0.0,
40 451            etamean     = 0.0,
41 452            berrmean    = 0.0,
42 453            Sinitmean   = 0.0,
43 454            Iinitmean   = 0.0,
44 455            Rinitmean   = 0.0;
45 456
46 457     // means
47 458
48 459     for (int n = 0; n < NP; n++) {
49 460
50 461         R0mean      += particles[n].R0;
51 462         rmean       += particles[n].r;

```

```

1 462         remean      += particles[n].re;
2 463         etamean     += particles[n].eta,
3 464         berrmean     += particles[n].berr,
4 465         sigmamean    += particles[n].sigma;
5 466         Sinitmean    += particles[n].Sinit;
6 467         Iinitmean    += particles[n].Iinit;
7 468         Rinitmean    += particles[n].Rinit;
8 469
9 470     }
10 471
11 472     R0mean      /= NP;
12 473     rmean       /= NP;
13 474     remean      /= NP;
14 475     sigmamean   /= NP;
15 476     etamean     /= NP;
16 477     berrmean    /= NP;
17 478     Sinitmean   /= NP;
18 479     Iinitmean   /= NP;
19 480     Rinitmean   /= NP;
20 481
21 482     // standard deviations
22 483
23 484     double  R0sd    = 0.0,
24 485             rsd     = 0.0,
25 486             resd    = 0.0,
26 487             sigmasd = 0.0,
27 488             etasd   = 0.0,
28 489             berrsd  = 0.0,
29 490             Sinitsd = 0.0,
30 491             Iinitsd = 0.0,
31 492             Rinitsd = 0.0;
32 493
33 494     for (int n = 0; n < NP; n++) {
34 495
35 496         R0sd    += ( particles[n].R0 - R0mean ) * ( particles[n].R0 -
36 497                 R0mean );
37 497         rsd     += ( particles[n].r - rmean ) * ( particles[n].r -
38 498                 rmean );
39 498         resd    += ( particles[n].re - rmean ) * ( particles[n].re -
40 499                 rmean );
41 499         sigmasd += ( particles[n].sigma - sigmamean ) * ( particles[n]
42 500                 ].sigma - sigmamean );
43 500         etasd   += ( particles[n].eta - etamean ) * ( particles[n].
44 501                 eta - etamean );
45 501         berrsd  += ( particles[n].berr - berrmean ) * ( particles[n].
46 502                 berr - berrmean );
47 502         Sinitsd += ( particles[n].Sinit - Sinitmean ) * ( particles[n]
48 503                 ].Sinit - Sinitmean );
49 503         Iinitsd += ( particles[n].Iinit - Iinitmean ) * ( particles[n]
50 504                 ].Iinit - Iinitmean );
51 504         Rinitsd += ( particles[n].Rinit - Rinitmean ) * ( particles[n]

```

```

1         ].Rinit - Rinitmean );
2 505
3 506     }
4 507
5 508     R0sd      /= NP;
6 509     rsd       /= NP;
7 510     resd      /= NP;
8 511     sigmasd   /= NP;
9 512     etasd     /= NP;
10 513     berrsd    /= NP;
11 514     Sinitsd   /= NP;
12 515     Iinitsd   /= NP;
13 516     Rinitsd   /= NP;
14 517
15 518     partInfo->R0mean    = R0mean;
16 519     partInfo->R0sd      = R0sd;
17 520     partInfo->rmean     = rmean;
18 521     partInfo->rsd       = rsd;
19 522     partInfo->remean    = remean;
20 523     partInfo->resd      = resd;
21 524     partInfo->sigmamean = sigmamean;
22 525     partInfo->sigmasd   = sigmasd;
23 526     partInfo->etamean   = etamean;
24 527     partInfo->etasd     = etasd;
25 528     partInfo->berrmean   = berrmean;
26 529     partInfo->berrsd    = berrsd;
27 530     partInfo->Sinitmean = Sinitmean;
28 531     partInfo->Sinitsd   = Sinitsd;
29 532     partInfo->Iinitmean = Iinitmean;
30 533     partInfo->Iinitsd   = Iinitsd;
31 534     partInfo->Rinitmean = Rinitmean;
32 535     partInfo->Rinitsd   = Rinitsd;
33 536
34 537 }
35 538
36 539 double randu() {
37 540
38 541     return (double) rand() / (double) RAND_MAX;
39 542
40 543 }
41 544
42 545 void getStateMeans(State * state, Particle* particles, int NP) {
43 546
44 547     double Smean = 0, Imean = 0, Rmean = 0;
45 548
46 549     for (int n = 0; n < NP; n++) {
47 550         Smean += particles[n].S;
48 551         Imean += particles[n].I;
49 552         Rmean += particles[n].R;
50 553     }
51 554

```

```
1 555     state->S = (double) Smean / NP;
2 556     state->I = (double) Imean / NP;
3 557     state->R = (double) Rmean / NP;
4 558
5 559 }
6 560
7 561
8 562 /*  Return a normally distributed random number with mean 0 and
9      standard deviation 1
10 563     Uses the polar form of the Box-Muller transformation
11 564     From http://www.design.caltech.edu/erik/Misc/Gaussian.html
12 565     */
13 566 double randn() {
14 567
15 568     double x1, x2, w, y1;
16 569
17 570     do {
18 571         x1 = 2.0 * randu() - 1.0;
19 572         x2 = 2.0 * randu() - 1.0;
20 573         w = x1 * x1 + x2 * x2;
21 574     } while ( w >= 1.0 );
22 575
23 576     w = sqrt( (-2.0 * log( w ) ) / w );
24 577     y1 = x1 * w;
25 578
26 579     return y1;
27 580
28 581 }
```

# 1 Appendix F

## 2 Spatial Epidemics

### 3 F.1 Spatial SIR R Function Code

4 R code to simulate the outlined Spatial SIR function.

```
5  
6 1 ## Dexter Barrows  
7 2 ## dbarrows.github.io  
8 3 ## McMaster University  
9 4 ## 2016  
10 5  
11 6 ## ymat: Contains the initial conditions where:  
12 7 # - rows are locations  
13 8 # - columns are S, I, R  
14 9 ## pars: Contains the parameters: global values for  $R_0$ ,  $r$ ,  $N$ ,  $\eta$ ,  
15 10 berr  
16 11 ## T: The stop time. Since 0 is included, there should be T+1  
17 12 time steps in the simulation  
18 13 ## neinum: Number of neighbors for each location, in order  
19 14 ## neibmat: Contains lists of neighbors for each location  
20 15 # - rows are parent locations (nodes)  
21 16 # - columns are locations each parent is attached to (edges)  
22 17  
23 18 StocSSIR ← function(ymat, pars, T, steps, neinum, neibmat) {  
24 19  
25 20 ## number of locations  
26 21 nloc ← dim(ymat)[1]  
27 22  
28 23 ## storage  
29 24 ## dims are locations, (S,I,R,B), times  
30 25 # output array  
31 26 out ← array(NA, c(nloc, 4, T+1), dimnames = list(NULL, c("S", "I",  
32 "R", "B"), NULL))  
33 27  
34 28 # temp storage
```

```

1 27 BSI ← numeric(nloc)
2 28 rI ← numeric(nloc)
3 29
4 30 ## extract parameters
5 31 R0 ← pars[['R0']]
6 32 r ← pars[['r']]
7 33 N ← pars[['N']]
8 34 eta ← pars[['eta']]
9 35 berr ← pars[['berr']]
10 36 phi ← pars[['phi']]
11 37
12 38 B0 ← rep(R0*r/N, nloc)
13 39
14 40 ## state vectors
15 41 S ← ymat[, 'S']
16 42 I ← ymat[, 'I']
17 43 R ← ymat[, 'R']
18 44 B ← B0
19 45
20 46 ## assign starting to output matrix
21 47 out[, , 1] ← cbind(ymat, B0)
22 48
23 49 h ← 1 / steps
24 50
25 51 for ( i in 1:(T*steps) ) {
26 52
27 53     B ← exp( log(B) + eta*(log(B0) - log(B)) + rnorm(nloc, 0,
28 54 berr) )
29 54
30 55     for (loc in 1:nloc) {
31 56         n ← neinum[loc]
32 57         sphI ← 1 - phi*(n/(n+1))
33 58         ophi ← phi/(n+1)
34 59         nBIsu ← B[neibmat[loc, 1:n]] %*% I[neibmat[loc, 1:n]]
35 60         BSI[loc] ← S[loc]*( sphI*B[loc]*I[loc] + ophi*nBIsu )
36 61     }
37 62
38 63     rI ← r*I
39 64
40 65     dS ← -BSI
41 66     dI ← BSI - rI
42 67     dR ← rI
43 68
44 69     S ← S + h*dS
45 70     I ← I + h*dI
46 71     R ← R + h*dR
47 72
48 73     if (i %% steps == 0) {
49 74         out[, , i/steps+1] ← cbind(S, I, R, B)
50 75     }
51 76

```

```

1 77     }
2 78
3 79     return(out)
4 80
5 81 }
6 82
7 83 ### Suggested parameters
8 84 #
9 85 # T          ← 60
10 86 # i_infec ← 5
11 87 # steps    ← 7
12 88 # N        ← 500
13 89 # sigma    ← 10
14 90 #
15 91 # pars ← c(R0 = 3.0,      # new infected people per infected person
16 92 #         r = 0.1,      # recovery rate
17 93 #         N = 500,      # population size
18 94 #         eta = 0.5,    # geometric random walk
19 95 #         berr = 0.5,   # Beta geometric walk noise
20 96 #         phi = 0.5)    # degree of connectivity

```

## 22 F.2 Spatial SIR HMC R Function Code

23 R code to simulate the outlined Spatial SIR function with HMC state reconstruction.  
24

```

25 1 ## Dexter Barrows
26 2 ## dbarrows.github.io
27 3 ## McMaster University
28 4 ## 2016
29 5
30 6 ## ymat: Contains the initial conditions where:
31 7 #         - rows are locations
32 8 #         - columns are S, I, R
33 9 ## pars: Contains the parameters: global values for R0, r, N, eta,
34 10 berr
35 11 ## T: The stop time. Since 0 is included, there should be T+1
36 12 time steps in the simulation
37 13 ## neinum: Number of neighbors for each location, in order
38 14 ## neibmat: Contains lists of neighbors for each location
39 15 #         - rows are parent locations (nodes)
40 16 #         - columns are locations each parent is attached to (edges)
41 17 StocSSIRstan ← function(ymat, pars, T, steps, neinum, neibmat,
42 18 berrmat, bmatlim) {
43 19
44 20     ## number of locations
45 21     nloc ← dim(ymat)[1]
46 22
47 23

```

```

1 20  ## storage
2 21  ## dims are locations, (S,I,R,B), times
3 22  # output array
4 23  out ← array(NA, c(nloc, 4, T+1), dimnames = list(NULL, c("S", "I",
5 24  "R", "B"), NULL))
6 24  # temp storage
7 25  BSI ← numeric(nloc)
8 26  rI ← numeric(nloc)
9 27
10 28  ## extract parameters
11 29  R0 ← pars[['R0']]
12 30  r ← pars[['r']]
13 31  N ← pars[['N']]
14 32  eta ← pars[['eta']]
15 33  berr ← pars[['berr']]
16 34  phi ← pars[['phi']]
17 35
18 36  B0 ← rep(R0*r/N, nloc)
19 37
20 38  ## state vectors
21 39  S ← ymat[, 'S']
22 40  I ← ymat[, 'I']
23 41  R ← ymat[, 'R']
24 42  B ← B0
25 43
26 44  ## assign starting to output matrix
27 45  out[, , 1] ← cbind(ymat, B0)
28 46
29 47  h ← 1 / steps
30 48
31 49  for ( i in 1:(T*steps) ) {
32 50
33 51    if ( i <= bmatlim ) {
34 52      B ← exp( log(B) + eta*(log(B0) - log(B)) + berrmat[,i])
35 53    } else {
36 54      B ← exp( log(B) + eta*(log(B0) - log(B)) + rnorm(nloc, 0,
37 55      berr) )
38 56    }
39 57
40 58    for ( loc in 1:nloc ) {
41 59      n ← neinum[loc]
42 60      sphI ← 1 - phi*(n/(n+1))
43 61      ophi ← phi/(n+1)
44 62      nBIsu ← B[neibmat[loc, 1:n]] %*% I[neibmat[loc, 1:n]]
45 63      BSI[loc] ← S[loc]*( sphI*B[loc]*I[loc] + ophi*nBIsu )
46 64    }
47 65
48 66    rI ← r*I
49 67
50 68    dS ← -BSI

```



```

1 69      dI ← BSI - rI
2 70      dR ← rI
3 71
4 72      S ← S + h*dS
5 73      I ← I + h*dI
6 74      R ← R + h*dR
7 75
8 76      if (i %% steps == 0)
9 77          out[,i/steps+1] ← cbind(S,I,R,B)
10 78
11 79    }
12 80
13 81    return(out)
14 82
15 83  }
16 84
17 85  ### Suggested parameters
18 86  #
19 87  # T          ← 60
20 88  # i_infec ← 5
21 89  # steps    ← 7
22 90  # N          ← 500
23 91  # sigma    ← 10
24 92  #
25 93  # pars ← c(R0 = 3.0,      # new infected people per infected person
26 94  #          r = 0.1,      # recovery rate
27 95  #          N = 500,      # population size
28 96  #          eta = 0.5,    # geometric random walk
29 97  #          berr = 0.5,   # Beta geometric walk noise
30 98  #          phi = 0.5 )   # interconnectivity degree
31

```

### 32 F.3 RStan Spatial SIR Code

33 This code implements a Spatial SIR model in Rstan.

```

34
35 1  ## Dexter Barrows
36 2  ## dbarrows.github.io
37 3  ## McMaster University
38 4  ## 2016
39 5
40 6  data {
41 7
42 8      int      <lower=1> T;      // total integration steps
43 9      int      <lower=1> nloc;   // number of locations
44 10     real      y[nloc, T];     // observed number of cases
45 11     int      <lower=1> N;      // population size
46 12     real      h;              // step size

```

```

1 13      int      <lower=0>   neinum[nloc];           // number of neighbors
2      each location has
3 14      int      neibmat[nloc, nloc]; // neighbor list for
4      each location
5 15
6 16 }
7 17
8 18 parameters {
9 19
10 20      real <lower=0, upper=10>      R0;           // R0
11 21      real <lower=0, upper=10>      r;           // recovery rate
12 22      real <lower=0, upper=20>      sigma;       // observation error
13 23      real <lower=0, upper=30>      Iinit[nloc]; // initial
14      infected for each location
15 24      real <lower=0, upper=1>      eta;          // geometric walk
16      attraction strength
17 25      real <lower=0, upper=1>      berr;         // beta walk noise
18 26      real <lower=-1.5, upper=1.5> Bnoise[nloc,T]; // Beta vector
19 27      real <lower=0, upper=1>      phi;          // interconnectivity
20      strength
21 28
22 29 }
23 30
24 31 model {
25 32
26 33      real S[nloc, T];
27 34      real I[nloc, T];
28 35      real R[nloc, T];
29 36      real B[nloc, T];
30 37      real B0;
31 38
32 39      real BSI[nloc, T];
33 40      real rI[nloc, T];
34 41      int n;
35 42      real sphi;
36 43      real ophi;
37 44      real nBisum;
38 45
39 46      B0 ← R0 * r / N;
40 47
41 48      for (loc in 1:nloc) {
42 49          S[loc, 1] ← N - Iinit[loc];
43 50          I[loc, 1] ← Iinit[loc];
44 51          R[loc, 1] ← 0.0;
45 52          B[loc, 1] ← B0;
46 53      }
47 54
48 55      for (t in 2:T) {
49 56          for (loc in 1:nloc) {
50 57
51 58              Bnoise[loc, t] ~ normal(0,berr);

```

```

1 59      B[loc, t] ← exp( log(B[loc, t-1]) + eta * ( log(B0) - log
2      (B[loc, t-1]) ) + Bnoise[loc, t] );
3 60
4 61      n ← neinum[loc];
5 62      sphi ← 1.0 - phi*( n/(n+1.0) );
6 63      ophi ← phi/(n+1.0);
7 64
8 65      nBIsun ← 0.0;
9 66      for (j in 1:n)
10 67          nBIsun ← nBIsun + B[neibmat[loc, j], t-1] * I[neibmat
11          [loc, j], t-1];
12 68
13 69      BSI[loc, t] ← S[loc, t-1]*( sphi*B[loc, t-1]*I[loc, t-1]
14      + ophi*nBIsun );
15 70      rI[loc, t] ← r*I[loc, t-1];
16 71
17 72      S[loc, t] ← S[loc, t-1] + h*( - BSI[loc, t] );
18 73      I[loc, t] ← I[loc, t-1] + h*( BSI[loc, t] - rI[loc, t] );
19 74      R[loc, t] ← R[loc, t-1] + h*( rI[loc, t] );
20 75
21 76      if (y[loc, t] > 0) {
22 77          y[loc, t] ~ normal( I[loc, t], sigma );
23 78      }
24 79
25 80      }
26 81  }
27 82
28 83      R0      ~ lognormal(1,1);
29 84      r        ~ lognormal(1,1);
30 85      sigma    ~ lognormal(1,1);
31 86      for (loc in 1:nloc) {
32 87          Iinit[loc] ~ normal(y[loc, 1], sigma);
33 88      }
34 89
35 90 }

```

## 37 F.4 IF2 Spatial SIR Code

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

```

39
40 1 /* Dexter Barrows
41 2   dbarrows.github.io
42 3   McMaster University
43 4   2016
44 5
45 6   */
46 7
47 8 #include <stdio.h>

```

```

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

```

```

1      int passnum, double coolrate);
2  55 double randu();
3  56 double randn();
4  57
5  58 // [[Rcpp::export]]
6  59 Rcpp::List if2_spa(NumericMatrix data, int T, int N, int NP, int
7      nPasses, double coolrate, NumericVector neinum, NumericMatrix
8      neibmat, int nloc) {
9  60
10 61     NumericMatrix paramdata(NP, 6);      // for R0, r, sigma, eta,
11      berr, phi
12 62     NumericMatrix initInfec(nloc, NP); // for Iinit
13 63     NumericMatrix infecmeans(nloc, T); // mean infection counts for
14      each location
15 64     NumericMatrix finalstate(nloc, 4); // SIRB means for each
16      location
17 65
18 66     srand(time(NULL));      // Seed PRNG with system time
19 67
20 68     double w[NP];          // particle weights
21 69
22 70     // initialize particles
23 71     printf("Initializing particle states\n");
24 72     Particle * particles = NULL;      // particle estimates for
25      current step
26 73     Particle * particles_old = NULL;  // intermediate particle
27      states for resampling
28 74     initializeParticles(&particles, NP, nloc, N);
29 75     initializeParticles(&particles_old, NP, nloc, N);
30 76
31 77     // START PASSES THROUGH DATA
32 78
33 79     printf("Starting filter\n");
34 80     printf("-----\n");
35 81     printf("Pass\n");
36 82
37 83
38 84     for (int pass = 0; pass < nPasses; pass++) {
39 85
40 86         printf("...%d / %d\n", pass, nPasses);
41 87
42 88         // reset particle system evolution states
43 89         for (int n = 0; n < NP; n++) {
44 90             for (int loc = 0; loc < nloc; loc++) {
45 91                 particles[n].S[loc] = N - particles[n].Iinit[loc];
46 92                 particles[n].I[loc] = particles[n].Iinit[loc];
47 93                 particles[n].R[loc] = 0.0;
48 94                 particles[n].B[loc] = (double) particles[n].R0 *
49      particles[n].r / N;
50 95             }
51 96         }

```

```

1  97
2  98     if (pass == (nPasses-1)) {
3  99         double means[nloc];
4 100         for (int loc = 0; loc < nloc; loc++) {
5 101             means[loc] = 0.0;
6 102             for (int n = 0; n < NP; n++) {
7 103                 means[loc] += particles[n].I[loc] / NP;
8 104             }
9 105             infecmeans(loc, 0) = means[loc];
10 106         }
11 107     }
12 108
13 109     for (int t = 1; t < T; t++) {
14 110
15 111         // generate individual predictions and weight
16 112         for (int n = 0; n < NP; n++) {
17 113
18 114             exp_euler_SSIR(1.0/7.0, 0.0, 1.0, N, &particles[n],
19             neinum, neibmat, nloc);
20 115
21 116             double merr_par = particles[n].sigma;
22 117
23 118             w[n] = 1.0;
24 119             for (int loc = 0; loc < nloc; loc++) {
25 120                 double y_diff = data(loc, t) - particles[n].I[
26                 loc];
27 121                 w[n] *= 1.0/(merr_par*sqrt(2.0*M_PI)) * exp( -
28                 y_diff*y_diff / (2.0*merr_par*merr_par) );
29 122             }
30 123         }
31 124
32 125         // cumulative sum
33 126         for (int n = 1; n < NP; n++) {
34 127             w[n] += w[n-1];
35 128         }
36 129
37 130         // save particle states to resample from
38 131         for (int n = 0; n < NP; n++){
39 132             copyParticle(&particles_old[n], &particles[n], nloc);
40 133         }
41 134
42 135         // resampling
43 136         for (int n = 0; n < NP; n++) {
44 137
45 138             double w_r = randu() * w[NP-1];
46 139             int i = 0;
47 140             while (w_r > w[i]) {
48 141                 i++;
49 142             }
50 143         }
51 144

```

```

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

```

```

1 194         Imeans[loc] = 0.0;
2 195         Rmeans[loc] = 0.0;
3 196         Bmeans[loc] = 0.0;
4 197         for (int n = 0; n < NP; n++) {
5 198             Smeans[loc] += particles[n].S[loc] / NP;
6 199             Imeans[loc] += particles[n].I[loc] / NP;
7 200             Rmeans[loc] += particles[n].R[loc] / NP;
8 201             Bmeans[loc] += particles[n].B[loc] / NP;
9 202         }
10 203         finalstate(loc, 0) = Smeans[loc];
11 204         finalstate(loc, 1) = Imeans[loc];
12 205         finalstate(loc, 2) = Rmeans[loc];
13 206         finalstate(loc, 3) = Bmeans[loc];
14 207     }
15 208
16 209
17 210     return Rcpp::List::create( Rcpp::Named("paramdata") = paramdata,
18 211                               Rcpp::Named("initInfec") = initInfec,
19 212                               Rcpp::Named("infecmeans") =
20                               infecmeans,
21 213                               Rcpp::Named("finalstate") =
22                               finalstate);
23 214
24 215
25 216
26 217 }
27 218
28 219
29 220 /* Use the Explicit Euler integration scheme to integrate SIR model
30 forward in time
31 221 double h      - time step size
32 222 double t0     - start time
33 223 double tn     - stop time
34 224 double * y    - current system state; a three-component vector
35                  representing [S I R], susceptible-infected-recovered
36 225
37 226 */
38 227 void exp_euler_SSIR(double h, double t0, double tn, int N, Particle *
39 particle,
40 228                     NumericVector neinum, NumericMatrix neibmat, int
41                     nloc) {
42 229
43 230     int num_steps = floor( (tn-t0) / h );
44 231
45 232     double * S = particle->S;
46 233     double * I = particle->I;
47 234     double * R = particle->R;
48 235     double * B = particle->B;
49 236
50 237     // create last state vectors
51 238     double S_last[nloc];

```



```

1 239 double I_last[nloc];
2 240 double R_last[nloc];
3 241 double B_last[nloc];
4 242
5 243 double R0 = particle->R0;
6 244 double r = particle->r;
7 245 double B0 = R0 * r / N;
8 246 double eta = particle->eta;
9 247 double berr = particle->berr;
10 248 double phi = particle->phi;
11 249
12 250 for(int t = 0; t < num_steps; t++) {
13 251
14 252     for (int loc = 0; loc < nloc; loc++) {
15 253         S_last[loc] = S[loc];
16 254         I_last[loc] = I[loc];
17 255         R_last[loc] = R[loc];
18 256         B_last[loc] = B[loc];
19 257     }
20 258
21 259     for (int loc = 0; loc < nloc; loc++) {
22 260
23 261         B[loc] = exp( log(B_last[loc]) + eta*(log(B0) - log(
24 262             B_last[loc])) + berr*randn() );
25 263
26 264         int n = neinum[loc];
27 265         double sphi = 1.0 - phi*( (double) n/(n+1.0) );
28 266         double ophi = phi/(n+1.0);
29 267
30 268         double nBIsum = 0.0;
31 269         for (int j = 0; j < n; j++)
32 270             nBIsum += B_last[(int) neibmat(loc, j) - 1] * I_last
33 271                 [(int) neibmat(loc, j) - 1];
34 272
35 273         double BSI = S_last[loc]*( sphi*B_last[loc]*I_last[loc] +
36 274             ophi*nBIsum );
37 275         double rI = r*I_last[loc];
38 276
39 277         // get derivatives
40 278         double dS = - BSI;
41 279         double dI = BSI - rI;
42 280         double dR = rI;
43 281
44 282         // step forward by h
45 283         S[loc] += h*dS;
46 284         I[loc] += h*dI;
47 285         R[loc] += h*dR;
48 286     }
49 287 }
50 288
51 289 }

```

```

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

```

```

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

```

```

1 384
2 385     do {
3 386         sigmacan = particles[n].sigma + spreadsigma*randn();
4 387     } while (sigmacan < 0);
5 388     particles[n].sigma = sigmacan;
6 389
7 390     do {
8 391         etacan = particles[n].eta + PSC*spreadeta*randn();
9 392     } while (etacan < 0 || etacan > 1);
10 393     particles[n].eta = etacan;
11 394
12 395     do {
13 396         berrcan = particles[n].berr + PSC*spreadberr*randn();
14 397     } while (berrcan < 0);
15 398     particles[n].berr = berrcan;
16 399
17 400     do {
18 401         phican = particles[n].phi + PSC*spreadphi*randn();
19 402     } while (phican <= 0 || phican >= 1);
20 403     particles[n].phi = phican;
21 404
22 405     for (int loc = 0; loc < nloc; loc++) {
23 406         do {
24 407             Iinitcan = particles[n].Iinit[loc] + spreadIinit*
25             randn();
26 408         } while (Iinitcan < 0 || Iinitcan > 500);
27 409         particles[n].Iinit[loc] = Iinitcan;
28 410     }
29 411 }
30 412
31 413 }
32 414
33 415 /* Convenience function for particle resampling process
34 416 */
35 417 void copyParticle(Particle * dst, Particle * src, int nloc) {
36 418
37 419     dst->R0      = src->R0;
38 420     dst->r        = src->r;
39 421     dst->sigma    = src->sigma;
40 422     dst->eta      = src->eta;
41 423     dst->berr     = src->berr;
42 424     dst->phi      = src->phi;
43 425
44 426     for (int n = 0; n < nloc; n++) {
45 427         dst->S[n]      = src->S[n];
46 428         dst->I[n]      = src->I[n];
47 429         dst->R[n]      = src->R[n];
48 430         dst->B[n]      = src->B[n];
49 431         dst->Iinit[n]  = src->Iinit[n];
50 432     }
51 433

```

```

1 434 }
2 435
3 436
4 437 double randu() {
5 438
6 439     return (double) rand() / (double) RAND_MAX;
7 440
8 441 }
9 442
10 443
11 444 /* Return a normally distributed random number with mean 0 and
12 445    standard deviation 1
13 446    Uses the polar form of the Box-Muller transformation
14 447    From http://www.design.caltech.edu/erik/Misc/Gaussian.html
15 448    */
16 449 double randn() {
17 450
18 451     double x1, x2, w, y1;
19 452
20 453     do {
21 454         x1 = 2.0 * randu() - 1.0;
22 455         x2 = 2.0 * randu() - 1.0;
23 456         w = x1 * x1 + x2 * x2;
24 457     } while ( w >= 1.0 );
25 458
26 459     w = sqrt( (-2.0 * log( w ) ) / w );
27 460     y1 = x1 * w;
28 461
29 462     return y1;
30 463 }
31 464
32 465 }

```

## 33 F.5 CUDA IF2 Spatial Fitting Code

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

```

37
38 1 /* Dexter Barrows
39 2    dbarrows.github.io
40 3    McMaster University
41 4    2016
42 5
43 6    */
44 7
45 8 #include <cuda.h>
46 9 #include <iostream>
47 10 #include <fstream>

```

```

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

```

```

1 57 __host__ std::string getHRmemsize (size_t memsize);
2 58 __host__ std::string getHRtime (float runtime);
3 59
4 60 __device__ void exp_euler_SSIR(float h, float t0, float tn, Particle
5    * particle, int * neinum, int * neibmat, int nloc);
6 61 __device__ void copyParticle(Particle * dst, Particle * src, int nloc
7    );
8 62
9 63
10 64 /* Initialize all PRNG states, get starting state vector using
11    initial distribution
12    */
13 66 __global__ void initializeParticles (Particle * particles, int nloc)
14    {
15 67
16 68     int id = blockIdx.x*blockDim.x + threadIdx.x; // global thread
17        ID
18 69
19 70     if (id < NP) {
20 71
21 72         // initialize PRNG state
22 73         curandState state;
23 74         curand_init(id, 0, 0, &state);
24 75
25 76         float R0can, rcan, sigmacan, linitcan, etacan, berrcan,
26            phican;
27 77
28 78         do {
29 79             R0can = R0true + R0true*curand_normal(&state);
30 80         } while (R0can < 0);
31 81         particles[id].R0 = R0can;
32 82
33 83         do {
34 84             rcan = rtrue + rtrue*curand_normal(&state);
35 85         } while (rcan < 0);
36 86         particles[id].r = rcan;
37 87
38 88         for (int loc = 0; loc < nloc; loc++)
39 89             particles[id].B[loc] = (float) R0can * rcan / N;
40 90
41 91         do {
42 92             sigmacan = merr + merr*curand_normal(&state);
43 93         } while (sigmacan < 0);
44 94         particles[id].sigma = sigmacan;
45 95
46 96         do {
47 97             etacan = etatrue + PSC*etatrue*curand_normal(&state);
48 98         } while (etacan < 0 || etacan > 1);
49 99         particles[id].eta = etacan;
50 100
51 101         do {

```

```

1 102         berrcan = berrtrue + PSC*berrtrue*curand_normal(&state);
2 103     } while (berrcan < 0);
3 104     particles[id].berr = berrcan;
4 105
5 106     do {
6 107         phican = phitrue + PSC*phitrue*curand_normal(&state);
7 108     } while (phican <= 0 || phican >= 1);
8 109     particles[id].phi = phican;
9 110
10 111     for (int loc = 0; loc < nloc; loc++) {
11 112         do {
12 113             Iinitcan = I0 + I0*curand_normal(&state);
13 114         } while (Iinitcan < 0 || N < Iinitcan);
14 115         particles[id].Iinit[loc] = Iinitcan;
15 116     }
16 117
17 118     particles[id].randState = state;
18 119
19 120 }
20 121
21 122 }
22 123
23 124 __global__ void resetStates (Particle * particles, int nloc) {
24 125
25 126     int id  = blockIdx.x*blockDim.x + threadIdx.x;  // global thread
26 127     ID
27 128
28 129     if (id < NP) {
29 130
30 131         for (int loc = 0; loc < nloc; loc++) {
31 132             particles[id].S[loc] = N - particles[id].Iinit[loc];
32 133             particles[id].I[loc] = particles[id].Iinit[loc];
33 134             particles[id].R[loc] = 0.0;
34 135         }
35 136
36 137     }
37 138 }
38 139
39 140 __global__ void clobberParams (Particle * particles, int nloc) {
40 141
41 142     int id  = blockIdx.x*blockDim.x + threadIdx.x;  // global thread
42 143     ID
43 144
44 145     if (id < NP) {
45 146
46 147         particles[id].R0 = R0true;
47 148         particles[id].r = rtrue;
48 149         particles[id].sigma = merr;
49 150         particles[id].eta = etatrue;
50 151         particles[id].berr = berrtrue;
51 152

```



```

1 151         particles[id].phi = phitru;
2 152
3 153         for (int loc = 0; loc < nloc; loc++) {
4 154             particles[id].Iinit[loc] = I0;
5 155         }
6 156
7 157     }
8 158
9 159 }
10 160
11 161
12 162 /* Project particles forward, perturb, and save weight based on data
13 163    int t - time step number (1,...,T)
14 164    */
15 165 __global__ void project (Particle * particles, int * neinum, int *
16 166    neibmat, int nloc) {
17 166
18 167     int id = blockIdx.x*blockDim.x + threadIdx.x;    // global id
19 168
20 169     if (id < NP) {
21 170         // project forward
22 171         exp_euler_SSIR(1.0/7.0, 0.0, 1.0, &particles[id], neinum,
23 172         neibmat, nloc);
24 172     }
25 173
26 174 }
27 175
28 176 __global__ void weight(float * data, Particle * particles, double * w
29 177    , int t, int T, int nloc) {
30 177
31 178     int id = blockIdx.x*blockDim.x + threadIdx.x;    // global id
32 179
33 180     if (id < NP) {
34 181
35 182         float merr_par = particles[id].sigma;
36 183
37 184         // Get weight and save
38 185         double w_local = 1.0;
39 186         for (int loc = 0; loc < nloc; loc++) {
40 187             float y_diff = data[loc*T + t] - particles[id].I[loc];
41 188             w_local *= 1.0/(merr_par*sqrt(2.0*PI)) * exp( - y_diff*
42 189             y_diff / (2.0*merr_par*merr_par) );
43 189         }
44 190
45 191         w[id] = w_local;
46 192
47 193     }
48 194
49 195 }
50 196
51 197 __global__ void stashParticles (Particle * particles, Particle *

```

```

1      particles_old, int nloc) {
2 198
3 199     int id = blockIdx.x*blockDim.x + threadIdx.x;    // global id
4 200
5 201     if (id < NP) {
6 202         // COPY PARTICLE
7 203         copyParticle(&particles_old[id], &particles[id], nloc);
8 204     }
9 205
10 206 }
11 207
12 208
13 209 /* The 0th thread will perform cumulative sum on the weights.
14 210    There may be a faster way to do this, will investigate.
15 211    */
16 212 __global__ void cumsumWeights (double * w) {
17 213
18 214     int id  = blockIdx.x*blockDim.x + threadIdx.x;    // global thread
19 215     ID
20 216
21 217     // compute cumulative weights
22 218     if (id == 0) {
23 219         for (int i = 1; i < NP; i++)
24 220             w[i] += w[i-1];
25 221     }
26 222 }
27 223
28 224
29 225 /* Resample from all particle states within cell
30 226    */
31 227 __global__ void resample (Particle * particles, Particle *
32 228     particles_old, double * w, int nloc) {
33 229
34 230     int id  = blockIdx.x*blockDim.x + threadIdx.x;
35 231
36 232     if (id < NP) {
37 233
38 234         // resampling proportional to weights
39 235         double w_r = curand_uniform(&particles[id].randState) * w[NP
40 236             -1];
41 237         int i = 0;
42 238         while (w_r > w[i]) {
43 239             i++;
44 240         }
45 241
46 242         // i is now the index of the particle to copy from
47 243         copyParticle(&particles[id], &particles_old[i], nloc);
48 244     }
49 245 }
50 246
51 247

```

```

1 245 }
2 246
3 247 // launch this with probably just nloc threads... block structure/
4 248 // size probably not important
5 248 __global__ void reduceStates (Particle * particles, float *
6 249 countmeans, int t, int T, int nloc) {
7 249
8 250     int id = blockIdx.x*blockDim.x + threadIdx.x;
9 251
10 252     if (id < nloc) {
11 253
12 254         int loc = id;
13 255
14 256         double countmean_local = 0.0;
15 257         for (int n = 0; n < NP; n++) {
16 258             countmean_local += particles[n].I[loc] / NP;
17 259         }
18 260
19 261         countmeans[loc*T + t] = (float) countmean_local;
20 262
21 263     }
22 264
23 265 }
24 266
25 267 __global__ void perturbParticles(Particle * particles, int nloc, int
26 268 passnum, double coolrate) {
27 268
28 269     //double coolcoef = exp( - (double) passnum / coolrate );
29 270     double coolcoef = pow(coolrate, passnum);
30 271
31 272     double spreadR0 = coolcoef * R0true / 10.0;
32 273     double spreadr = coolcoef * rtrue / 10.0;
33 274     double spreadsigma = coolcoef * merr / 10.0;
34 275     double spreadIinit = coolcoef * I0 / 10.0;
35 276     double spreadeta = coolcoef * etatrue / 10.0;
36 277     double spreadberr = coolcoef * berrtrue / 10.0;
37 278     double spreadphi = coolcoef * phitrue / 10.0;
38 279
39 280     double R0can, rcan, sigmacan, Iinitcan, etacan, berrcan, phican;
40 281
41 282     int id = blockIdx.x*blockDim.x + threadIdx.x;
42 283
43 284     if (id < NP) {
44 285
45 286         do {
46 287             R0can = particles[id].R0 + spreadR0*curand_normal(&
47 288                 particles[id].randState);
48 289         } while (R0can < 0);
49 290         particles[id].R0 = R0can;
50 291
51 292         do {

```

```

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

```

```

1 337
2 338     if (argc < 4) {
3 339         std::cout << "Not enough arguments" << std::endl;
4 340         return 0;
5 341     }
6 342
7 343     std::string arg1(argv[1]); // infection counts
8 344     std::string arg2(argv[2]); // neighbour counts
9 345     std::string arg3(argv[3]); // neighbour indices
10 346     std::string arg4(argv[4]); // outfile: params + runtime
11 347
12 348     std::cout << "Arguments:" << std::endl;
13 349     std::cout << "Infection data: " << arg1 << std::endl;
14 350     std::cout << "Neighbour counts: " << arg2 << std::endl;
15 351     std::cout << "Neighbour indices: " << arg3 << std::endl;
16 352     std::cout << "Outfile " << arg4 << std::endl;
17 353
18 354     // *****
19 355
20 356
21 357     // Read count data *****
22 358
23 359     std::cout << "Getting count data" << std::endl;
24 360     float * data = getDataFloat(arg1, &T, &nloc);
25 361     size_t datasize = nloc*T*sizeof(float);
26 362
27 363     // *****
28 364
29 365     // Read neinum matrix data *****
30 366
31 367     std::cout << "Getting neighbour count data" << std::endl;
32 368     int * neinum = getDataInt(arg2, NULL, NULL);
33 369     size_t neinumsize = nloc * sizeof(int);
34 370
35 371     // *****
36 372
37 373     // Read neibmat matrix data *****
38 374
39 375     std::cout << "Getting neighbour count data" << std::endl;
40 376     int * neibmat = getDataInt(arg3, NULL, NULL);
41 377     size_t neibmatsize = nloc * nloc * sizeof(int);
42 378
43 379     // *****
44 380
45 381     //
46
47
48 382
49 383     // start timing
50 384     gettimeofday (&tdr0, NULL);
51 385

```

```

1 386 // CUDA data *****
2 387
3 388 std::cout << "Allocating device storage" << std::endl;
4 389
5 390 float      * d_data;           // device copy of data
6 391 Particle   * particles;        // particles
7 392 Particle   * particles_old;    // intermediate particle states
8 393 double     * w;                // weights
9 394 int        * d_neinum;        // device copy of adjacency
10
11 395 int        * d_neibmat;        // device copy of neighbour
12 counts matrix
13 396 float      * countmeans;      // host copy of reduced infection
14 count means from last pass
15 397 float      * d_countmeans;    // device copy of reduced
16 infection count means from last pass
17 398
18 399 CUDA_CALL( cudaMalloc( (void**) &d_data           , datasize )
19 );
20 400 CUDA_CALL( cudaMalloc( (void**) &particles        , NP*sizeof(
21 Particle)) );
22 401 CUDA_CALL( cudaMalloc( (void**) &particles_old    , NP*sizeof(
23 Particle)) );
24 402 CUDA_CALL( cudaMalloc( (void**) &w                , NP*sizeof(
25 double)) );
26 403 CUDA_CALL( cudaMalloc( (void**) &d_neinum         , neinumsize)
27 );
28 404 CUDA_CALL( cudaMalloc( (void**) &d_neibmat        , neibmatsize)
29 );
30 405 CUDA_CALL( cudaMalloc( (void**) &d_countmeans     , nloc*T*sizeof(
31 float)) );
32 406
33 407
34 408 gettimeofday (&tdr1, NULL);
35 409 timeval_subtract (&restime, &tdr1, &tdr0);
36 410
37 411 std::cout << "\t" << getHRtime(restime) << std::endl;
38 412
39 413 size_t avail, total;
40 414 cudaMemGetInfo( &avail, &total );
41 415 size_t used = total - avail;
42 416
43 417 std::cout << "\t[" << getHRmemsize(used) << "]" used of [" <<
44 getHRmemsize(total) << "]" <<std::endl;
45 418
46 419 std::cout << "Copying data to device" << std::endl;
47 420
48 421 gettimeofday (&tdr0, NULL);
49 422
50 423 CUDA_CALL( cudaMemcpy(d_data      , data      , datasize      ,
51 cudaMemcpyHostToDevice) );

```

```

1 424   CUDA_CALL( cudaMemcpy(d_neinum , neinum , neinumsize ,
2      cudaMemcpyHostToDevice) );
3 425   CUDA_CALL( cudaMemcpy(d_neibmat , neibmat , neibmatsize ,
4      cudaMemcpyHostToDevice) );
5 426
6 427   gettimeofday (&tdr1, NULL);
7 428   timeval_subtract (&restime, &tdr1, &tdr0);
8 429
9 430   std::cout << "\t" << getHRtime(restime) << std::endl;
10 431
11 432   // *****
12 433
13 434
14 435
15 436   // Initialize particles *****
16 437
17 438   std::cout << "Initializing particles" << std::endl;
18 439
19 440   //gettimeofday (&tdr0, NULL);
20 441
21 442   int nThreads    = 32;
22 443   int nBlocks     = ceil( (float) NP / nThreads);
23 444
24 445   initializeParticles <<< nBlocks, nThreads >>> (particles, nloc);
25 446   CUDA_CALL( cudaGetLastError() );
26 447   CUDA_CALL( cudaDeviceSynchronize() );
27 448
28 449   initializeParticles <<< nBlocks, nThreads >>> (particles_old,
29      nloc);
30 450   CUDA_CALL( cudaGetLastError() );
31 451   CUDA_CALL( cudaDeviceSynchronize() );
32 452
33 453   //gettimeofday (&tdr1, NULL);
34 454   //timeval_subtract (&restime, &tdr1, &tdr0);
35 455   //std::cout << "\t" << getHRtime(restime) << std::endl;
36 456
37 457   cudaMemGetInfo( &avail, &total );
38 458   used = total - avail;
39 459   std::cout << "\t[" << getHRmemsize(used) << "]" used of [" <<
40      getHRmemsize(total) << "]" <<std::endl;
41 460
42 461   // *****
43 462
44 463   // Starting filtering *****
45 464
46 465   for (int pass = 0; pass < 50; pass++) {
47 466
48 467       nThreads    = 32;
49 468       nBlocks     = ceil( (float) NP / nThreads);
50 469
51 470       resetStates <<< nBlocks, nThreads >>> (particles, nloc);

```

```

1 471     CUDA_CALL( cudaGetLastError() );
2 472     CUDA_CALL( cudaDeviceSynchronize() );
3 473
4 474     nThreads = 1;
5 475     nBlocks  = 10;
6 476
7 477     if (pass == 49) {
8 478         reduceStates <<< nBlocks, nThreads >>> (particles,
9         d_countmeans, 0, T, nloc);
10 479     CUDA_CALL( cudaGetLastError() );
11 480     CUDA_CALL( cudaDeviceSynchronize() );
12 481 }
13 482
14 483 int Tlim = T;
15 484
16 485 for (int t = 1; t < Tlim; t++) {
17 486
18 487     // Projection
19     *****
20 488
21 489     nThreads    = 32;
22 490     nBlocks     = ceil( (float) NP / nThreads);
23 491
24 492     project <<< nBlocks, nThreads >>> (particles, d_neinum,
25     d_neibmat, nloc);
26 493     CUDA_CALL( cudaGetLastError() );
27 494     CUDA_CALL( cudaDeviceSynchronize() );
28 495
29 496     // Weighting
30     *****
31 497
32 498     nThreads    = 32;
33 499     nBlocks     = ceil( (float) NP / nThreads);
34 500
35 501     weight <<< nBlocks, nThreads >>>(d_data, particles, w, t,
36     T, nloc);
37 502     CUDA_CALL( cudaGetLastError() );
38 503     CUDA_CALL( cudaDeviceSynchronize() );
39 504
40 505     // Cumulative sum
41     *****
42 506
43 507     nThreads    = 1;
44 508     nBlocks     = 1;
45 509
46 510     cumsumWeights <<< nBlocks, nThreads >>> (w);
47 511     CUDA_CALL( cudaGetLastError() );
48 512     CUDA_CALL( cudaDeviceSynchronize() );
49 513
50 514     // Save particles for resampling from
51     *****

```



```

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

```

```

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

```

```

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

```

```

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

```

```

1 702     std::string valstring;
2 703
3 704     int kb = 1024;
4 705     int mb = kb*1024;
5 706     int gb = mb*1024;
6 707
7 708     if (memsize <= kb)
8 709         ss << memsize << " B";
9 710     else if (memsize > kb && memsize <= mb)
10 711         ss << (float) memsize/ kb << " KB";
11 712     else if (memsize > mb && memsize <= gb)
12 713         ss << (float) memsize/ mb << " MB";
13 714     else
14 715         ss << (float) memsize/ gb << " GB";
15 716
16 717     valstring = ss.str();
17 718
18 719     return valstring;
19 720
20 721 }
21 722
22 723
23 724 /* Convert time in seconds to human readable format
24 725 */
25 726 std::string getHRtime (float runtime) {
26 727
27 728     std::stringstream ss;
28 729     std::string valstring;
29 730
30 731     int mt = 60;
31 732     int ht = mt*60;
32 733     int dt = ht*24;
33 734
34 735     if (runtime <= mt)
35 736         ss << runtime << " s";
36 737     else if (runtime > mt && runtime <= ht)
37 738         ss << runtime/mt << " m";
38 739     else if (runtime > ht && runtime <= dt)
39 740         ss << runtime/dt << " h";
40 741     else
41 742         ss << runtime/ht << " d";
42 743
43 744     valstring = ss.str();
44 745
45 746     return valstring;
46 747
47 748 }

```

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

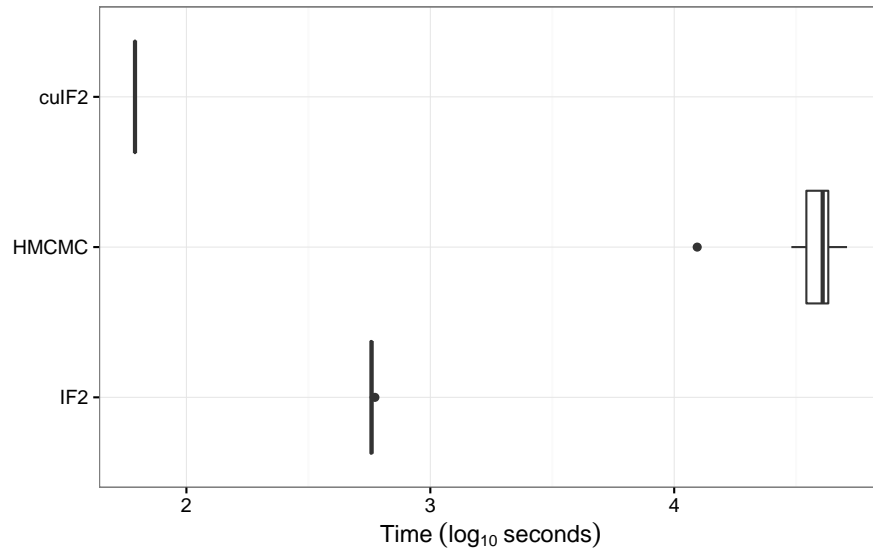


Figure F.1: Running times for fitting the spatial SIR model to data.

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