

ESTIMATION AND INFERENCE OF NONLINEAR
STOCHASTIC TIME SERIES

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

By
DEXTER BARROWS, B.SC.

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

MASTER OF SCIENCE (2016) (Mathematics)

McMaster University
Hamilton, Ontario, Canada

TITLE	A Comparative Study of Techniques for Estimation and Inference of Nonlinear Stochastic Time Series
AUTHOR	Dexter Barrows, B.Sc. (Honours), Ryerson University
SUPERVISOR	Dr. Benjamin Bolker
NUMBER OF PAGES	x, 145

Abstract

Forecasting tools play an important role in public response to in 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 IF2, Hamiltonian MCMC, and S-mapping. These were compared in several forecasting scenarios, including a seasonal epidemic and a spatiotemporal epidemic. IF2 combined with parametric bootstrapping produced superior predictions in all scenarios. S-mapping combined with Dewdrop Regression produced forecasts slightly less-accurate than IF2 and HMCMC, but demonstrated vastly reduced running times. Hence, S-mapping with or without Dewdrop Regression should be used to glean initial insight into future epidemic behaviour, while IF2 and parametric bootstrapping should be used to refine forecast estimates in time.

Acknowledgements

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

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

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

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

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

Contents

1	Introduction	1
2	Hamiltonian MCMC	5
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	9
2.8	Hamiltonian Monte Carlo	10
2.9	RStan Fitting	13
3	Iterated Filtering	18
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
4	Parameter Fitting	26
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 MC Fitting	35
4.7	HMC MC Densities	35
4.8	HMC MC and Bootstrapping	36
4.9	Multi-trajectory Parameter Estimation	37

5 Forecasting Frameworks	39
5.1 Data Setup	39
5.2 IF2	40
5.2.1 Parametric Bootstrapping	41
5.2.2 IF2 Forecasts	41
5.3 HMCMC	43
5.4 Truncation vs. Error	44
6 S-map and SIRS	46
6.1 S-maps	46
6.2 S-map Algorithm	47
6.3 SIRS Model	49
6.4 SIRS Model Forecasting	51
7 Spatial Epidemics	54
7.1 Spatial SIR	54
7.2 Dewdrop Regression	56
7.3 Spatial Model Forecasting	57
8 Discussion and Future Directions	59
8.1 Parallel and Distributed Computing	59
8.2 IF2, Bootstrapping, and Forecasting Methodology	62
8.3 Fin	63
A Hamiltonian MCMC	69
A.1 Full R code	69
A.2 Full Stan code	72
B Iterated Filtering	74
B.1 Full R code	74
B.2 Full C++ code	76
C Parameter Fitting	87
D Forecasting Frameworks	88
D.1 IF2 Parametric Bootstrapping Function	88
D.2 RStan Forward Simulator	90
E S-map and SIRS	92
E.1 SIRS R Function Code	92
E.2 SMAP Code	93
E.3 SMAP Parameter Optimization Code	95
E.4 RStan SIRS Code	97
E.5 IF2 SIRS Code	98

F Spatial Epidemics	112
F.1 Spatial SIR R Function Code	112
F.2 RStan Spatial SIR Code	114
F.3 IF2 Spatial SIR Code	116
F.4 CUDA IF2 Spatial Fitting Code	127

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 warmup.	15
2.4	Traceplot of samples drawn for parameter R_0 , including warmup.	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 show in Equation [4.1].	27
4.2	Density plot of values shown if 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 $\epsilon_{\text{obvs}} = \mathcal{N}(0, 10)$, 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	As before, the solid grey lines show the true parameter values and the dashed grey lines show the density means.	35

4.9	As before, the solid grey lines show the true parameter values and the dashed grey lines show the density means.	36
4.10	Result from 100 HMCMC bootstrap trajectories. The solid line shows the true states, the dots show the data, the dotted line shows the average system behaviour, the dashed line shows the bootstrap mean, and the grey ribbon shows the centre 95th quantile of the bootstrap trajectories.	37
4.11	IF2 point estimate densities are shown in black and HMCMC point estimate densities are shown in grey. The vertical black lines show the true parameter values.	38
4.12	Fitting times for IF2 and HMCMC, in seconds. The centre box in each plot shows the centre 50th quantile, with the bold centre line showing the median.	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 HMCMC / bootstrapping framework with $M = 200$ trajectories. The dotted line shows the mean estimate of the forecasts, and the grey ribbon shows the 95% confidence interval on the estimated true states as described in Figure [5.3].	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 quantile, the bold line is the median, and the dots are outliers. Note that these are not “true” outliers, simply ones outside a ranges based on the interquartile range.	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 observa- tion 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 mid- dle 50th quantile, the bold line is the median, and the dots are outliers.	58
F.1	Running times for fitting the spatial SIR model to data.	145

¹ Chapter 1

² Introduction

maybe use `\usepackage{citesort}` ? or
equivalent

- 3 Epidemic forecasting is an important tool that can help inform public policy and
4 decision-making in the face of an infectious disease outbreak [27][9][39]. Successful
5 intervention relies on accurate predictions of the number of cases, when they will occur,
6 and where. Without this information it is difficult to efficiently allocate resources, a
7 critical step in curbing the size and breadth of an epidemic. what's the diff between size and breadth?
breadth= duration?
- 8 Despite the importance of reliable forecasts, obtaining them remains a challenge from
9 both a theoretical and practical standpoint [27]. Mathematical models can capture the
10 essential drivers in disease dynamics, and extend them past the present into the future.
11 However, different epidemics may present ~~with~~ varying dynamics and require different
12 model parameters to be accurately represented [7]. These parameters can be inferred
13 by using statistical model fitting techniques, but this can become computationally
14 intensive, and the modeller risks “overfitting” by attempting to capture too many
15 drivers with too little data. Thus, The The modeller must exercise restraint in model
16 selection and fitting technique [5].
- 17 Securing precise, error-free data in the midst of an outbreak can be difficult if not
18 impossible [33]; thus uncertainty in what we observe in building mathematical models
19 of disease spread must be accounted for from the get-go. Models must differentiate
20 between natural variation in the intensity of disease spread (process error) and error
21 in data collection (observation error) in order to accurately determine the dynamics
22 underlying a data set, adding another layer of complexity [22]. With these caveats
23 and concerns acknowledged, we can turn to a discussion of technique.
- 24 Broadly, there are three primary categories of techniques used in forecasting: phe-
25 nomenological, pure mechanistic, and semi-mechanistic.
- 26 Phenomenological methods operate purely on data, fitting models that do not try
27 to reconstruct disease dynamics, but rather focus purely on trend. A long-standing
28 and widely-used example is the Autoregressive Integrated Moving Average (ARIMA)

hmm. Technically I don't know if p,d,q are *parameters* -- I think of an ARIMA(p,d,q) as having p+q parameters (d only determines the order of differencing -- not associated with any free parameters. But I'm not sure I know what to call them instead ...

- 1 model. ARIMA assumes a linear underlying process and Gaussian error distributions.
- 2 It uses three parameters representing the degree of autoregression (p), integration
- 3 (trend removal) (d), and the moving average (q), where the orders of the autoregression
- 4 and the moving average are determined through the use of an autocorrelation function
- 5 (ACF) and partial autocorrelation function (PACF), respectively, applied to the the
- 6 data *a priori* [40].
- 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 compartment model in which individuals are divided into categories based
- 10 on whether they are susceptible to infection or infected but not yet themselves infec-
- 11 tious, infectious, or recovered. These models are referred to as susceptible-infectious-
- 12 removed (SIR) models and are heavily used in epidemiological study. Typically the
- 13 transition between compartments is governed by a set of ordinary differential equa-
- 14 tions, such as

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

SIR-based models also simply called "compartamental" studies?

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

- 25 Combining the phenomenological and mechanistic approaches are the semi-mechanistic
- 26 techniques. These methods use a model to define the expected underlying dynamics of
- 27 the system, but integrate data into the model in order to refine estimates of the model
- 28 parameters and produce more accurate forecasts. Typically the first step in imple-
- 29 menting such a technique is fitting the desired model to existing data. There are many
- 30 ways to do this, most of which fall into two main categories: particle filter-based (PF)
- 31 methods [3][31][39], and Markov chain Monte Carlo-based (MCMC) methods [2][26].
- 32 From there data can either be integrated into the model by refitting the model to
- 33 the new longer data set, or in an “on-line” fashion in which data points can be di-

this is not I think
what is generally
meant by semi-mech
SM refers to
models, not
estimation
techniques,
e.g. an SIR
model with
incidence =
 $b(S,I)*S*I$ where
 b is a tensor
spline, or e.g. 2
your model with
autoregressive beta

- maybe ref. that spatiotemporal AR model we read?
- 1 rectly integrated without the need to refit the entire model. Normally, MCMC-based
 2 machinery must refit the entire model whereas PF-based approaches can sometimes
 3 integrate data in an on-line fashion.
- 4 Another, broader, distinction among techniques can be drawn between those that rely
 5 on assumptions of linearity, and those that make no such assumption. As epidemic
 6 dynamics are highly non-linear, it can be questionable as to even consider linear ap-
 7 proaches to epidemic forecasting at all. In particular, stalwart approaches such as
 8 ARIMA and the venerable Kalman filter face a distinct (at least theoretical) disad-
 9 vantage in the face of newer PF-based methods [39][36]. Additionally, these methods
 10 are very-well-studied, and further work showing their viability would likely prove
 11 extraneous in the modern academic landscape. although ensemble KF, various extended KFs deal with nonlinearity ...
- 12 Somewhat frustratingly, there exists no “gold standard” in forecasting [27][39][9]. As
 13 methodology varies widely in theoretical justification, implementation, and operation,
 14 it is difficult to compare the state of the art in forecasting methods from a first-
 15 principles perspective. Further, published work making use of any of these methods
 16 to forecast uses different prediction accuracy metrics, such as SSE, peak time/dura-
 17 tion/intensity, correlation tests, or RMSE, among others [28][9]. Thus is is difficult
 18 to select the best tool for the job when faced with a forecasting problem.
- 19 The primary focus of this work is to compare best-in-class methods for forecasting
 20 in several epidemically-focused scenarios. These include the a “standard” one-shot
 21 forecast outbreak in which the outbreak subsides and does not recur, a seasonal out-
 22 break scenario such as the one we see with influenza each year, and a spatiotemporal
 23 scenario in which multiple spatial location¹ are connected and disease is free to spread
 24 from one to another. HMCNC has been around but NUTS, useful implementations with AD are relatively recent ...
- 25 From MCMC-based methods we have selected Hamiltonian MCMC [26], a (slightly)
 26 less cutting-edge but nonetheless highly effective technique. We are using HMCNC
 27 through an implementation in the R package RStan [8], which at its core uses HM-
 28 CMC, but also contains implementations of several other innovative techniques. In-
 29 terestingly, the original goal of this package was not to implement a statistical pro-
 30 gramming language similar to Just Another Gibbs Sampler (JAGS) or Bayesian In-
 31 ference Using Gibbs Sampling (BUGS), but with an HMCNC backend. In fact the
 32 developers’ original goal was to implement any method that could fit multilevel hi-
 33 erarchical models without halting as they were witnessing with BUGS and JAGS.
 34 Only after experimenting with several options and starting to hear about it more and
 35 more frequently did they attempt to work with HMCNC. In the end, the scope of the
 36 project grew to include the development and subsequent integration of the No-U-Turn
 37 Sampler (NUTS) [17], and an implementation of automatic differentiation machinery
 38 [34].
- 39 For PF-based methods we have selected IF2 [20], a very recently developed approach
 40 that uses multiple particle filtering rounds to generate Maximum Likelihood Estimates

maybe not cap?

1 (MLEs). It functions similarly to its predecessor, the Maximum likelihood via Iterated
2 Filtering (MIF) algorithm [19], but aims to be simpler, faster, and more accurate.
3 Theoretical justification and synthetic testing indicates that IF2 meets these goals,
4 and as such the authors recommend skipping MIF and jumping straight to IF2 if an
5 algorithm of their variety is sought. And so, we are doing just that. We wrote our
6 own IF2 implementation in C++ and integrated it into R using the Rcpp package
7 [12]. The developers of MIF and IF2 have their own R package that implements MIF
8 and IF2, Partially Observed Markov Processes (POMP) [20][23], but it didn't provide
9 some of the diagnostic information we needed, so it was not used here.

10 Finally, from the phenomenological methods we have selected the sequential locally
11 weighted global linear maps (S-map) [36][35][18][13], combined with Dewdrop Regres-
12 sion [13]. These methods stand on their own as a unique take on the forecasting
13 problem, and bear little resemblance to other methodology. The virtues of these tech-
14 niques have been long-extolled by their developers, but their efficacy when compared
15 to competing methods has not been well-studied. This work will mark one of the first
16 times this has been done.

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

skipping to chapter 8 now ...

¹ Chapter 2

² Hamiltonian MCMC

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

¹⁰ 2.1 Markov Chains

¹¹ Figure [2.1] shows a finite state machine with 3 states $S = \{x_1, x_2, x_3\}$.
¹² The transition probabilities can be summarized as a matrix as

$$\text{13} \quad T = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0.1 & 0.9 \\ 0.6 & 0.4 & 0 \end{bmatrix}. \quad (2.1)$$

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

$$\text{17} \quad \lim_{t \rightarrow \infty} \mu(x^{(1)})T^t = (27/122, 50/122, 45/122). \quad (2.2)$$

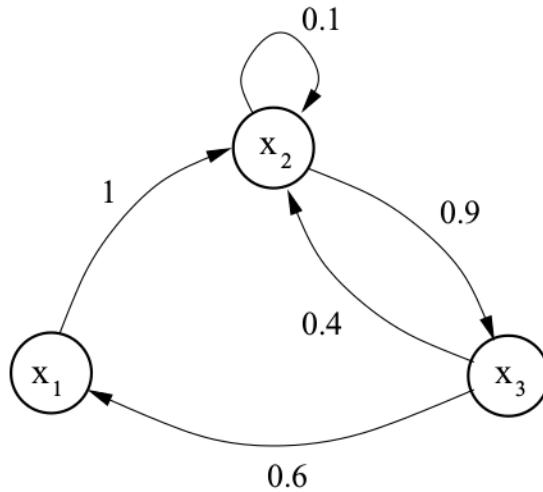


Figure 2.1: A finite state machine. States are shown as graph nodes, and the probability of transitioning from one particular state to another is shown as a weighted graph edge. [2]

- 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 probability
 - 5 • *Positive Recurrent* The number of steps required for the chain to reach state A from state B must be finite
 - 6 • *Aperiodic* The chain must be able to explore the parameter space without becoming 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 θ 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
$$\mathcal{L}(\theta) \propto P(D|\theta). \quad (2.3)$$

3 In some standard Maximum Likelihood approaches, $\mathcal{L}(\theta)$ is searched to find a value
4 of θ that maximizes $\mathcal{L}(\theta)$, then this θ is taken to be the most likely true value. Here
5 our aim is to not just maximize the likelihood but to also explore the area around it
6 [2].

7 2.3 Prior distribution

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

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

26 2.4 Proposal distribution

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

- 1 stand a decent chance of also being accepted, but far enough away that it doesn't get
 2 "trapped" in a particular region of the parameter space.
- 3 This is done through the use of a proposal or candidate distribution. This will usually
 4 be a distribution centred around our last accepted step and with a dispersion potential
 5 narrower than that of our prior distribution.
- 6 The choice of this distribution is theoretically not of the utmost importance, but in
 7 practice becomes important so as to not waste computer time [2].

8 2.5 Algorithm

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

11 We will denote the previously discussed quantities as

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

16 and define the acceptance ratio, r , as

$$17 \quad r = \frac{\mathcal{L}(\theta^*)p(\theta^*)q(\theta^*|\theta)}{\mathcal{L}(\theta)p(\theta)q(\theta|\theta^*)}, \quad (2.4)$$

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

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

$$22 \quad r = \frac{\mathcal{L}(\theta^*)p(\theta^*)}{\mathcal{L}(\theta)p(\theta)}. \quad (2.5)$$

23 Algorithm [1] shows the Metropolis MCMC algorithm.

24 In this way we are ensuring that steps that lead to better likelihood outcomes are
 25 likely to be accepted, but steps that do not will not be accepted as frequently. Note
 26 that these less "advantageous" moves will still occur but that this is by design – it
 27 ensures that as much of the parameter space as possible will be explored but more
 28 efficiently than using pure brute force [2].

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 2.6 Burn-in

- 2 One critical aspect of MCMC-based algorithms has yet to be discussed. The algorithm
 3 requires an initial starting point θ to be selected, but as the proposal distribution
 4 is supposed to restrict moves to an area close to the last accepted state, then the
 5 posterior distribution will be biased towards this starting point. This issue is avoided
 6 through the use of a Burn-in period.
- 7 Burning in a chain is the act of running the MCMC algorithm normally without saving
 8 first M samples. As we are seeking a chain of length N , the total computation will
 9 be equivalent to generating a chain of length $M + N$ [2].

10 2.7 Thinning

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

¹ 2.8 Hamiltonian Monte Carlo

² The Metropolis-Hastings algorithm has a primary drawback in that the parameter
³ space may not be explored efficiently in some circumstances – a consequence of the
⁴ rudimentary proposal mechanism. Instead, smarter moves can be proposed through
⁵ the use of Hamiltonian dynamics, leading to a better exploration of the target distri-
⁶ bution and a potential decrease in overall computational complexity. This algorithm
⁷ is coined Hamiltonian MCMC (HMC or HMCMC) [26]. Prior to the advent of HM-
⁸ CMC, some work was conducted exploring adaptive step-sizing using MCMC-based
⁹ methods, but lack strong theoretical justification, and can lead to some samples being
¹⁰ drawn from an incorrect distribution [26].

¹¹ From physics, we will borrow the ideas of potential and kinetic energy. Here potential
¹² energy is analogous to the negative log likelihood of the parameter selection given the
¹³ data, formally

$$\text{14} \quad U(\theta) = -\log(\mathcal{L}(\theta)p(\theta)). \quad (2.6)$$

¹⁵ Kinetic energy will serve as a way to “nudge” the parameters along a different moment
¹⁶ for each component of θ . We introduce n auxiliary variables $r = (r_1, r_1, \dots, r_n)$, where
¹⁷ n is the number of components in θ . Note that the samples drawn for r are not of
¹⁸ interest, they are only used to inform the evolution of the Hamiltonian dynamics of
¹⁹ the system. We can now define the kinetic energy as

$$\text{20} \quad K(r) = \frac{1}{2}r^T M^{-1}r, \quad (2.7)$$

²¹ where M is an $n \times n$ matrix. In practice M can simply be chosen as the identity matrix
²² of size n , however it can also be used to account for correlation between components
²³ of θ .

²⁴ The Hamiltonian of the system is defined as

$$\text{25} \quad H(\theta, r) = U(\theta) + K(r), \quad (2.8)$$

²⁶ where the Hamiltonian dynamics of the combined system can be simulated using the
²⁷ following system of ODEs:

$$\begin{aligned} \text{28} \quad \frac{d\theta}{dt} &= M^{-1}r \\ \frac{dr}{dt} &= -\nabla U(\theta). \end{aligned} \quad (2.9)$$

1 .

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

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

12
$$r = \exp [H(\theta, r) - H(\theta^*, r^*)], \quad (2.10)$$

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

14 Together, we have Algorithm 2.

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

Algorithm 2: Hamiltonian MCMC

```

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

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

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

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

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

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

    /* 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 2.9 RStan Fitting

² Here we will examine a test case in which Hamiltonian MCMC will be used to
³ fit a Susceptible-Infected-Removed (SIR) epidemic model to mock infectious count
⁴ data.

⁵ The synthetic data was produced by taking the solution to a basic SIR ODE model,
⁶ sampling it at regular intervals, and perturbing those values by adding in observation
⁷ noise. The SIR model used was outlined in the introduction in Equation [2.1].

⁸ The solution to this system was obtained using the `ode()` function from the `deSolve`
⁹ package. The required derivative array function in the format required by `ode()` was
¹⁰ specified as the gradient in Equation [2.1].

¹¹ The true parameter values were set to $R_0 = 3.0, r = 0.1, N = 500$. The initial
¹² conditions were set to 5 infectious individuals, 495 people susceptible to infection,
¹³ and no one had yet recovered from infection and been removed. The system was
¹⁴ integrated over $[0, 100]$ with infected counts drawn at each integer time step.

¹⁵ The observation error was taken to be $\varepsilon_{obs} \sim \mathcal{N}(0, \sigma)$, where individual values were
¹⁶ drawn for each synthetic data point.

¹⁷ Figure [2.2] shows the system simulation results.

¹⁸ The Hamiltonian MCMC model fitting was done using Stan (<http://mc-stan.org/>),
¹⁹ a program written in C++ that does Bayesian statistical inference using Hamiltonian
²⁰ MCMC. Stan's R interface (<http://mc-stan.org/interfaces/rstan.html>) was used
²¹ to ease implementation.

²² Throughout this paper, the explicit Euler integration scheme was used to obtain solu-
²³ tions to our ODE-based models. While this scheme is not the most accurate or
²⁴ efficient one available, it was chosen for its ease of implementation in the required lan-
²⁵ guages and transparency with regards to stochastic processes, which have been added
²⁶ into later models. Using a more advanced integrator such as Runge-Kutta makes it
²⁷ harder to properly specify how stochastic process evolution should be handled, and
²⁸ would have required significantly more implementation work to boot. Hence, we have
²⁹ opted for the lo-fi solution we know will function the way we require.

³⁰ In order to use an Explicit Euler-like stepping method in the later Stan model, the
³¹ synthetic observation counts were treated as weekly observations in which the counts
³² on the other six days of the week were unobserved.

³³ Figure [2.3] shows the traceplot for the post-warmup chain data returned by
³⁴ the `stan()` function in the `fit` object. We see that the chains are mixing well and
³⁵ convergence has likely been reached.

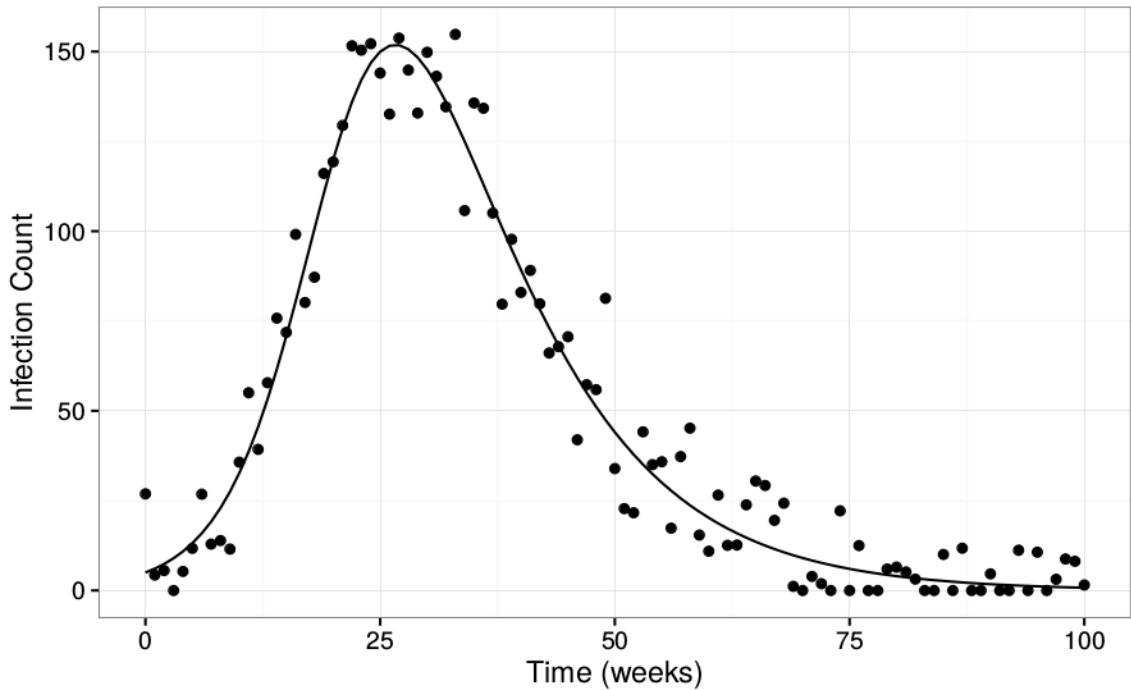


Figure 2.2: True SIR ODE solution infected counts, and with added observation noise.

- 1 Figure [2.4] shows the chain data including the warmup samples in. We can see why
- 2 is it wise to discard these samples (note the scale).
- 3 Figure [2.5] shows the kernel density estimates for each of the model parameters
- 4 and the initial number of cases. We see that while the estimates are not perfect, they
- 5 are more than satisfactory.

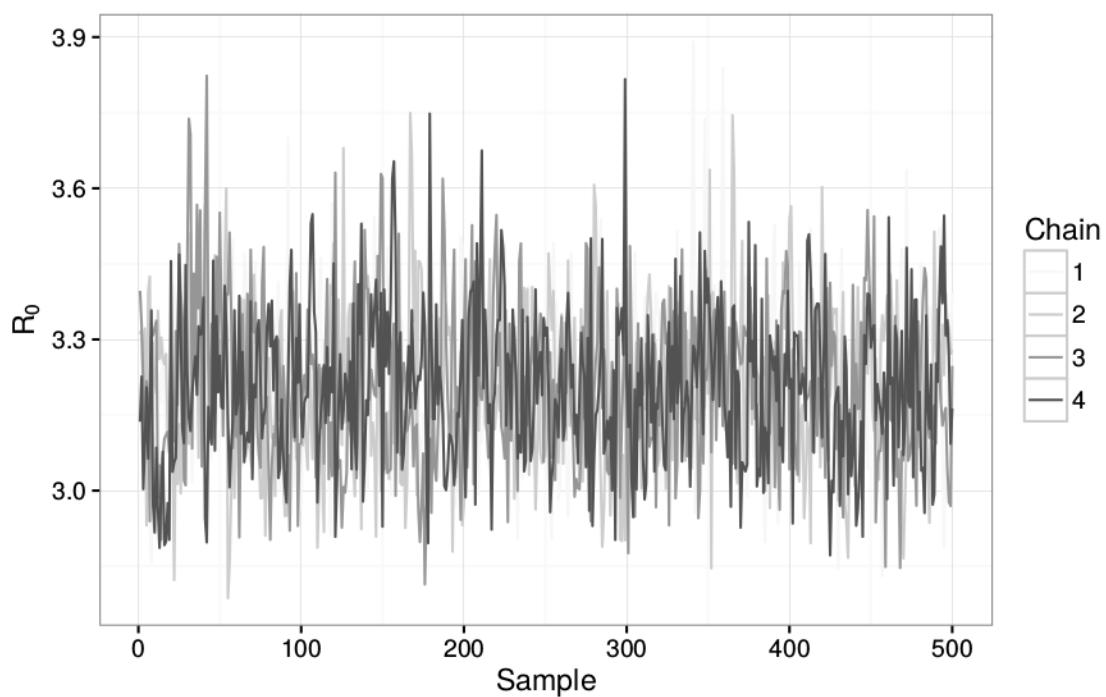


Figure 2.3: Traceplot of samples drawn for parameter R_0 , excluding warmup.

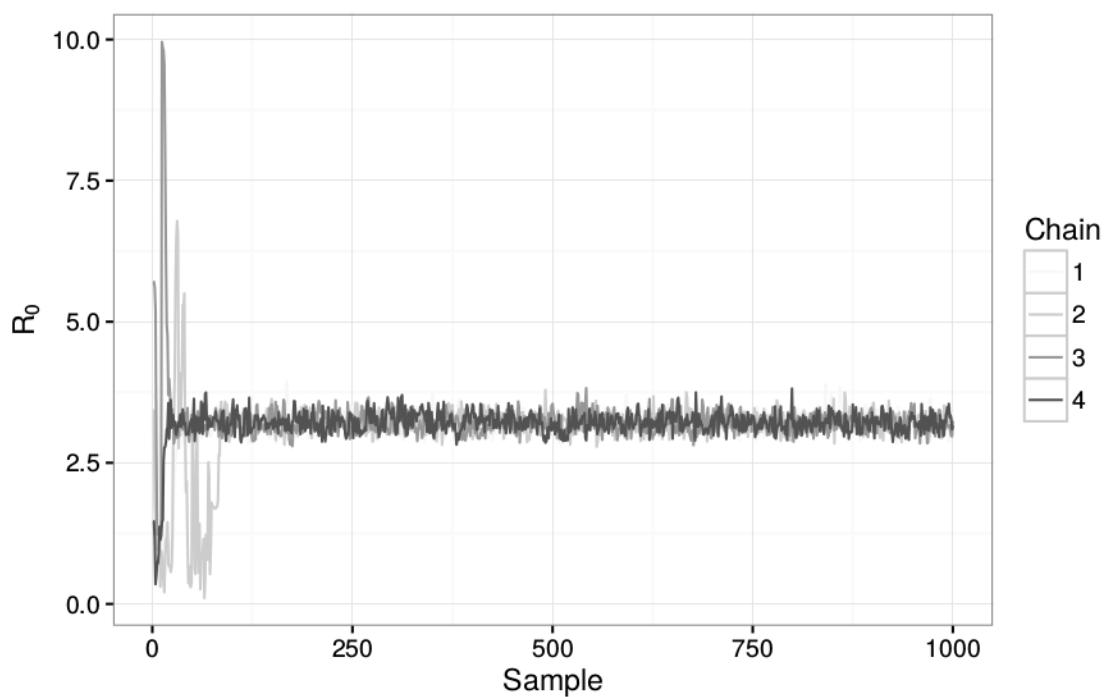


Figure 2.4: Traceplot of samples drawn for parameter R_0 , including warmup.

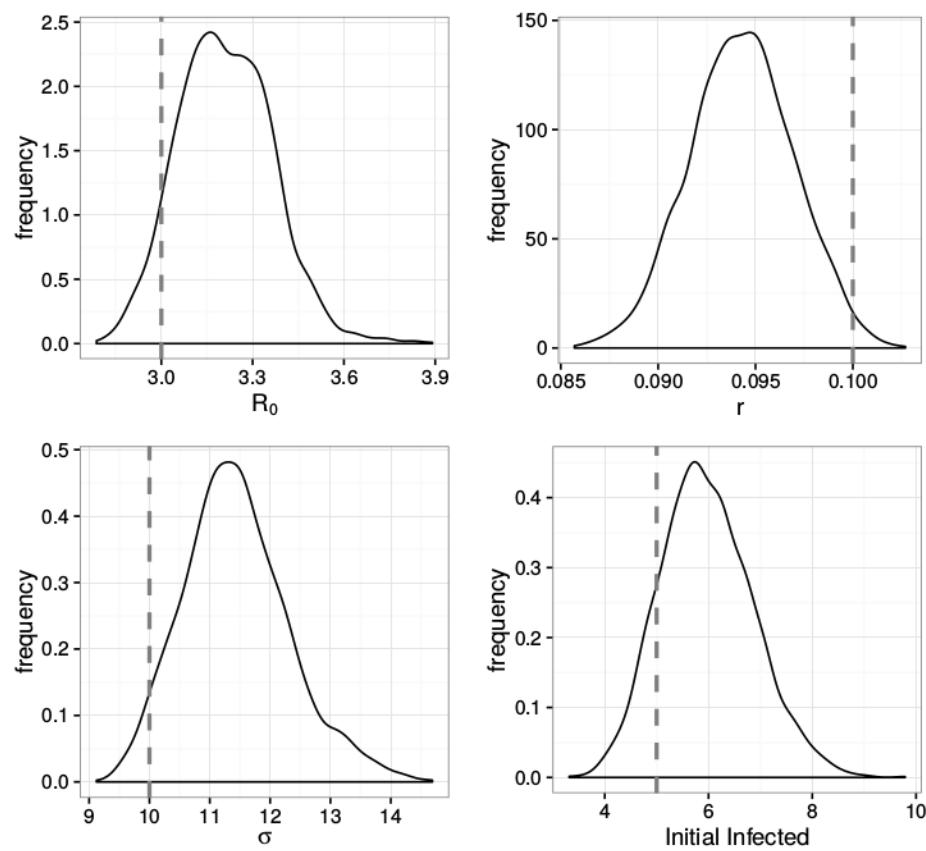


Figure 2.5: Kernel density estimates produced by Stan. Dashed lines show true parameter values.

¹ Chapter 3

² Iterated Filtering

³ Particle filters are similar to MCMC-based methods in that they use likelihoods to
⁴ evaluate the validity of proposed parameter sets given observed data D , but differ in
⁵ that they are largely trying to produce point estimates of the parameters instead of
⁶ samples from the posterior distribution.

⁷ Instead of constructing a Markov chain and approximating its stationary distribution,
⁸ a cohort of “particles” are used to move through the data in an on-line (sequential)
⁹ fashion with the cohort being culled of poorly-performing particles at each iteration
¹⁰ via importance sampling. If the culled particles are not replenished, this will be
¹¹ a Sequential Importance Sampling (SIS) particle filter. If the culled particles are
¹² replenished from surviving particles, in a sense setting up a process analogous to
¹³ Darwinian selection, then this will be a Sequential Importance Resampling (SIR)
¹⁴ particle filter [3].

¹⁵ 3.1 Formulation

¹⁶ Particle filters, also called Sequential Monte-Carlo (SMC) filters, feature similar core
¹⁷ functionality as the venerable Kalman Filter. As the algorithm moves through the
¹⁸ data (sequence of observations), a prediction-update cycle is used to simulate the
¹⁹ evolution of the model M with different particular parameter selections, track how
²⁰ closely these predictions approximate the new observed value, and update the current
²¹ cohort appropriately [3].

²² Two separate functions are used to simulate the evolution and observation processes.
²³ The “true” state evolution is specified by

$$\text{24} \quad X_{t+1} \sim f_1(X_t, \theta), \quad (3.1)$$

1 And the observation process by

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

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

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

13 3.2 Algorithm

14 Now we will formalize the particle filter.

15 We will denote each particle $p^{(j)}$ as the j^{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  $(\theta^{(1)}, \theta^{(2)}, \dots, \theta^{(J)})$ 

---



```

¹ 3.3 Particle Collapse

² Often, a situation may arise in which a single particle is assigned a normalized weight
³ very close to 1 and all the other particles are assigned weights very close to 0. When
⁴ this occurs, the next generation of the cohort will overwhelmingly consist of de-
⁵ scendants of the heavily-weighted particle, termed particle collapse or degeneracy
⁶ [6] [3].

⁷ Since the basic SIR particle filter does not perturb either the particle system states
⁸ or system parameter values, the cohort will quickly consist solely of identical par-
⁹ ticles, effectively halting further exploration of the parameter space as new data is
¹⁰ introduced.

¹¹ A similar situation occurs when a small number of particles (but not necessarily a
¹² single particle) split almost all of the normalized weight between them, then jointly
¹³ dominate the resampling process for the remainder of the iterations. This again halts
¹⁴ the exploration of the parameter space with new data.

¹⁵ In either case, the hallmark feature used to detect collapse is the same – at some point
¹⁶ the cohort will consist of particles with very similar or identical parameter sets which
¹⁷ will consequently result in their assigned weights being extremely close together.

¹⁸ Mathematically, we are interested in the number of effective particles, N_{eff} , which
¹⁹ represents the number of particles that are acceptably dissimilar. This is estimated
²⁰ by evaluating

$$\text{21} \quad N_{\text{eff}} = \frac{1}{\sum_1^J (w^{(j)})^2}. \quad (3.3)$$

²² This can be used to diagnose not only when collapse has occurred, but can also
²³ indicate when it is near [3].

²⁴ 3.4 Iterated Filtering and Data Cloning

²⁵ A particle filter hinges on the idea that as it progresses through the data set D ,
²⁶ its estimate of the posterior carried in the cohort of particles approaches maximum
²⁷ likelihood. However, this convergence may not be fast enough so that the estimate
²⁸ it produces is of quality before the data runs out. One way around this problem is
²⁹ to “clone” the data and make multiple passes through it as if it were a continuation
³⁰ of the original time series. Note that the system state contained in each particle will
³¹ have to be reset with each pass.

1 Rigorous proofs have been developed [19][20] that show that by treating the parameters as stochastic processes instead of fixed values, the multiple passes through the data will indeed force convergence of the process mean toward maximum likelihood, and the process variance toward 0.

5 3.5 Iterated Filtering 2 (IF2)

6 The successor to Iterated Filtering 1 [19], Iterated Filtering 2 [20] 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 $h(\theta|, \sigma)$. Typically this is multivariate Normal with σ being a vector of variances proportional to
22 the expected values of θ . In practice the proportionality can be derived from current
23 means or specified ahead of time. Further, these intensities must decrease over time.
24 This can be done via exponential or geometric cooling, a decreasing step function, a
25 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        $p^{(j)} \sim h(p^{(j)}, \sigma_m)$ 
      /* Evolve */
10       $X_t^{(j)} \leftarrow f_1(X_{t-1}^{(j)}, \theta^{(j)})$ 
      /* Weight */
11       $w^{(j)} \leftarrow P(y_t | X_t^{(j)}, \theta^{(j)}) = f_2(X_t^{(j)}, \theta^{(j)})$ 
      /* */

12     /* Normalize */
13     for  $j = 1:J$  do
14        $w^{(j)} \leftarrow w^{(j)} / \sum_1^J w^{(j)}$ 
      /* Resample */
15        $p^{(1:J)} \leftarrow \text{sample}(p^{(1:J)}, \text{prob} = w, \text{replace} = \text{true})$ 
      /* */

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

/* 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 HM-
²⁷ CMC, these distributions are not meant to consist of samples from the true posterior
²⁸ distribution, but rather serve a diagnostic role.

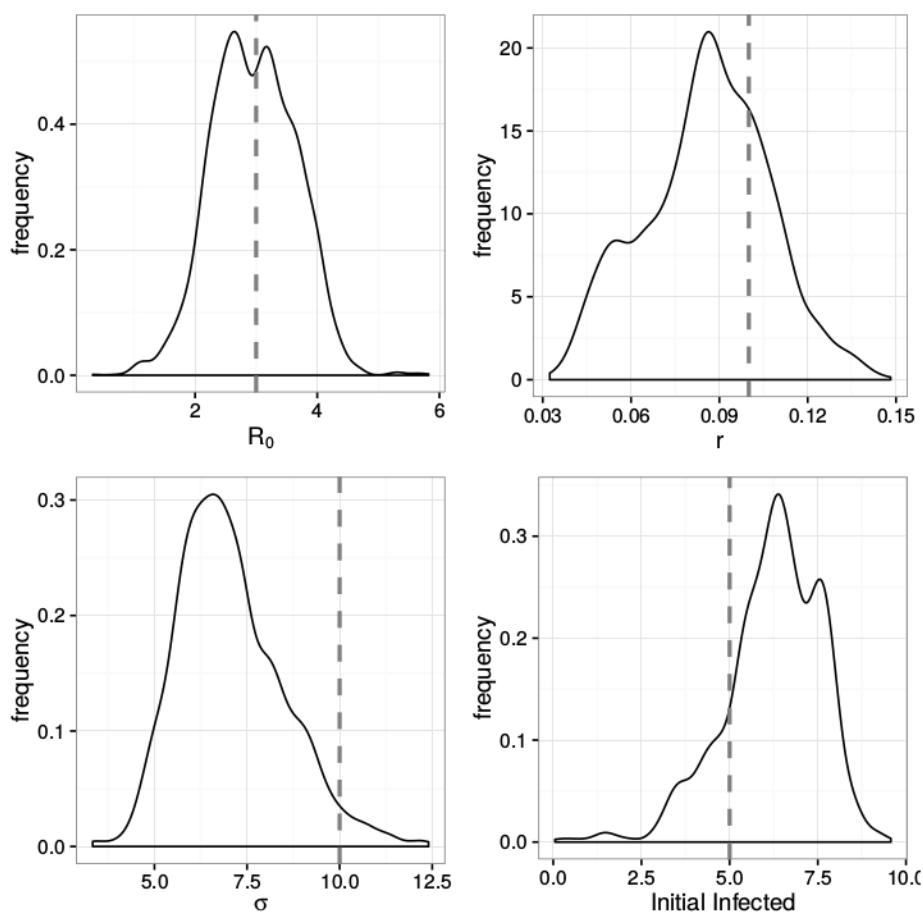


Figure 3.1: Kernel estimates for four essential system parameters.
True values are indicated by dashed lines.

¹ Chapter 4

² Parameter Fitting

³ 4.1 Fitting Setup

⁴ Now that we have established which methods we wish to evaluate the efficacy of for
⁵ epidemic forecasting, it is prudent to see how they perform when fitting parameters
⁶ for a known epidemic model. We have already seen how they perform when fitting
⁷ parameters for a model with a deterministic evolution process and observation noise,
⁸ but a more realistic model will have both process and observation noise.

⁹ To form such a model, we will take a deterministic SIR ODE model specified in
¹⁰ Equation [□] and add process noise by allowing β to follow a geometric random walk
¹¹ given by

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

¹² We will take ϵ_t to be normally distributed with standard deviation ρ^2 such that
¹³ $\epsilon_t \sim \mathcal{N}(0, \rho^2)$. The geometric attraction term constrains the random walk, the force
¹⁴ of which is $\eta \in [0, 1]$. If we take $\eta = 0$ then the walk will be unconstrained; if we
¹⁵ let $\eta = 1$ then all values of β_t will be independent from the previous value (and
¹⁶ consequently all other values in the sequence).

¹⁷ When $\eta \in (0, 1)$, we have an autoregressive process of order 1 on the logarithmic scale
¹⁸ of the form

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

¹⁹ where ϵ_t is normally distributed noise with mean 0 and standard deviation σ_E . This
²⁰ process has a theoretical expected mean of $\mu = c/(1-\rho)$ and variance $\sigma = \sigma_E^2/(1-\rho^2)$.

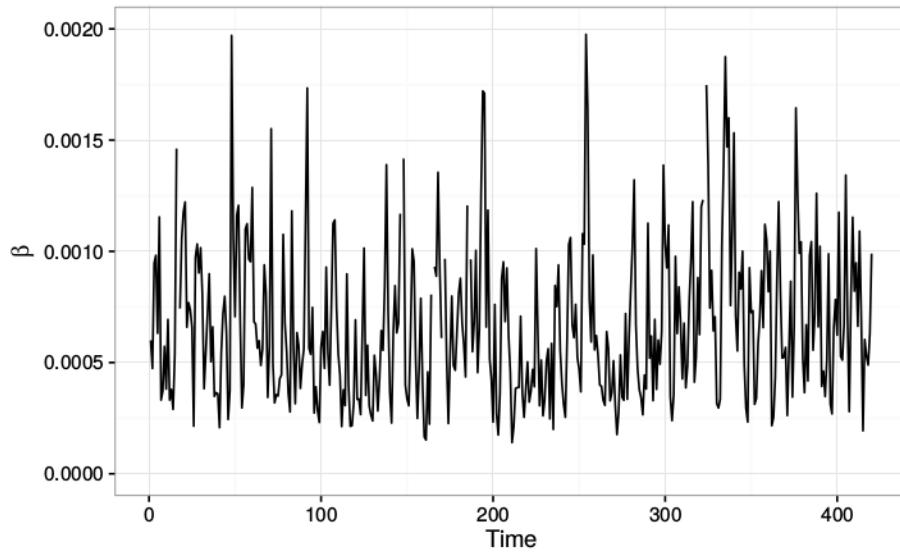


Figure 4.1: Simulated geometric autoregressive process show in Equation [4.1].

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

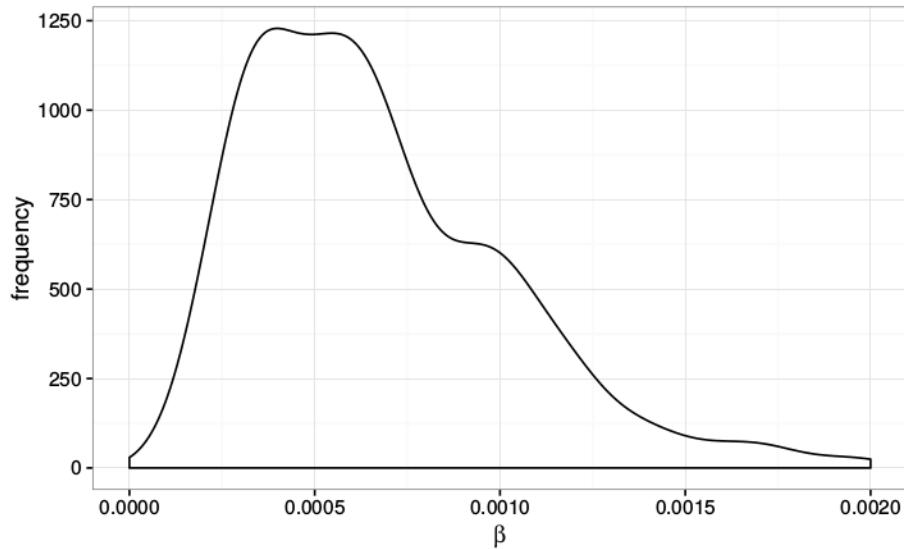


Figure 4.2: Density plot of values shown if Figure[4.1].

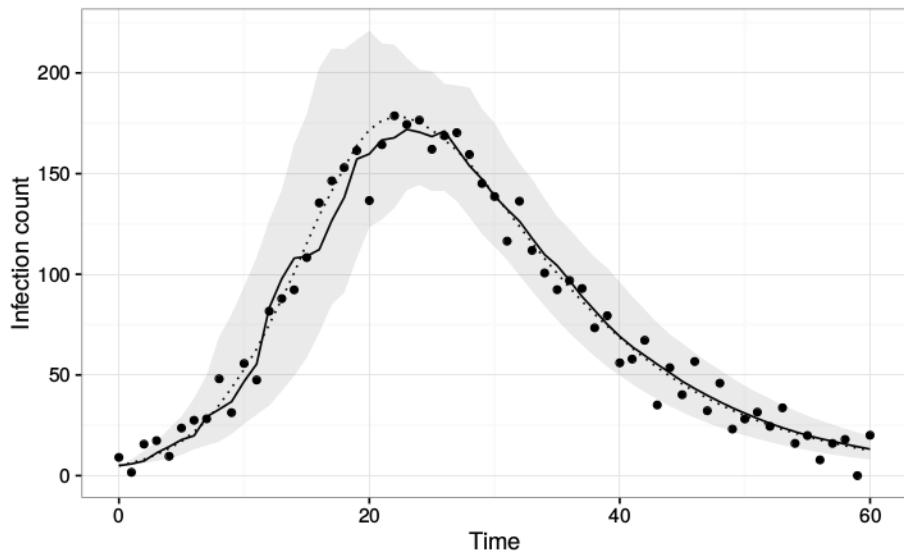


Figure 4.3: Stochastic SIR model simulated using an explicit Euler stepping scheme. The solid line is a single random trajectory, the dots show the data points obtained by adding in observation error defined as $\epsilon_{\text{obvs}} = \mathcal{N}(0, 10)$, and the grey ribbon is centre 95th quantile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

¹ 4.2 Calibrating Samples

² In order to compare HMCMD and IF2 we need to set up a fair and theoretically
³ justified way to select the number of samples to draw for the HMCMD iterations and
⁴ the number of particles to use for IF2. As we wish to compare, among other things,
⁵ approximate computational cost using runtimes, we need to determine how many
⁶ sample draws for each method are required to obtain a certain accuracy. Sample
⁷ draws are typically not comparable in terms of quality when considering multiple
⁸ methods. For example, vanilla MCMC draws are computationally cheap compared
⁹ to those from HMCMD, but HMCMD produces draws that more efficiently cover the
¹⁰ sampling space. Thus we cannot just set the number of HMCMD draws equal to the
¹¹ number of particles used in IF2 – we must calibrate both quantities based on a desired
¹² target error. We assume that we are working with a problem that has an unknown
¹³ real solution, so we use the Monte Carlo Standard Error (MCSE) [15].

¹⁴ Suppose we are using a Monte-Carlo based method to obtain a mean estimate $\hat{\mu}_n$ for
¹⁵ a quantity μ , 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 σ^2 is the variance of the samples drawn.

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

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

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

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

²⁶ where Θ^* 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

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

1 where P is the number of particles.

2 The goal here is to then pick the number of HMCMC 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 HMCMC 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 stochastic
13 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 [21][23],
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	IF2			HMCMC	
	True	Fit	Error	Fit	Error
R_0	3.0	3.27	9.08×10^{-2}	3.12	1.05×10^{-1}
r	10^{-1}	1.04×10^{-1}	3.61×10^{-2}	9.99×10^{-2}	-7.56×10^{-4}
Initial Infected	5	7.90	5.80×10^{-1}	6.64	3.28×10^{-1}
σ	10	8.84	-1.15×10^{-1}	8.5	-1.50×10^{-1}
η	5×10^{-1}	5.87×10^{-1}	1.73×10^{-1}	4.57×10^{-1}	-8.27×10^{-2}
ε_{err}	5×10^{-1}	1.63×10^{-1}	-6.73×10^{-1}	1.60×10^{-1}	-6.80×10^{-1}

Figure 4.4: Fitting errors.

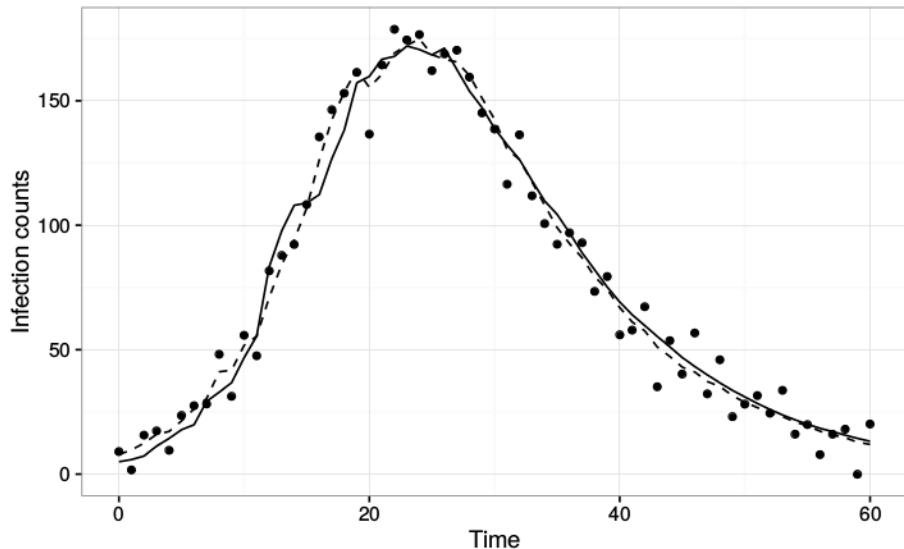


Figure 4.5: True system trajectory (solid line), observed data (dots), and IF2 estimated real state (dashed line).

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 parame-
¹⁰ ters.
- 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.

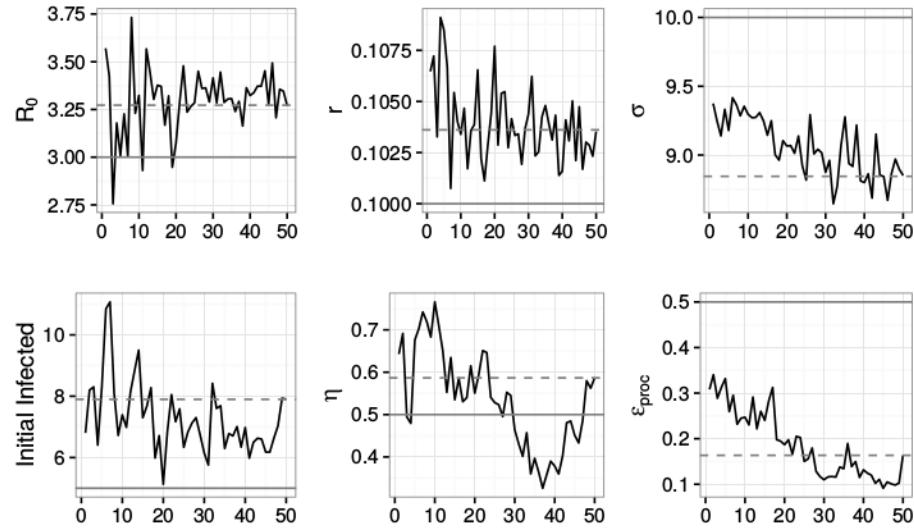


Figure 4.6: The horizontal axis shows the IF2 pass number. The solid black lines show the evolution of the ML estimates, the solid grey lines show the true value, and the dashed grey lines show the mean parameter estimates from the particle swarm after the final pass.

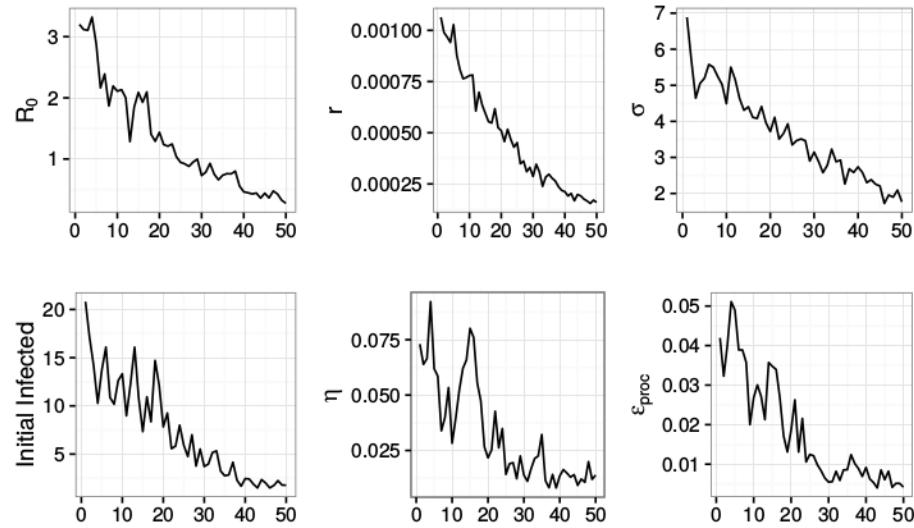


Figure 4.7: The horizontal axis shows the IF2 pass number and the solid black lines show the evolution of the standard deviations of the particle swarm values.

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.

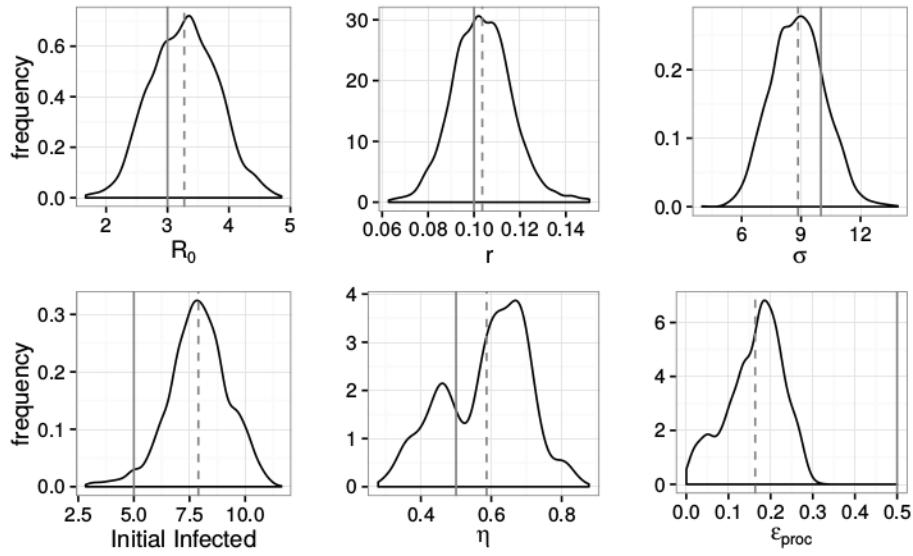


Figure 4.8: As before, the solid grey lines show the true parameter values and the dashed grey lines show the density means.

¹ 4.6 HMC MCMC Fitting

- ² We can use the Hamiltonian Monte Carlo algorithm implemented in the ‘Rstan’ pack-
- ³ age to fit the stochastic SIR model as above. This was done with a single HMC chain
- ⁴ of 2000 iterations with 1000 of those being warm-up iterations.
- ⁵ The MLE parameter estimates, taken to be the means of the samples in the chain, were
- ⁶ shown in the table in Figure [4.4] along with the true values and relative error.

⁷ 4.7 HMC MCMC Densities

- ⁸ Figure [4.9] shows the parameter estimation densities from the Stan HMC MCMC fit-
- ⁹ ting.
- ¹⁰ the densities shown here represent a “true” MLE density estimate in that they rep-
- ¹¹ resent HMC’s attempt to directly sample from the parameter space according to the
- ¹² likelihood surface, unlike IF2 which is in theory only trying to get a ML point esti-
- ¹³ mate. Hence, these densities are potentially more robust than those produced by the
- ¹⁴ IF2 implementation.

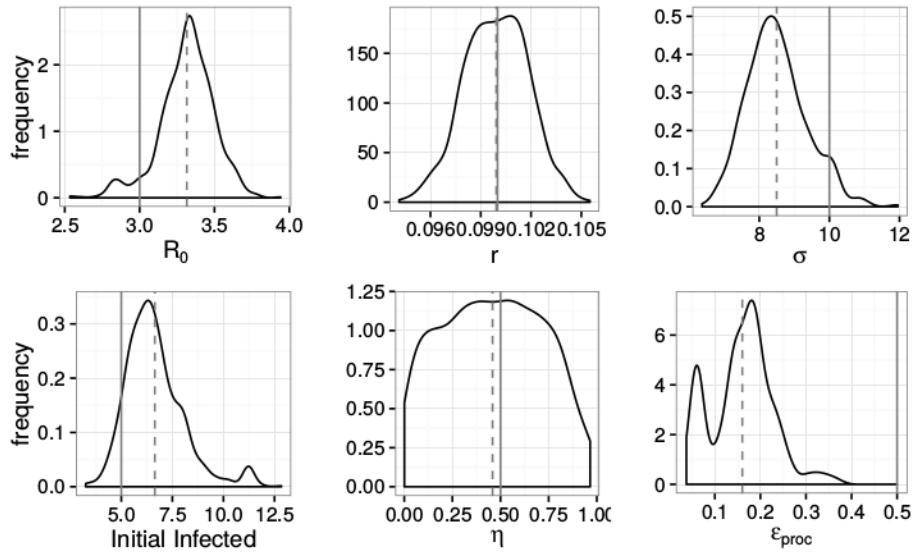


Figure 4.9: As before, the solid grey lines show the true parameter values and the dashed grey lines show the density means.

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.

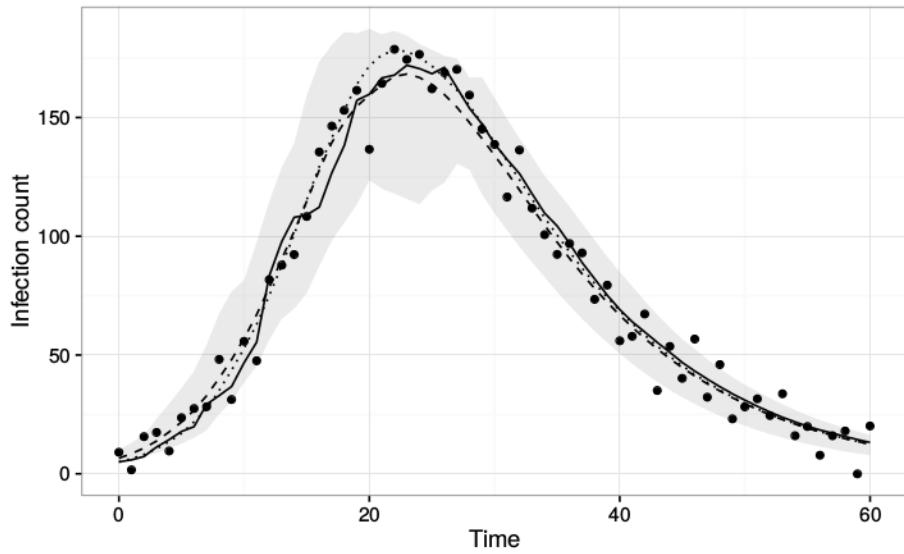


Figure 4.10: Result from 100 HMCMB bootstrap trajectories. The solid line shows the true states, the dots show the data, the dotted line shows the average system behaviour, the dashed line shows the bootstrap mean, and the grey ribbon shows the centre 95th quantile of the bootstrap trajectories.

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 HMCMB for
- 5 200 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 ε_{proc} showing slightly wider coverage than the HMCMB densities for the same parameters.
- 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 HMCMB respectively, representing a 5.7x speedup for IF2 over HMCMB. While
- 12 IF2 may be able to fit the model to data faster than HMCMB, we are obtaining less
- 13 information; this will become important in the next section. Further, the results in
- 14 Figure [4.12] show that while the running time for IF2 is relatively fixed, the times
- 15 for HMCMB are anything but, showing a wide spread of potential times.

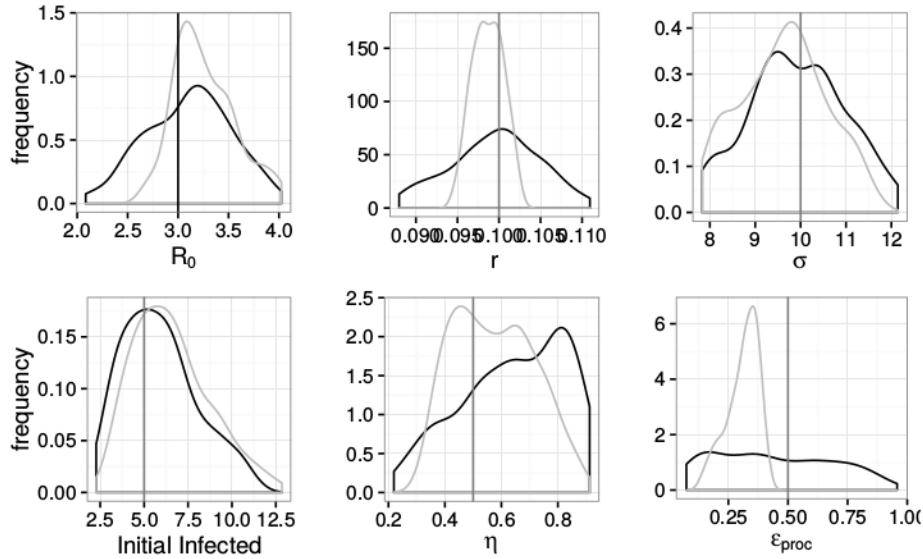


Figure 4.11: IF2 point estimate densities are shown in black and HMCMD point estimate densities are shown in grey. The vertical black lines show the true parameter values.

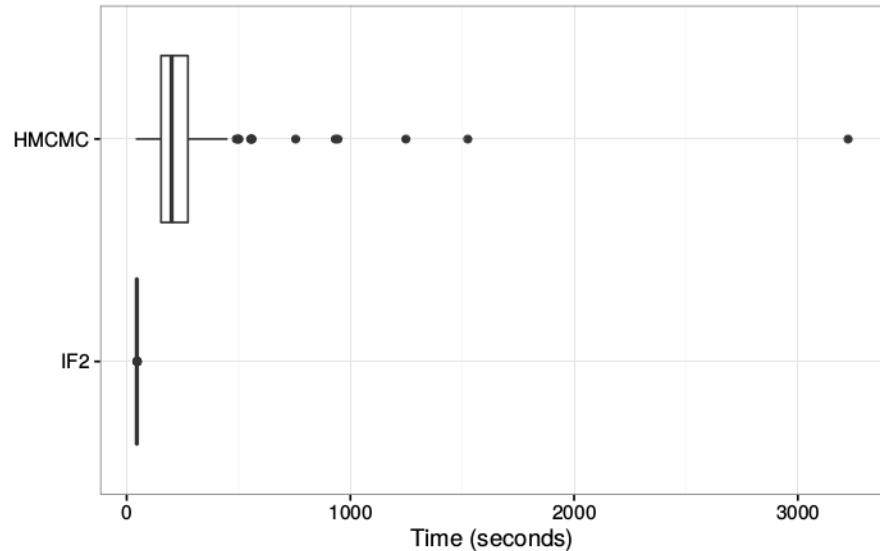


Figure 4.12: Fitting times for IF2 and HMCMD, in seconds. The centre box in each plot shows the centre 50th quantile, with the bold centre line showing the median.

¹ Chapter 5

² Forecasting Frameworks

³ 5.1 Data Setup

- ⁴ This section will focus on taking the stochastic SIR model from the previous section,
- ⁵ truncating the synthetic data output from realizations of that model, and seeing how
- ⁶ well IF2 and HMCMC can reconstruct out-of-sample forecasts.
- ⁷ Figure [5.1] shows an example of a simulated system with truncated data.
- ⁸ In essence, we want to be able to give either IF2 or HMCMC only the data points
- ⁹ and have it reconstruct the entirety of the true system states.

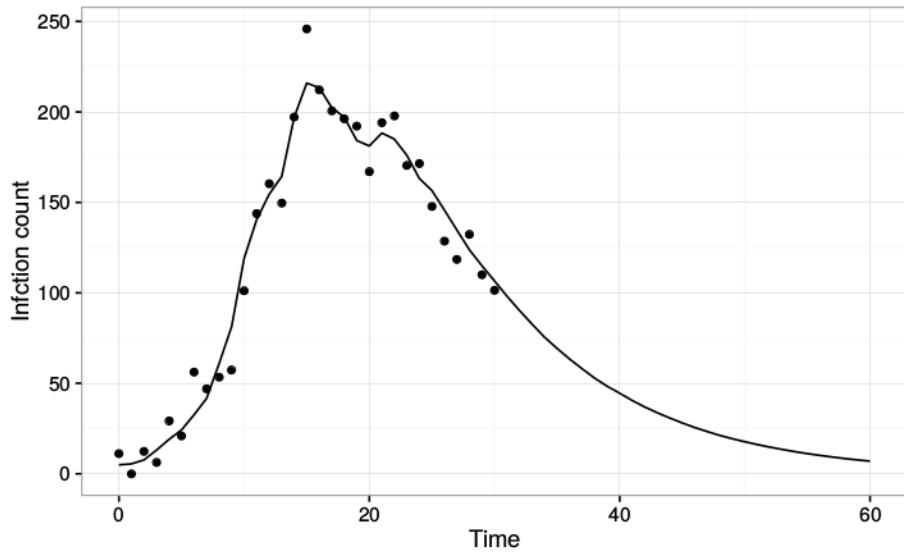


Figure 5.1: Infection count data truncated at $T = 30$. The solid line shows the true underlying system states, and the dots show those states with added observation noise. Parameters used were $R_0 = 3.0$, $r = 0.1$, $\eta = .05$, $\sigma_{proc} = 0.5$, and additive observation noise was drawn from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

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.

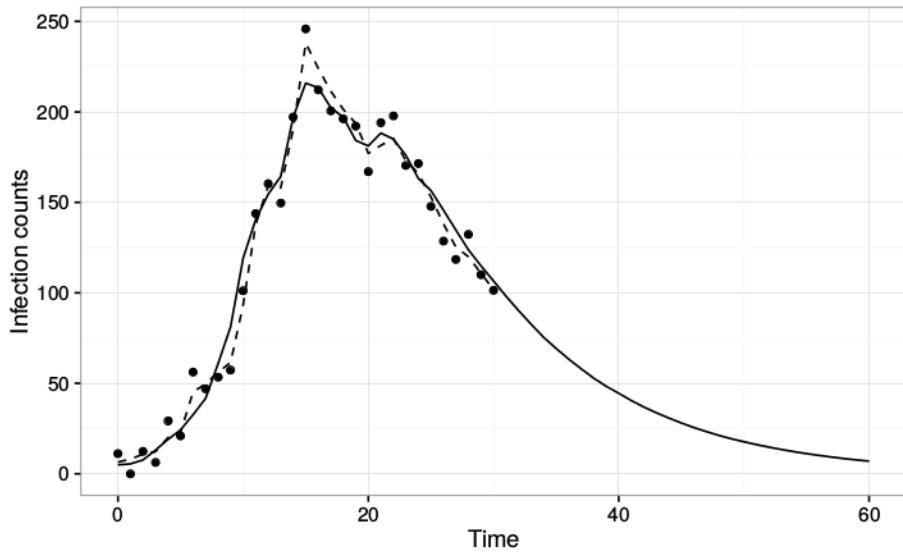


Figure 5.2: Infection count data truncated at $T = 30$ from Figure [5.1]. The dashed line shows IF2’s attempt to reconstruct the true underlying state from the observed data points.

¹ 5.2.1 Parametric Bootstrapping

- 2 The goal of the parametric bootstrap is use an initial density sample θ^* to generate
- 3 further samples $\theta_1, \theta_2, \dots, \theta_M$ from the sampling distribution of θ . It works by using θ 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 θ^* , then use the model’s forward
- 7 simulator to generate new data sets, in essence treating our original estimate θ^* as
- 8 the “truth” set.
- 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 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 SSE

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 $\theta_1, \theta_2, \dots, \theta_M$

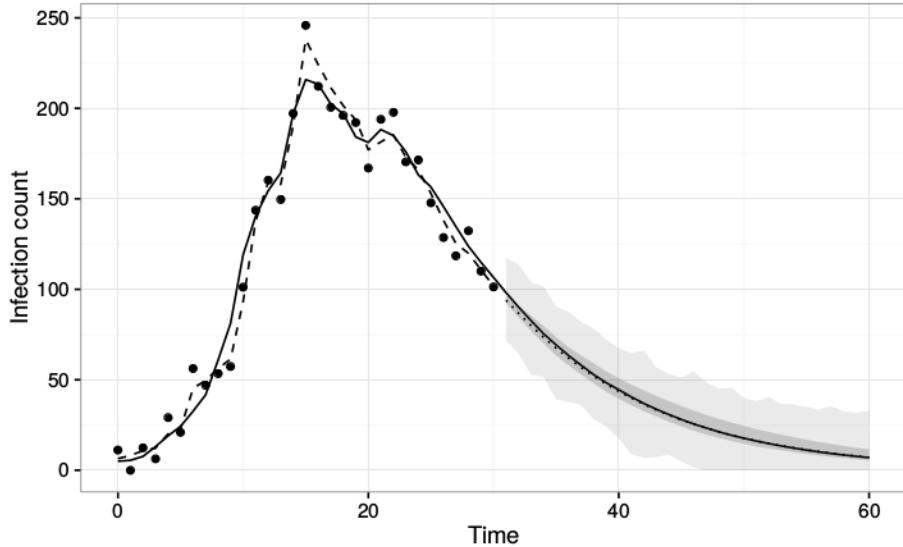


Figure 5.3: Forecast produced by the IF2 / parametric bootstrapping framework. The dotted line shows the mean estimate of the forecasts, the dark grey ribbon shows the 95% confidence interval based on the 0.025 and 0.975 quantiles on the true state estimates, and the lighter grey ribbon shows the same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

1 and dividing that value by the number of values predicted to get the average squared
2 error per point. For the data in Figure [5.3] the value was $\overline{SSE} = 1.67$. Normally
3 we would also want to address questions of forecast coverage, but this would require
4 at least a 100-fold increase in computational cost. This is potentially an avenue of
5 future investigation.

6 5.3 HMCMC

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

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

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

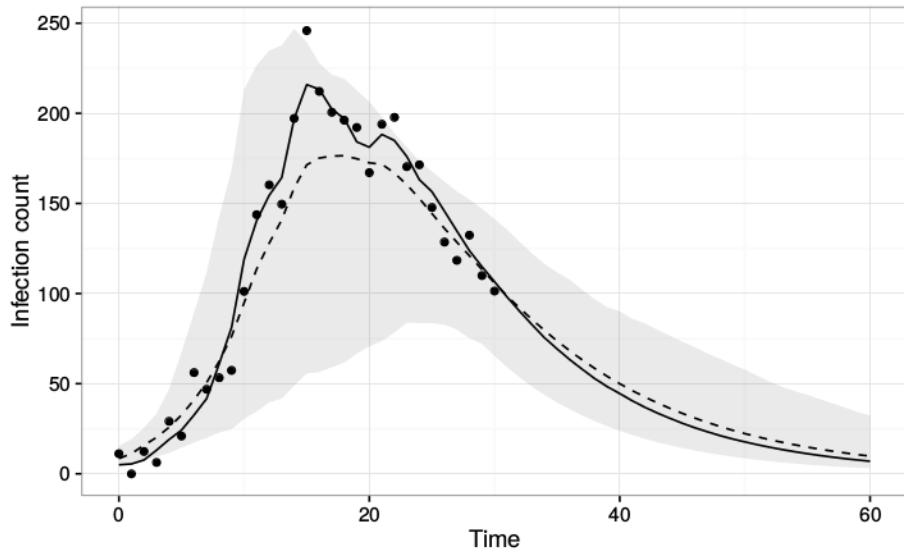


Figure 5.4: Forecast produced by the HMC / bootstrapping framework with $M = 200$ trajectories. The dotted line shows the mean estimate of the forecasts, and the grey ribbon shows the 95% confidence interval on the estimated true states as described in Figure [5.3].

¹ 5.4 Truncation vs. Error

- 2 Of course the above mini-comparison only shows one truncation value for one trajectory.
- 3 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
- 11 to a 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.
- 15 However the now much longer running time can somewhat alleviated by parallelizing
- 16 the parametric bootstrapping process; as each of the parametric bootstrap fittings
- 17 in entirely independent, this can be done without a great deal of additional effort.

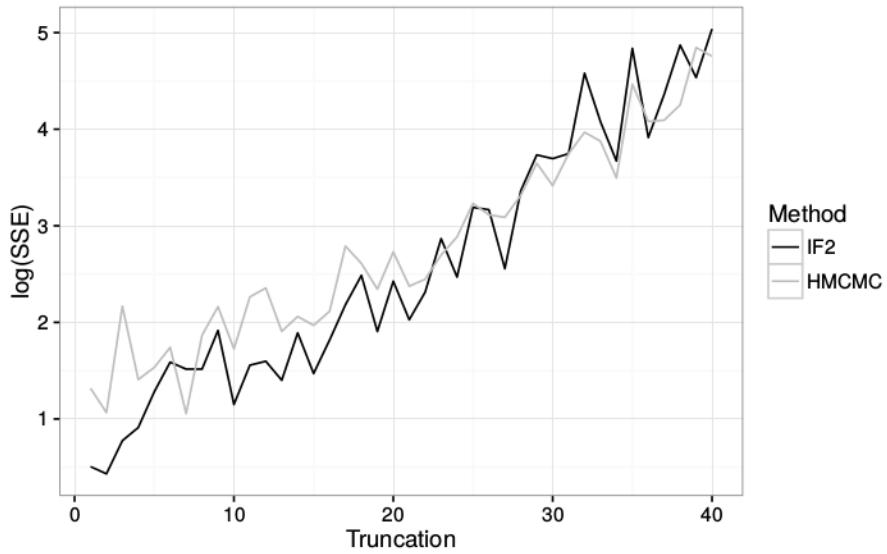


Figure 5.5: Error growth as a function of data truncation amount. Both methods used 200 bootstrap trajectories. Note that the y-axis shows the natural log of the averaged SSE, not the total SSE.

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

¹ Chapter 6

² S-map and SIRS

³ 6.1 S-maps

⁴ A family of forecasting methods that shy away from the mechanistic model-based
⁵ approaches outlined in the previous sections have been developed by George Sugihara
⁶ and collaborators [35][36][18][13] over the last several decades. As these methods do
⁷ not include a mechanistic model in their forecasting process, they also do not attempt
⁸ to perform parameter estimation or inference. Instead they attempt to reconstruct
⁹ the underlying dynamical process as a weighted linear model from a time series.

¹⁰ One such method, the sequential locally weighted global linear maps (S-map), builds
¹¹ a global linear map model and uses it to produce forecasts directly. Despite relying on
¹² a linear mapping, the S-map does not assume the time series on which it is operating
¹³ is the product of linear system dynamics, and in fact was developed to accommodate
¹⁴ non-linear dynamics. The linear component of the method only comes into play when
¹⁵ combining forecast components together to produce a single estimate

¹⁶ The S-map works by first constructing a time series embedding of length E , known as
¹⁷ the library and denoted $\{\mathbf{x}_i\}$. Consider a time series of length T denoted x_1, x_2, \dots, x_T .
¹⁸ Each element in the time series with indices in the range $E, E + 1, \dots, T$ will have a
¹⁹ corresponding entry in the library such that a given element x_t will correspond to a
²⁰ library vector of the form $\mathbf{x}_i = (x_t, x_{t-1}, \dots, x_{t-E+1})$. Next, given a forecast length L
²¹ (representing L time steps into the future), each library vector \mathbf{x}_i is assigned a predic-
²² tion from the time series $y_i = x_{t+L}$, where x_t is the first entry in \mathbf{x}_i . Finally, a forecast
²³ \hat{y}_t for specified predictor vector \mathbf{x}_t (usually from the library itself), is generated using
²⁴ an exponentially weighted function of the library $\{\mathbf{x}_i\}$, predictions $\{y_i\}$, and predictor
²⁵ vector \mathbf{x}_t .

²⁶ 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 θ serves
10 as a way to specify the appropriate level of penalization applied to poorly-matching
11 library vectors – if θ is 0 all weights are the same (no penalization), and increasing θ
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  $\hat{y}_t$ 

```

¹ 6.3 SIRS Model

- ² In an epidemic or infectious disease context, the S-map algorithm will only really work
³ on time series that appear cyclic. While there is nothing mechanically that prevents it
⁴ from operating on a time series that do not appear cyclic, S-mapping requires a long
⁵ time series in order to build a quality library. Without one the forecasting process
⁶ would produce unreliable data.
- ⁷ Given, the S-map's data requirements, we need to specify a modified version of the SIR
⁸ model. As IF2 and HMCMD in principle should be able operate on any reasonably
⁹ well-specified model, the easiest way to compare the efficacy of S-mapping to IF2 or
¹⁰ HMCMD is to generate data from a SIRS model with a seasonal component, and have
¹¹ all methods operate on the resulting time series.
- ¹² The basic skeleton of the SIRS model is similar to the stochastic SIR model described
¹³ previously, with one small addition. The deterministic ODE component of the model
¹⁴ is as follows.

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

- ¹⁶ There are two new features here. We have a rate of waning immunity η through which
¹⁷ people become able to be reinfected, and a seasonality factor function $\Gamma(t)$ defined
¹⁸ as

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

- ²⁰ This function oscillates between 1 and e^{-4} (close to 0) and is meant to represent
²¹ transmission damping during the off-season, for example summer for influenza. Fur-
²² ther, it displays flatter troughs and sharper peaks to exaggerate its effect in peak
²³ season.

- ²⁴ As before, β is allowed to walk restricted by a geometric mean, described by

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

- ²⁶ Figure [6.1] shows the SIRS model simulated for the equivalent of 5 years (260 weeks)
²⁷ and adding noise drawn from $\mathcal{N}(0, \sigma)$.

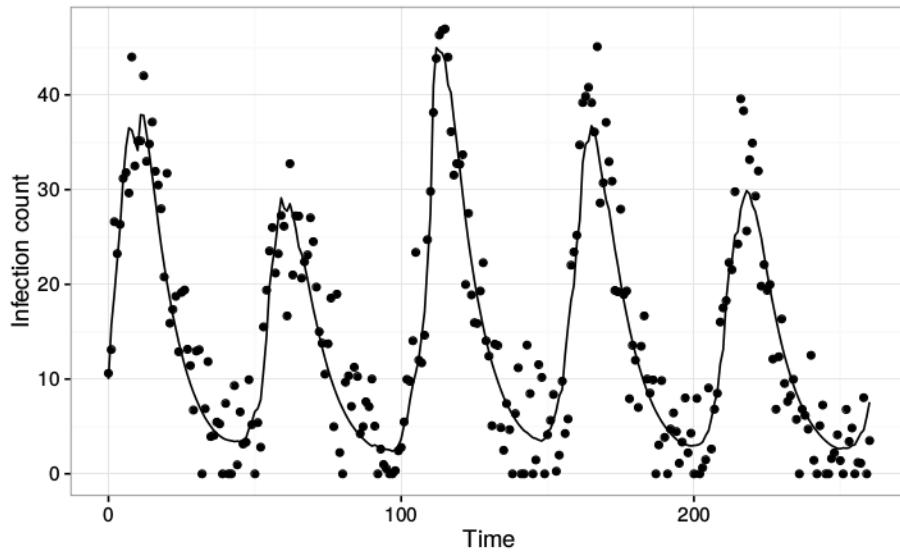


Figure 6.1: Five cycles generated by the SIRS function. The solid line the the true number of cases, dots show case counts with added observation noise. The parameter values were $R_0 = 3.0$, $\gamma = 0.1$, $\eta = 1$, $\sigma = 5$, and 10 initial cases.

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

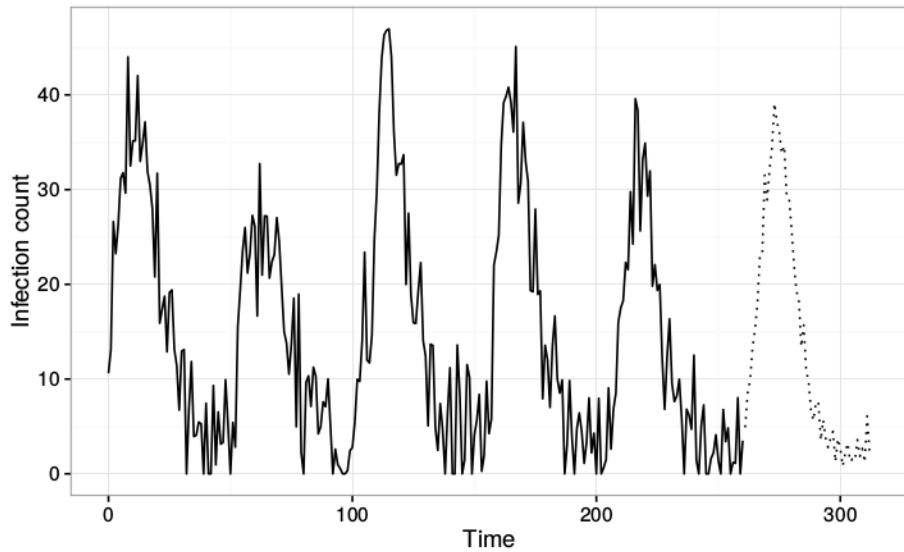


Figure 6.2: S-map applied to the data from the previous figure. The solid line shows the infection counts with observation noise from the previous plot, and the dotted line is the S-map forecast. Parameters chosen were $E = 14$ and $\theta = 3$.

- 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 [13]. 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 HMCMC.
- 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 HMCMC and the S-map for the majority of forecast lengths, with the

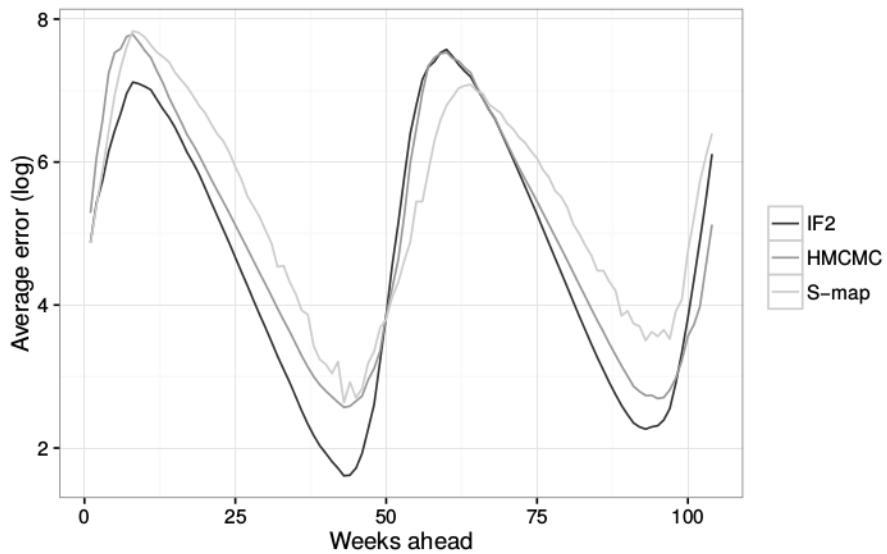


Figure 6.3: Error as a function of forecast length.

- 1 S-map producing the poorest results with the exception of the second rise in infection
- 2 rates where it outperforms the other methods.
- 3 While the S-map may not provide the same fidelity or forecast as IF2 or HMCMD, 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 HMCMD 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 HMCMD.

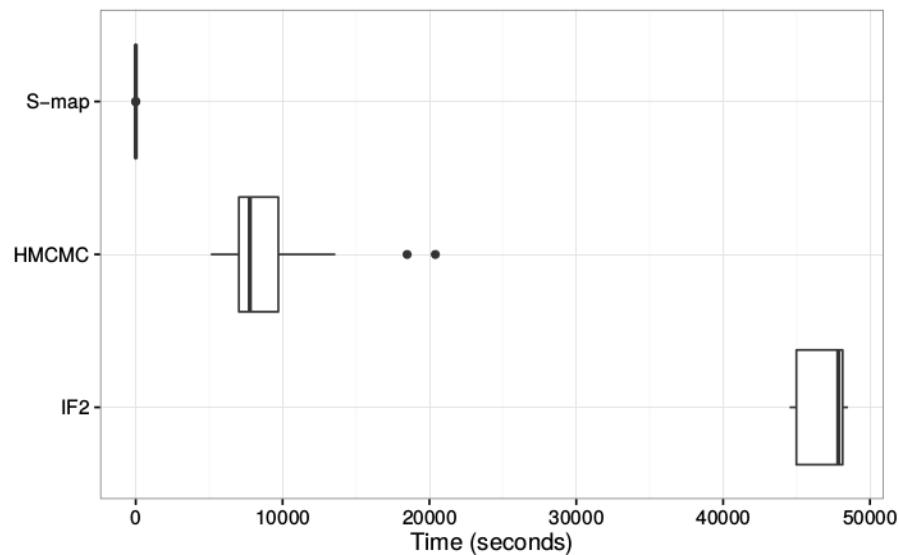


Figure 6.4: Runtimes for producing SIRS forecasts. The box shows the middle 50th quantile, the bold line is the median, and the dots are outliers. Note that these are not “true” outliers, simply ones outside a ranges based on the interquartile range.

¹ Chapter 7

² Spatial Epidemics

³ 7.1 Spatial SIR

⁴ Spatial epidemic models provide a way to capture not just the temporal trend in an
⁵ epidemic, but to also integrate spatial data and infer how the infection is spreading
⁶ in both space and time. One such model we can use is a dynamic spatiotemporal SIR
⁷ model.

⁸ We wish to construct a model build upon the stochastic SIR compartment model
⁹ described previously but one that consists of several connected spatial locations, each
¹⁰ with its own set of compartments. Consider a set of locations numbered $i = 1, \dots, N$,
¹¹ where N is the number of locations. Further, let N_i be the number of neighbours
¹² location i has. The model is then

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

¹⁴ Neighbours for a particular location are numbered $j = 1, \dots, N_i$. We have a new
¹⁵ parameter, $\phi \in [0, 1]$, which is the degree of connectivity. If we let $\phi = 0$ we have
¹⁶ total spatial isolation, and the dynamics reduce to the basic SIR model. If we let
¹⁷ $\phi = 1$ then each of the neighbouring locations will have weight equivalent to the
¹⁸ parent location.

¹⁹ As before we let β embark on a geometric random walk defined as

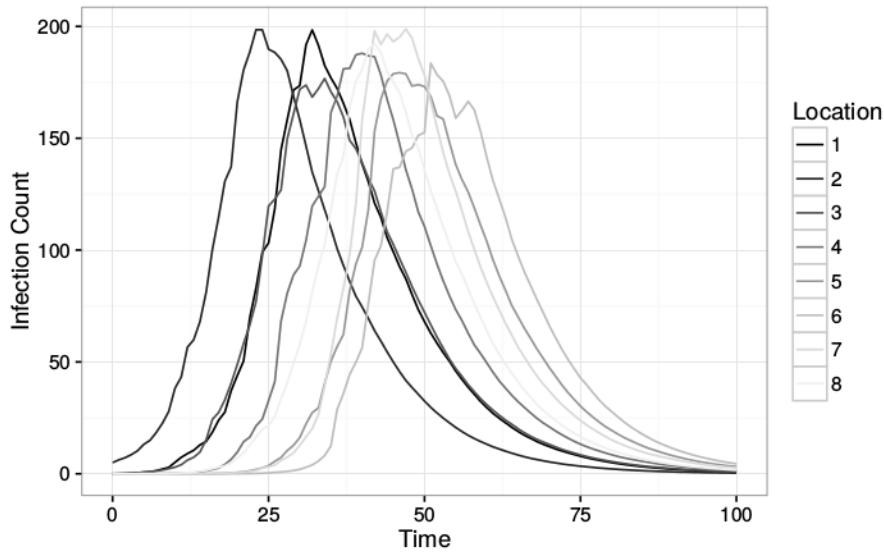


Figure 7.1: Evolution of a spatial epidemic in a ring topology. The outbreak was started with 5 cases in Location 2. Parameters were $R_0 = 3.0$, $\gamma = 0.1$, $\eta = 0.5$, $\sigma_{err} = 0.5$, and $\phi = 0.5$.

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

- 2 Note that as β is a state variable, each location has its own stochastic process driving
- 3 the evolution of its β state.
- 4 If we imagine a circular topology in which each of 10 locations is connected to exactly
- 5 two neighbours (i.e. location 1 is connected to locations N 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].

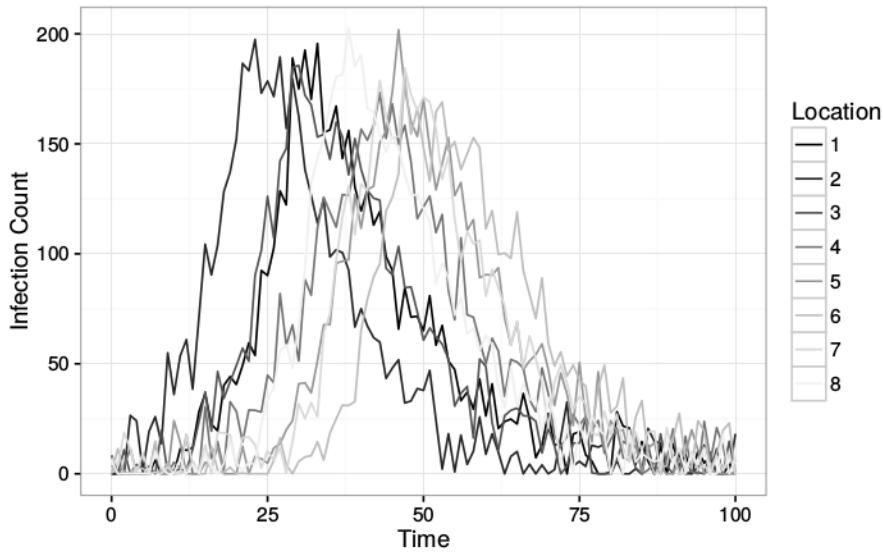


Figure 7.2: Evolution of a spatial epidemic as in Figure [7.1], with added observation noise drawn from $\mathcal{N}(0, 10)$.

¹ 7.2 Dewdrop Regression

- 2 Dewdrop regression [18] aims to overcome the primary disadvantage suffered by meth-
- 3 ods such as the S-map or its cousin Simplex Projection: the requirement of long time
- 4 series from which to build a library. Suggested by Sugihara's group in 2008, Dewdrop
- 5 Regression works by stitching together shorter, related, time series, in order to give
- 6 the S-map or similar methods enough data to operate on. The underlying idea is that
- 7 as long as the underlying dynamics of the time series display similar behaviour (such
- 8 as potentially collapsing to the same attractor), they can be treated as part of the
- 9 same overarching system.
- 10 It is not enough to simply concatenate the shorter time series together – several pro-
- 11 cedures must be carried out and a few caveats observed. First, as the individual time
- 12 series can be or drastically differing scales and breadths, they all must be rescaled to
- 13 unit mean and variance. Then the library is constructed as before with an embedding
- 14 dimension E , but any library vectors that span any of the seams joining the time
- 15 series are discarded. Further, and predictions stemming from a library vector must
- 16 stay within the time series from which they originated. In this way we are allowing
- 17 the “shadow” of the underlying dynamics of the separate time series to infer the
- 18 forecasts for segments of other time series. Once the library has been constructed,
- 19 S-mapping can be carried out as previously specified.
- 20 This procedure is especially well-suited to the spatial model we are using. While
- 21 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 HMCMC, 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 HMCMC 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 HMCMC. HMCMC 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 HMCMC were about 2.90×10^4 and 3.88×10^4 , respectively.
25 This is a speed-up of just over 116x over IF2 and 156x over HMCMC.

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

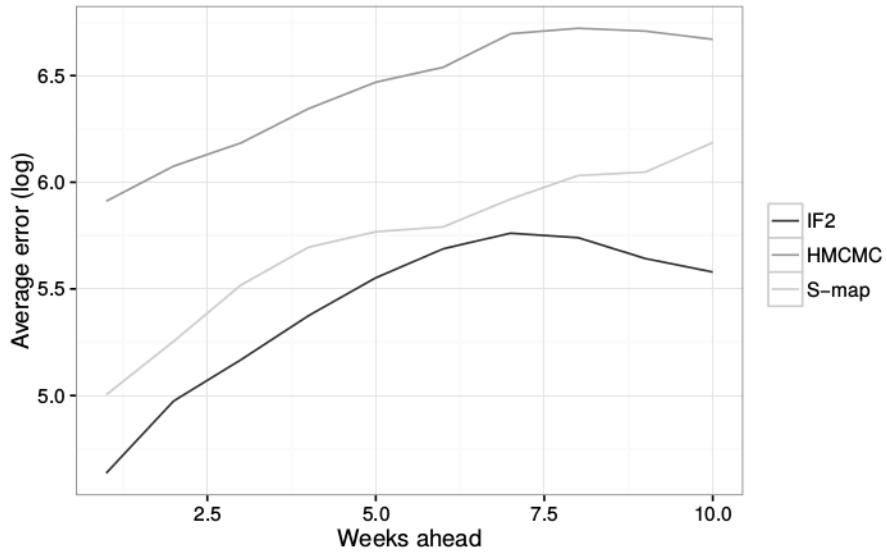


Figure 7.3: Average SSE (log scale) across each location and all trials as a function of the number of weeks ahead in the forecast.

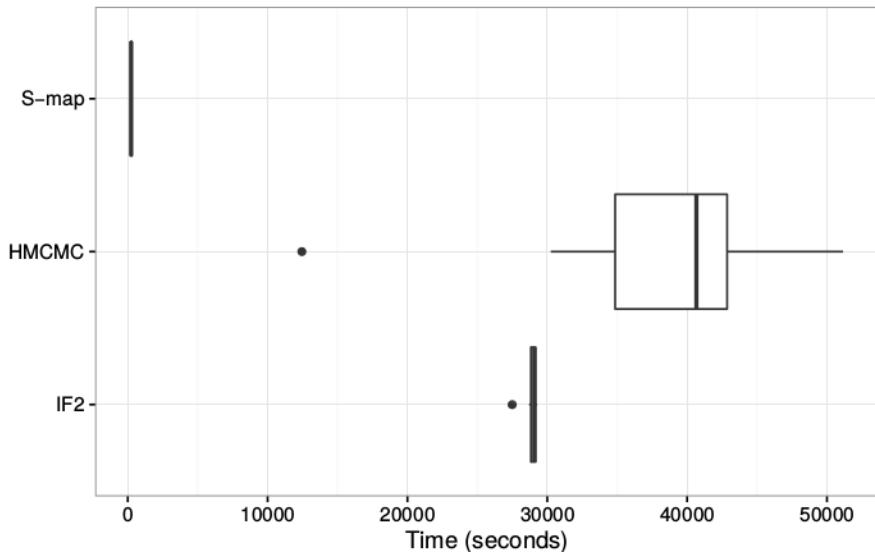


Figure 7.4: Runtimes for producing spatial SIR forecasts. The box shows the middle 50th quantile, the bold line is the median, and the dots are outliers.

¹ Chapter 8

² Discussion and Future Directions

Maybe have a sentence that leads into this, i.e. lets the reader know that this para. is going to summarize the results of the previous (XX) chapters

A little too informal?

- 3 Right off the bat, we can see that the IF2 / parametric bootstrapping framework
- 4 produces great results. This framework consistently out-performs both the HMCMC
- 5 framework and S-mapping by itself or with Dewdrop Regression. This isn't to say
- 6 that the results produced by the other methods are poor, but rather that the ones
- 7 produced by IF2 are **noticeably** better. This is true in every scenario we have explored
- 8 here, and is particularly pronounced in the SIRS and spatial forecasting set-ups.

- 9 A surprise has been how well S-mapping has performed. Given the almost ludicrously
- 10 shorter running times exhibited by S-mapping, it is almost shocking how well it per-
- 11 forms. In the SIRS scenario it produces results only slightly less accurate than the
- 12 other two methods, and is even the most accurate at predicting the rise to the second
- 13 outbreak peak. And in the spatial scenario it performs almost as well as IF2, and
- 14 much better than HMCMC. The critical point here is that S-mapping, with its rela-
- 15 tive ease of implementation, efficiency, and accuracy would make a great "first-blush"
- 16 forecasting tool that could be run and give a good prediction well before the other
- 17 methods could even be set up.

any thoughts about relative cost/guidance of "tuning" stage?

¹⁸ 8.1 Parallel and Distributed Computing

- 19 Whenever running times are discussed, we must consider the current computing land-
- 20 scape and hardware boundaries. In 1965, Intel co-founder Gordon E. Moore published
- 21 a paper in which he observed that the number of transistors per unit area in integrated
- 22 circuits double roughly every year. The consequence of this growth is the approximate
- 23 year-over-year doubling of clock speeds (maximum number of sequential calculations
- 24 performed per second), equivalent to raw performance of the chip. This forecast was
- 25 updated in 1975 to double every 2 years and has held steady until the very recent
- 26 past [38].

the mechanistic
reasons for the
slowing of Moore's
Law are cool, but
I wonder if we
need it. e.g. all
we *really* need
is the knowledge
that due to
physical constraints
current chip
technologies can't
be packed much
more tightly ...

again:
automatic
threading etc.
maybe a *little*
too much detail?

I would give
yourself the
challenge of
cutting this section
by 10-20%

- 1 Recently, transistor growth has begin to falter. This is due to several factors. The
- 2 size of the transistors themselves has become so small that the next generation of
- 3 processors would need to use transistors only 10-15 atoms across, at which point
- 4 their ability to transport electrons becomes unreliable, and their behaviours will start
- 5 to be affected by quantum uncertainty. Second, denser transistor packing would
- 6 require aggressive cooling strategies as the Thermal Design Power (TDP), or the heat
- 7 generated by such chips would increase dramatically.
- 8 To compensate for these limitations, chip manufacturers have instead redesigned the
- 9 internal chip structures to consists to smaller “cores” within a single CPU die. The
- 10 resulting processing power per processor then stays on track with Moore’s Law, but
- 11 keeps the clock speeds of each individual core, and consequently the thermal dissipati-
- 12 on requirement, under control.
- 13 Of course this raises many problems on the software and algorithm side of computing.
- 14 Using several smaller cores instead of a single large has the distinct disadvantage
- 15 of lack of cohesion – the cores must execute instructions completely decoupled from
- 16 each other. This means algorithms have to be redesigned, or at least rewritten at the
- 17 software level to consists of multiple independent pieces that can be run in parallel.
- 18 This practice is known as parallelization.
- 19 Some compilers can actually detect areas in source code that contain obvious room
- 20 for parallel execution (for example loop iterations with no dependence), and auto-
- 21 matically generate machine code that can run on a multiprocessor with little to no
- 22 performance overhead. This technology is still nascent and cannot be relied to oper-
- 23 ate successfully on anything but the most basic algorithms, and so usually we musts
- 24 identify areas for parallelization and take advantage of them or risk not utilizing the
- 25 full power of our machines. Further, high-performance computing essentially requires
- 26 parallelization in its current form as large clusters and supercomputers rely on dis-
- 27 tributed computing “nodes”.
- 28 When working with computationally intensive algorithms, particularly iterative meth-
- 29 ods such those used in this paper, the question of parallelism naturally arises. It may
- 30 come as no surprise that the potential degrees of parallelism varies between meth-
- 31 ods.
- 32 Hamiltonian MCMC is cursed with high dependence between iterations. While HM-
- 33 CMC has an advantage over “vanilla” MCMC formulations in terms of efficiency of
- 34 step acceptance and ease of exploration of the parameter per number of samples, each
- 35 sample still depends entirely on the preceding one, and at a conceptual level the con-
- 36 struction of a Markov Chain *requires* iterative dependence. We cannot simply take
- 37 an accepted step, compute several proposed steps accept/reject them independently –
- 38 doing so would break the chain construction and could potentially bias our posterior
- 39 estimate to boot. We can, however, process multiple chains simultaneously and merge
- 40 the resulting samples. If the required number of samples for a problem were large and

with HMC/MC I wonder if there are *lower-level* opportunities for parallelism (much harder, probably, but worth considering). For example, evaluation of the posterior prob. often involves applying the same function to a bunch of (conditionally) iid observations ... that could be trivially threaded ... this gets deep fast, though.

maybe worth citing "no burn-in" stuff from Geyer (although I don't know if it's in a peer-reviewed pub anywhere)

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

I agree that MCMC has constraints, but something like 5-fold parallelization probably **is** reasonable. Useful for desktops ...

or "distributed"

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

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

- 14 • Forward evolution of the particles' internal system state using their parameter
15 state
16 • Weighting those state estimates against the data point using the observation
17 function
18 • Particle weight normalizations
19 • Resampling from the particle weight distribution
20 • Particle parameter perturbations

21 Luckily, 4 of the 5 steps can be individually parallelized and run on a per-particle
22 basis. The particle weight normalizations, however, cannot. Summation "reductions"
23 are a well-known problem for parallel algorithms; they can be parallelized to a degree
24 using binary reduction, but that only reduces the approximate running time from
25 $\mathcal{O}(n)$ to $\mathcal{O}(\log(n))$. The normalization process requires the particles' weight sum
26 to be determined, hence the unavoidable obstacle of summation reductions rears its
27 head. However this is in practice a less-taxing step, and its more demanding siblings
28 are more amenable to parallelization.

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

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

highly?

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

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

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

for what it's worth prediction times for epidemics are usually on the order of hours/days ... the cases where computational speed will become severely limiting are for e.g. very large complex (spatial/agent-based) models. As I recall foot&mouth disease virus forecasts in early 1990s were somewhat constrained by need for daily forecast updates. (Scenario planning etc. will always multiply these costs ...)

8.2 IF2, Bootstrapping, and Forecasting Methodology

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

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

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

1 tools are best for which jobs.

2 **8.3 Fin**

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

¹ Bibliography

- 2 [1] Mohammad Ali et al. “Time Series Analysis of Cholera in Matlab, Bangladesh,
3 during 1988–2001”. In: *Journal of health, population, ...* 31.1 (2013), pp. 11–19.
4 URL: <http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3702354/>.
- 5 [2] Christophe Andrieu et al. “An introduction to MCMC for machine learning”.
6 In: *Machine Learning* 50.1-2 (2003), pp. 5–43. ISSN: 08856125. DOI: [10.1023/A:1020281327116](https://doi.org/10.1023/A:1020281327116). arXiv: [1109.4435v1](https://arxiv.org/abs/1109.4435v1).
- 8 [3] M.S. Arulampalam et al. “A tutorial on particle filters for online nonlinear/non-
9 Gaussian Bayesian tracking”. In: *IEEE Transactions on Signal Processing* 50.2
10 (2002), pp. 174–188. ISSN: 1053587X. DOI: [10.1109/78.978374](https://doi.org/10.1109/78.978374). URL: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=978374>.
- 12 [4] Jacob Bock Axelsen et al. “Multiannual forecasting of seasonal influenza dy-
13 namics reveals climatic and evolutionary drivers.” In: *Proceedings of the Na-
14 tional Academy of Sciences of the United States of America* 111.26 (July 2014),
15 pp. 9538–42. ISSN: 1091-6490. DOI: [10.1073/pnas.1321656111](https://doi.org/10.1073/pnas.1321656111). URL: <http://www.ncbi.nlm.nih.gov/articlerender.fcgi?artid=4084473%7B%5C&%7Dtool=pmcentrez%7B%5C&%7Drendertype=abstract>.
- 18 [5] M Babyak. “What You See May Not Be What You Get: A Brief, Nontechnical
19 Introduction to Overfitting in Regression Type Models”. In: *J. Bio. Med.*
20 66.3 (2004), pp. 411–421. ISSN: 0033-3174. DOI: [10.1097/01.psy.0000127692.23278.a9](https://doi.org/10.1097/01.psy.0000127692.23278.a9).
- 22 [6] Thomas Bengtsson, Peter Bickel, and Bo Li. “Curse-of-dimensionality revisited:
23 Collapse of the particle filter in very large scale systems”. In: *Probability and
24 Statistics* 2 (2008), pp. 316–334. DOI: [10.1214/193940307000000518](https://doi.org/10.1214/193940307000000518). arXiv:
25 [0805.3034](https://arxiv.org/abs/0805.3034). URL: <http://arxiv.org/abs/0805.3034>.
- 26 [7] a. Camacho et al. “Explaining rapid reinfections in multiple-wave influenza out-
27 breaks: Tristan da Cunha 1971 epidemic as a case study”. In: *Proceedings of the
28 Royal Society B: Biological Sciences* 278 (2011), pp. 3635–3643. ISSN: 0962-8452.
29 DOI: [10.1098/rspb.2011.0300](https://doi.org/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 [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](https://doi.org/10.1371/journal.pone.0094130). URL: <http://www.ncbi.nlm.nih.gov/pmc/articles/PMC4100000/>.
- 3 [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](https://doi.org/10.1371/journal.pone.0023610).
- 4 [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](https://doi.org/10.1371/journal.pone.0056176). URL: <http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3572967/>.
- 5 [12] Dirk Eddelbuettel and Romain Fran. “Rcpp : Seamless R and C ++ Integration”. In: *Journal Of Statistical Software* 40.8 (2011), pp. 1–18. ISSN: 15487660. DOI: [10.1863/73173_14614_6868_4](https://doi.org/10.1863/73173_14614_6868_4). arXiv: [arXiv:1011.1669v3](https://arxiv.org/abs/1011.1669v3). URL: <http://www.jstatsoft.org/v40/i08/>.
- 6 [13] Sarah M. Glaser, Hao Ye, and George Sugihara. “A nonlinear, low data requirement model for producing spatially explicit fishery forecasts”. In: *Fisheries Oceanography* 23 (2014), pp. 45–53. ISSN: 10546006. DOI: [10.1111/fog.12042](https://doi.org/10.1111/fog.12042).
- 7 [14] Andrea L Graham et al. “Explaining rapid reinfections in multiple-wave influenza outbreaks : Tristan da Cunha 1971 epidemic as a case study”. In: *Proc. R. Soc. B* (2016). DOI: [10.1098/rspb.2011.0300](https://doi.org/10.1098/rspb.2011.0300).
- 8 [15] Bradley Harding. “Standard errors : A review and evaluation of standard error estimators using Monte Carlo simulations”. In: (2014), pp. 107–123.
- 9 [16] Florian Hartig et al. “Statistical inference for stochastic simulation models - theory and application”. In: *Ecology Letters* 14.8 (2011), pp. 816–827. ISSN: 1461023X. DOI: [10.1111/j.1461-0248.2011.01640.x](https://doi.org/10.1111/j.1461-0248.2011.01640.x).
- 10 [17] Matthew D. Hoffman and Andrew Gelman. “The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo”. In: *Journal of Machine Learning Research* 15.April (2014), pp. 1593–1623. arXiv: [1111.4246](https://arxiv.org/abs/1111.4246). URL: <http://mc-stan.org/>.
- 11 [18] Chih-hao Hsieh, Christian Anderson, and George Sugihara. “Extending non-linear analysis to short ecological time series.” In: *The American naturalist* 171.1 (2008), pp. 71–80. ISSN: 0003-0147. DOI: [10.1086/524202](https://doi.org/10.1086/524202).

- 1 [19] E L Ionides, C Bret, and a a King. “Inference for nonlinear dynamical systems.”
2 In: *Proceedings of the National Academy of Sciences of the United States of
3 America* 103.49 (Dec. 2006), pp. 18438–43. ISSN: 0027-8424. DOI: [10.1073/pnas.0603181103](https://doi.org/10.1073/pnas.0603181103). URL: <http://www.ncbi.nlm.nih.gov/articlerender.fcgi?artid=3020138%7B%5C&%7Dtool=pmcentrez%7B%5C&%7Drendertype=abstract>.
- 4 [20] Edward L. Ionides et al. “Inference for dynamic and latent variable models
5 via iterated, perturbed Bayes maps”. In: *Proceedings of the National Academy
6 of Sciences* 112.3 (2015), pp. 719–724. ISSN: 0027-8424. DOI: [10.1073/pnas.1410597112](https://doi.org/10.1073/pnas.1410597112). URL: <http://www.pnas.org/lookup/doi/10.1073/pnas.1410597112>.
- 7 [21] Aaron A King, Dao Nguyen, and Edward L. Ionides. “Statistical Inference for
8 Partially Observed Markov Processes via the R Package pomp”. In: *Journal of
9 Statistical Software* 59.10 (2015). arXiv: [arXiv:1509.00503v1](https://arxiv.org/abs/1509.00503v1).
- 11 [22] Aaron A King et al. “Avoidable errors in the modelling of outbreaks of emerging
12 pathogens, with special reference to Ebola”. In: *Proceedings of the Royal
13 Society B: Biological Sciences* 282.1806 (2015), pp. 20150347–20150347. ISSN:
14 0962-8452. DOI: [10.1098/rspb.2015.0347](https://doi.org/10.1098/rspb.2015.0347). arXiv: [1412.0968](https://arxiv.org/abs/1412.0968). URL: [http://rspb.royalsocietypublishing.org/cgi/doi/10.1098/rspb.2015.0347](https://rspb.royalsocietypublishing.org/cgi/doi/10.1098/rspb.2015.0347).
- 16 [23] Aaron A King et al. *pomp: Statistical inference for partially observed Markov
17 processes*. 2016. URL: <http://kingaa.github.io/pomp>.
- 19 [24] William A. Link and Mitchell J. Eaton. “On thinning of chains in MCMC”.
20 In: *Methods in Ecology and Evolution* 3.1 (2012), pp. 112–115. ISSN: 2041210X.
22 DOI: [10.1111/j.2041-210X.2011.00131.x](https://doi.org/10.1111/j.2041-210X.2011.00131.x).
- 24 [25] Kanti V. Mardia et al. “The Kriged Kalman filter”. In: *Test* 7.2 (1998), pp. 217–
25 282. ISSN: 11330686. DOI: [10.1007/BF02565111](https://doi.org/10.1007/BF02565111).
- 26 [26] Radford M Neal. “Handbook of Markov Chain Monte Carlo”. In: *Handbook of
27 Markov Chain Monte Carlo* 20116022 (2011), pp. 113–162. ISSN: [null]. DOI:
28 [10.1201/b10905](https://doi.org/10.1201/b10905). arXiv: [arXiv:1206.1901v1](https://arxiv.org/abs/1206.1901v1). URL: <http://www.crcnetbase.com/doi/book/10.1201/b10905>.
- 30 [27] Elaine O Nsoesie et al. “A systematic review of studies on forecasting the dy-
31 namics of influenza outbreaks.” In: *Influenza and other respiratory viruses* 8.3
32 (May 2014), pp. 309–16. ISSN: 1750-2659. DOI: [10.1111/irv.12226](https://doi.org/10.1111/irv.12226). URL: <http://www.ncbi.nlm.nih.gov/articlerender.fcgi?artid=4181479%7B%5C&%7Dtool=pmcentrez%7B%5C&%7Drendertype=abstract>.
- 34 [28] Elaine Nsoesie, Madhav Marathe, and John Brownstein. “Forecasting peaks of
35 seasonal influenza epidemics.” In: *PLoS currents* 5 (Jan. 2013), pp. 1–14. ISSN:
36 2157-3999. DOI: [10.1371/currents.outbreaks.bb1e879a23137022ea79a8c508b030bc](https://doi.org/10.1371/currents.outbreaks.bb1e879a23137022ea79a8c508b030bc).
38 URL: <http://www.ncbi.nlm.nih.gov/articlerender.fcgi?artid=3712489%7B%5C&%7Dtool=pmcentrez%7B%5C&%7Drendertype=abstract>.
- 39

- 1 [29] Robert C Reiner et al. "Highly localized sensitivity to climate forcing drives
2 endemic cholera in a megacity." In: *Proceedings of the National Academy of
3 Sciences of the United States of America* 109.6 (Feb. 2012), pp. 2033–6. ISSN:
4 1091-6490. DOI: [10.1073/pnas.1108438109](https://doi.org/10.1073/pnas.1108438109). URL: <http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3277579/>.
5 [rendertype=abstract](http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3277579/#rendertype=abstract).
- 6
- 7 [30] Sujit K Sahu. "A Bayesian Kriged-Kalman model for short-term fore- casting of
8 air pollution levels". In: *Appl. Statist* 54 (2005), pp. 223–244. ISSN: 0035-9254.
9 DOI: [10.1111/j.1467-9876.2005.00480.x](https://doi.org/10.1111/j.1467-9876.2005.00480.x).
- 10 [31] Jacques Sau et al. "Particle filter-based real-time estimation and prediction of
11 traffic conditions Modeling framework". In: *Information Systems* (1918), pp. 1–
12 8.
- 13 [32] Jeffrey Shaman and Alicia Karspeck. "Forecasting seasonal outbreaks of in-
14 fluenza." In: *Proceedings of the National Academy of Sciences of the United
15 States of America* 109.50 (Dec. 2012), pp. 20425–30. ISSN: 1091-6490. DOI:
16 [10.1073/pnas.1208772109](https://doi.org/10.1073/pnas.1208772109). URL: <http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3528592/>.
17 [rendertype=abstract](http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3528592/#rendertype=abstract).
- 18
- 19 [33] Jeffrey Shaman, Wan Yang, and Sasikiran Kandula. "Inference and Forecast of
20 the Current West African Ebola Outbreak in Guinea, Sierra Leone and Liberia".
21 In: *PLoS Currents* 6 (2014), ecurrents.outbreaks.3408774290b1a0f2dd7cae877c8b8f.
22 ISSN: 2157-3999. DOI: [10.1371/currents.outbreaks.3408774290b1a0f2dd7cae877c8b8ff6](https://doi.org/10.1371/currents.outbreaks.3408774290b1a0f2dd7cae877c8b8ff6).
- 23 [34] Stan Development Team. "Stan Modeling Language Users Guide and Reference
24 Manual". In: 2.9.0 (2015).
- 25 [35] G Sugihara and R M May. *Nonlinear forecasting as a way of distinguishing
26 chaos from measurement error in time series*. 1990. DOI: [10.1038/344734a0](https://doi.org/10.1038/344734a0).
- 27 [36] George Sugihara. "Nonlinear Forecasting for the Classification of Natural Time
28 Series". In: *Philosophical Transactions of the Royal Society A: Mathematical,
29 Physical and Engineering Sciences* 348 (1994), pp. 477–495. ISSN: 1364-503X.
30 DOI: [10.1098/rsta.1994.0106](https://doi.org/10.1098/rsta.1994.0106).
- 31 [37] Carmen L Vidal Rodeiro and Andrew B Lawson. "Online updating of space-
32 time disease surveillance models via particle filters." In: *Statistical methods in
33 medical research* 15.5 (2006), pp. 423–444. ISSN: 0962-2802. DOI: [10.1177/0962280206071640](https://doi.org/10.1177/0962280206071640).
- 34
- 35 [38] Mitchell Waldrop. "More then Moore". In: *Nature* 530.11. February (2016),
36 p. 145.

- 1 [39] Wan Yang, Alicia Karspeck, and Jeffrey Shaman. “Comparison of filtering meth-
2 ods for the modeling and retrospective forecasting of influenza epidemics.” In:
3 *PLoS computational biology* 10.4 (Apr. 2014), e1003583. ISSN: 1553-7358. DOI:
4 [10.1371/journal.pcbi.1003583](https://doi.org/10.1371/journal.pcbi.1003583). URL: <http://www.ncbi.nlm.nih.gov/pmc/articles/PMC4098879/#h1>
- 5 [40] Xingyu Zhang et al. “Comparative study of four time series methods in fore-
6 casting typhoid fever incidence in China.” In: *PloS one* 8.5 (Jan. 2013), e63116.
7 ISSN: 1932-6203. DOI: [10.1371/journal.pone.0063116](https://doi.org/10.1371/journal.pone.0063116). URL: <http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3641111/#h1>
- 8
9
10
11

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 ## McMaster University
8 3 ## 2016
9 4
10 5 library(deSolve)
11 6 library(rstan)
12 7 library(shinystan)
13 8 library(ggplot2)
14 9 library(RColorBrewer)
15 10 library(reshape2)
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 pars <- c(R0 <- 3.0,      # average number of new infected individuals
2          per infectious person
3 31         r <- 0.1,        # recovery rate
4 32         N <- 500)      # population size
5 33
6 34 T <- 100
7 35 y_ini <- c(S = 495, I = 5, R = 0)
8 36 times <- seq(0, T, by = 1)
9 37
10 38 odeout <- ode(y_ini, times, SIR, pars)
11 39
12 40 set.seed(1001)
13 41 sigma <- 10
14 42 infec_counts_raw <- odeout[,3] + rnorm(T+1, 0, sigma)
15 43 infec_counts     <- ifelse(infec_counts_raw < 0, 0, infec_counts_raw)
16 44
17 45 g <- qplot(0:T, odeout[,3], geom = "line", xlab = "Time (weeks)", ylab
18           = "Infection Count") +
19 46   geom_point(aes(y = infec_counts)) +
20 47   theme_bw()
21 48
22 49 print(g)
23 50 ggsave(g, filename="dataplot.pdf", height=4, width=6.5)
24 51
25 52 sPw <- 7
26 53 datlen <- (T-1)*7 + 1
27 54
28 55 data <- matrix(data = -1, nrow = T+1, ncol = sPw)
29 56 data[,1] <- infec_counts
30 57 standata <- as.vector(t(data))[1:datlen]
31 58
32 59 sir_data <- list( T = datlen,      # simulation time
33           y = standata,    # infection count data
34           N = 500,        # population size
35           h = 1/sPw )    # step size per day
36 63
37 64 rstan_options(auto_write = TRUE)
38 65 options(mc.cores = parallel::detectCores())
39 66 stan_options <- list( chains = 4,      # number of chains
40           iter = 2000,    # iterations per chain
41           warmup = 1000,  # warmup interations
42           thin = 2)      # thinning number
43 70 fit <- stan(file      = "d_sirode_euler.stan",
44           data      = sir_data,
45           chains   = stan_options$chains,
46           iter     = stan_options$iter,
47           warmup   = stan_options$warmup,
48           thin     = stan_options$thin )
49 76
50 77 exfit <- extract(fit, permuted = TRUE, inc_warmup = FALSE)
51 78

```

```

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

```

```

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

```

26 A.2 Full Stan code

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

```

28
29 1## Dexter Barrows
30 2## McMaster University
31 3## 2016
32
33 5data {
34
35 7    int      <lower=1> T;      // total integration steps
36 8    real     y[T];      // observed number of cases
37 9    int      <lower=1> N;      // population size
38 10   real     h;        // step size
39
40 12}
41
42 14parameters {
43
44 16   real <lower=0, upper=10> R0;      // R0
45 17   real <lower=0, upper=10> r;        // recovery rate
46 18   real <lower=0, upper=20> sigma;    // observation error
47 19   real <lower=0, upper=500> y0[3];   // initial conditions

```

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

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 ## Author: Dexter Barrows
7 2 ## Github: dbarrows.github.io
8 3
9 4 library(deSolve)
10 5 library(ggplot2)
11 6 library(reshape2)
12 7 library(gridExtra)
13 8 library(Rcpp)
14 9
15 10 SIR ← function(Time, State, Pars) {
16 11
17 12     with(as.list(c(State, Pars)), {
18 13
19 14         B ← R0*r/N
20 15         BSI ← B*S*I
21 16         rI ← r*I
22 17
23 18         dS = -BSI
24 19         dI = BSI - rI
25 20         dR = rI
26 21
27 22         return(list(c(dS, dI, dR)))
28 23
29 24     })
30 25
31 26 }
32 27
33 28 T ← 100
34 29 N ← 500
```

```

1 30 sigma   ← 10
2 31 i_infec ← 5
3 32
4 33 ## Generate true trajectory and synthetic data
5 34 ##
6 35
7 36 true_init_cond ← c(S = N - i_infec,
8 37           I = i_infec,
9 38           R = 0)
10 39
11 40 true_pars ← c(R0 = 3.0,
12 41           r = 0.1,
13 42           N = 500.0)
14 43
15 44 odeout ← ode(true_init_cond, 0:T, SIR, true_pars)
16 45 trueTraj ← odeout[,3]
17 46
18 47 set.seed(1001)
19 48
20 49 infec_counts_raw ← odeout[,3] + rnorm(T+1, 0, sigma)
21 50 infec_counts      ← ifelse(infec_counts_raw < 0, 0, infec_counts_raw)
22 51
23 52 g ← qplot(0:T, odeout[,3], geom = "line", xlab = "Time (weeks)", ylab
24 53     = "Infection Count") +
25 53     geom_point(aes(y = infec_counts)) +
26 54     theme_bw()
27 55
28 56 print(g)
29 57 ggsave(g, filename="dataplot.pdf", height=4, width=6.5)
30 58
31 59 ## Rcpp stuff
32 60 ##
33 61
34 62 sourceCpp(paste(getwd(),"d_if2.cpp",sep="/"))
35 63
36 64 paramdata ← data.frame(if2(infec_counts, T+1, N))
37 65 colnames(paramdata) ← c("R0", "r", "sigma", "Sinit", "Iinit", "Rinit"
38 66     )
39 66
40 67 ## Parameter density kernels
41 68 ##
42 69
43 70 R0points ← paramdata$R0
44 71 R0kernel ← qplot(R0points, geom = "density", xlab = expression(R[0]),
45 71     ylab = "frequency") +
46 72     geom_vline(aes(xintercept=true_pars[["R0"]]), linetype="
47 72         dashed", size=1, color="grey50") +
48 73     theme_bw()
49 74
50 75 print(R0kernel)
51 76 ggsave(R0kernel, filename="kernelR0.pdf", height=3, width=3.25)

```

```

1 77 rpoints <- paramdata$r
2 78 rkernel <- qplot(rpoints, geom = "density", xlab = "r", ylab =
3 79   frequency) +
4 80     geom_vline(aes(xintercept=true_pars[["r"]]), linetype="dashed"
5 81       , size=1, color="grey50") +
6 82       theme_bw()
7 83 print(rkernel)
8 84 ggsave(rkernel, filename="kernelr.pdf", height=3, width=3.25)
9 85
10 86 sigmapoints <- paramdata$sigma
11 87 sigmakernel <- qplot(sigmapoints, geom = "density", xlab = expression(
12 88   sigma), ylab = "frequency") +
13 89     geom_vline(aes(xintercept=sigma), linetype="dashed", size=1,
14 90       color="grey50") +
15 91       theme_bw()
16 92 print(sigmakernel)
17 93 ggsave(sigmakernel, filename="kernelsigma.pdf", height=3, width=3.25)
18 94
19 95 infecpoints <- paramdata$Iinit
20 96 infeckernel <- qplot(infecpoints, geom = "density", xlab = "Initial
21 97   Infected", ylab = "frequency") +
22 98     geom_vline(aes(xintercept=true_init_cond[['I']]), linetype="
23 99       dashed", size=1, color="grey50") +
24 100       theme_bw()
25 101
26 102 # show grid
27 103 grid.arrange(R0kernel, rkernel, sigmakernel, infeckernel, ncol = 2,
28 104   nrow = 2)
29 105 pdf("if2kernels.pdf", height = 6.5, width = 6.5)
30 106 grid.arrange(R0kernel, rkernel, sigmakernel, infeckernel, ncol = 2,
31 107   nrow = 2)
32 108 dev.off()
33 109 #ggsave(filename="if2kernels.pdf", g2, height=6.5, width=6.5)

```

42 B.2 Full C++ code

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

```

44
45 /* Author: Dexter Barrows
46 Github: dbarrows.github.io
47 3

```

```
1   4      */
2   5
3   6 #include <stdio.h>
4   7 #include <math.h>
5   8 #include <sys/time.h>
6   9 #include <time.h>
7  10 #include <stdlib.h>
8  11 #include <string>
9  12 #include <cmath>
10 13 #include <cstdlib>
11 14 #include <fstream>
12 15
13 16 //#include "rand.h"
14 17 //#include "timer.h"
15 18
16 19 #define Treal    100          // time to simulate over
17 20 #define R0true   3.0          // infectiousness
18 21 #define rtrue    0.1          // recovery rate
19 22 #define Nreal    500.0        // population size
20 23 #define merr     10.0         // expected measurement error
21 24 #define I0       5.0          // Initial infected individuals
22 25
23 26 #include <Rcpp.h>
24 27 using namespace Rcpp;
25 28
26 29
27 30 struct Particle {
28 31     double R0;
29 32     double r;
30 33     double sigma;
31 34     double S;
32 35     double I;
33 36     double R;
34 37     double Sinit;
35 38     double Iinit;
36 39     double Rinit;
37 40 };
38 41
39 42 struct ParticleInfo {
40 43     double R0mean;      double R0sd;
41 44     double rmean;        double rsd;
42 45     double sigmamean;   double sigmasd;
43 46     double Sinitmean;   double Sinitsd;
44 47     double Iinitmean;   double Iinitsd;
45 48     double Rinitmean;   double Rinitsd;
46 49 };
47 50
48 51
49 52 int timeval_subtract (double *result, struct timeval *x, struct
50      timeval *y);
51 53 int check_double(double x, double y);
```

```

1  54 void exp_euler_SIR(double h, double t0, double tn, int N, Particle *
2      particle);
3  55 void copyParticle(Particle * dst, Particle * src);
4  56 void perturbParticles(Particle * particles, int N, int NP, int
5      passnum, double coolrate);
6  57 bool isCollapsed(Particle * particles, int NP);
7  58 void particleDiagnostics(ParticleInfo * partInfo, Particle *
8      particles, int NP);
9  59 NumericMatrix if2(NumericVector * data, int T, int N);
10 60 double randu();
11 61 double randn();
12 62
13 // [[Rcpp::export]]
14 64 NumericMatrix if2(NumericVector data, int T, int N) {
15 65
16 66     int      NP          = 2500;
17 67     int      nPasses     = 50;
18 68     double   coolrate    = 0.975;
19 69
20 70     int      i_infec     = 10;
21 71
22 72     NumericMatrix paramdata(NP, 6);
23 73
24 74     srand(time(NULL));    // Seed PRNG with system time
25 75
26 76     double w[NP];        // particle weights
27 77
28 78     Particle particles[NP]; // particle estimates for current
29         step
30 79     Particle particles_old[NP]; // intermediate particle states for
31         resampling
32 80
33 81     printf("Initializing particle states\n");
34 82
35 83     // initialize particle parameter states (seeding)
36 84     for (int n = 0; n < NP; n++) {
37 85
38 86         double R0can, rcan, sigmacan, Iinitcan;
39 87
40 88         do {
41 89             R0can = R0true + R0true*randn();
42 90         } while (R0can < 0);
43 91         particles[n].R0 = R0can;
44 92
45 93         do {
46 94             rcan = rtrue + rtrue*randn();
47 95         } while (rcan < 0);
48 96         particles[n].r = rcan;
49 97
50 98         do {
51 99             sigmacan = merr + merr*randn();

```

```

1 100      } while (sigmacan < 0);
2 101      particles[n].sigma = sigmacan;
3 102
4 103      do {
5 104          Iinitcan = i_infec + i_infec*randn();
6 105      } while (Iinitcan < 0 || N < Iinitcan);
7 106      particles[n].Sinit = N - Iinitcan;
8 107      particles[n].Iinit = Iinitcan;
9 108      particles[n].Rinit = 0.0;
10 109
11 110 }
12 111
13 112 // START PASSES THROUGH DATA
14 113
15 114     printf("Starting filter\n");
16 115     printf("-----\n");
17 116     printf("Pass\n");
18 117
19 118
20 119     for (int pass = 0; pass < nPasses; pass++) {
21 120
22 121         printf("...%d / %d\n", pass, nPasses);
23 122
24 123         perturbParticles(particles, N, NP, pass, coolrate);
25 124
26 125         // initialize particle system states
27 126         for (int n = 0; n < NP; n++) {
28 127
29 128             particles[n].S = particles[n].Sinit;
30 129             particles[n].I = particles[n].Iinit;
31 130             particles[n].R = particles[n].Rinit;
32 131
33 132     }
34 133
35 134     // between-pass perturbations
36 135
37 136     for (int t = 1; t < T; t++) {
38 137
39 138         // between-iteration perturbations
40 139         perturbParticles(particles, N, NP, pass, coolrate);
41 140
42 141         // generate individual predictions and weight
43 142         for (int n = 0; n < NP; n++) {
44 143
45 144             exp_euler_SIR(1.0/10.0, 0.0, 1.0, N, &particles[n]);
46 145
47 146             double merr_par = particles[n].sigma;
48 147             double y_diff = data[t] - particles[n].I;
49 148
50 149             w[n] = 1.0/(merr_par*sqrt(2.0*M_PI)) * exp( - y_diff*
51 150                 y_diff / (2.0*merr_par*merr_par) );

```

```
1 150        }
2 151    }
3 152
4 153    // cumulative sum
5 154    for (int n = 1; n < NP; n++) {
6 155        w[n] += w[n-1];
7 156    }
8 157
9 158    // save particle states to resample from
10 159    for (int n = 0; n < NP; n++){
11 160        copyParticle(&particles_old[n], &particles[n]);
12 161    }
13 162
14 163    // resampling
15 164    for (int n = 0; n < NP; n++) {
16 165
17 166        double w_r = randu() * w[NP-1];
18 167        int i = 0;
19 168        while (w_r > w[i]) {
20 169            i++;
21 170        }
22 171
23 172        // i is now the index to copy state from
24 173        copyParticle(&particles[n], &particles_old[i]);
25 174
26 175    }
27 176
28 177}
29 178
30 179}
31 180
32 181 ParticleInfo pInfo;
33 182 particleDiagnostics(&pInfo, particles, NP);
34 183
35 184 printf("Parameter results (mean | sd)\n");
36 185 printf("-----\n");
37 186 printf("R0      %f %f\n", pInfo.R0mean, pInfo.R0sd);
38 187 printf("r       %f %f\n", pInfo.rmean, pInfo.rsd);
39 188 printf("sigma   %f %f\n", pInfo.sigmaxmean, pInfo.sigmasd);
40 189 printf("S_init  %f %f\n", pInfo.Sinitmean, pInfo.Sinitsd);
41 190 printf("I_init   %f %f\n", pInfo.Iinitmean, pInfo.Iinitsd);
42 191 printf("R_init   %f %f\n", pInfo.Rinitmean, pInfo.Rinitsd);
43 192
44 193 printf("\n");
45 194
46 195
47 196
48 197    // Get particle results to pass back to R
49 198
50 199    for (int n = 0; n < NP; n++) {
51 200}
```

```

1 201      paramdata(n, 0) = particles[n].R0;
2 202      paramdata(n, 1) = particles[n].r;
3 203      paramdata(n, 2) = particles[n].sigma;
4 204      paramdata(n, 3) = particles[n].Sinit;
5 205      paramdata(n, 4) = particles[n].Iinit;
6 206      paramdata(n, 5) = particles[n].Rinit;
7 207
8 208  }
9 209
10 210  return paramdata;
11 211
12 212 }

13 213
14 214

15 215 /* Use the Explicit Euler integration scheme to integrate SIR model
16 forward in time
17 216 double h      - time step size
18 217 double t0     - start time
19 218 double tn     - stop time
20 219 double * y    - current system state; a three-component vector
21 representing [S I R], susceptible-infected-recovered
22 220 */
23 221 void exp_euler_SIR(double h, double t0, double tn, int N, Particle *
24 particle) {
25
26 224     int num_steps = floor( (tn-t0) / h );
27 225
28 226     double S = particle->S;
29 227     double I = particle->I;
30 228     double R = particle->R;
31 229
32 230     double R0    = particle->R0;
33 231     double r     = particle->r;
34 232     double B     = R0 * r / N;
35 233
36 234     for(int i = 0; i < num_steps; i++) {
37 235         // get derivatives
38 236         double dS = - B*S*I;
39 237         double dI = B*S*I - r*I;
40 238         double dR = r*I;
41 239         // step forward by h
42 240         S += h*dS;
43 241         I += h*dI;
44 242         R += h*dR;
45 243     }
46 244
47 245     particle->S = S;
48 246     particle->I = I;
49 247     particle->R = R;
50 248
51 249

```

```

1 249 }
2 250
3 251
4 252 /* Particle perturbation function to be run between iterations and
5 passes
6 253 */
7 254 void perturbParticles(Particle * particles, int N, int NP, int
8 passnum, double coolrate) {
9
10 256     double coolcoef = pow(coolrate, passnum);
11 257
12 258     double spreadR0      = coolcoef * R0true / 10.0;
13 259     double spreadr       = coolcoef * rtrue   / 10.0;
14 260     double spreadsigma   = coolcoef * merr    / 10.0;
15 261     double spreadIinit   = coolcoef * I0      / 10.0;
16 262
17 263
18 264     double R0can, rcan, sigmacan, Iinitcan;
19 265
20 266     for (int n = 0; n < NP; n++) {
21 267
22 268         do {
23 269             R0can = particles[n].R0 + spreadR0*randn();
24 270         } while (R0can < 0);
25 271         particles[n].R0 = R0can;
26 272
27 273         do {
28 274             rcan = particles[n].r + spreadr*randn();
29 275         } while (rcan < 0);
30 276         particles[n].r = rcan;
31 277
32 278         do {
33 279             sigmacan = particles[n].sigma + spreadsigma*randn();
34 280         } while (sigmacan < 0);
35 281         particles[n].sigma = sigmacan;
36 282
37 283         do {
38 284             Iinitcan = particles[n].Iinit + spreadIinit*randn();
39 285         } while (Iinitcan < 0 || Iinitcan > 500);
40 286         particles[n].Iinit = Iinitcan;
41 287         particles[n].Sinit = N - Iinitcan;
42 288
43 289     }
44 290
45 291 }
46 292
47 293
48 294 /* Convenience function for particle resampling process
49 295 */
50 296 void copyParticle(Particle * dst, Particle * src) {
51 297

```

```

1 298     dst->R0      = src->R0;
2 299     dst->r       = src->r;
3 300     dst->sigma   = src->sigma;
4 301     dst->S       = src->S;
5 302     dst->I       = src->I;
6 303     dst->R       = src->R;
7 304     dst->Sinit   = src->Sinit;
8 305     dst->Iinit   = src->Iinit;
9 306     dst->Rinit   = src->Rinit;
10 307
11 308
12 309 }
13 310
14 311
15 312 /* Checks to see if particles are collapsed
16 313 This is done by checking if the standard deviations between the
17 314 particles' parameter
18 315 values are significantly close to one another. Spread threshold
19 316 may need to be tuned.
20 317 */
21 318 bool isCollapsed(Particle * particles, int NP) {
22 319     bool retVal;
23 320
24 321     double R0mean = 0, rmean = 0, sigmamean = 0, Sinitmean = 0,
25 322         Iinitmean = 0, Rinitmean = 0;
26 323
27 324     // means
28 325
29 326     for (int n = 0; n < NP; n++) {
30 327
31 328         R0mean     += particles[n].R0;
32 329         rmean      += particles[n].r;
33 330         sigmamean  += particles[n].sigma;
34 331         Sinitmean  += particles[n].Sinit;
35 332         Iinitmean  += particles[n].Iinit;
36 333         Rinitmean  += particles[n].Rinit;
37 334
38 335     }
39 336
40 337     R0mean     /= NP;
41 338     rmean      /= NP;
42 339     sigmamean  /= NP;
43 340     Sinitmean  /= NP;
44 341     Iinitmean  /= NP;
45 342     Rinitmean  /= NP;
46 343
47 344     double R0sd = 0, rsd = 0, sigmasd = 0, Sinitsd = 0, Iinitsd = 0,
48
49 345             Rinitsd = 0;
50
51 346

```

```

1 345    for (int n = 0; n < NP; n++) {
2 346
3 347        R0sd    += ( particles[n].R0 - R0mean ) * ( particles[n].R0 -
4 348            R0mean );
5 348        rsd     += ( particles[n].r - rmean ) * ( particles[n].r -
6 349            rmean );
7 349        sigmasd += ( particles[n].sigma - sigmamean ) * ( particles[n].
8 350            sigma - sigmamean );
9 350        Sinitsd += ( particles[n].Sinit - Sinitmean ) * ( particles[n].
10 351            Sinit - Sinitmean );
11 351        Iinitsd += ( particles[n].Iinit - Iinitmean ) * ( particles[n].
12 352            Iinit - Iinitmean );
13 352        Rinitsd += ( particles[n].Rinit - Rinitmean ) * ( particles[n].
14 353            Rinit - Rinitmean );
15 353
16 354    }
17 355
18 356    R0sd      /= NP;
19 357    rsd       /= NP;
20 358    sigmasd   /= NP;
21 359    Sinitsd   /= NP;
22 360    Iinitsd   /= NP;
23 361    Rinitsd   /= NP;
24 362
25 363    if ( (R0sd + rsd + sigmasd) < 1e-5)
26 364        retVal = true;
27 365    else
28 366        retVal = false;
29 367
30 368    return retVal;
31 369
32 370}
33 371
34 372 void particleDiagnostics(ParticleInfo * partInfo, Particle *
35 373     particles, int NP) {
36 373
37 374     double R0mean      = 0.0,
38 375             rmean        = 0.0,
39 376             sigmamean    = 0.0,
40 377             Sinitmean    = 0.0,
41 378             Iinitmean    = 0.0,
42 379             Rinitmean    = 0.0;
43 380
44 381     // means
45 382
46 383     for (int n = 0; n < NP; n++) {
47 384
48 385         R0mean      += particles[n].R0;
49 386         rmean       += particles[n].r;
50 387         sigmamean   += particles[n].sigma;
51 388         Sinitmean   += particles[n].Sinit;

```

```

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

```

```
1 434     partInfo->Sinitmean = Sinitmean;
2 435     partInfo->Sinitsd   = Sinitsd;
3 436     partInfo->Iinitmean = Iinitmean;
4 437     partInfo->Iinitsd   = Iinitsd;
5 438     partInfo->Rinitmean = Rinitmean;
6 439     partInfo->Rinitsd   = Rinitsd;
7 440 }
8 441 }
9 442
10 443 double randu() {
11 444
12 445     return (double) rand() / (double) RAND_MAX;
13 446
14 447 }
15 448
16 449
17 450 /* Return a normally distributed random number with mean 0 and
18 standard deviation 1
19 Uses the polar form of the Box-Muller transformation
20 From http://www.design.caltech.edu/erik/Misc/Gaussian.html
21 */
22 451 double randn() {
23 452
24 453     double x1, x2, w, y1;
25 454
26 455     do {
27 456         x1 = 2.0 * randu() - 1.0;
28 457         x2 = 2.0 * randu() - 1.0;
29 458         w = x1 * x1 + x2 * x2;
30 459     } while ( w >= 1.0 );
31 460
32 461     w = sqrt( (-2.0 * log( w ) ) / w );
33 462     y1 = x1 * w;
34 463
35 464     return y1;
36 465
37 466 }
```

¹ Appendix C

² Parameter Fitting

¹ Appendix D

² Forecasting Frameworks

³ D.1 IF2 Parametric Bootstrapping Function

⁴ The parametric bootstrapping machinery used to produce forecasts.

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

```

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

```

```

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

28 D.2 RStan Forward Simulator

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

```

31
32 1 StocSIRstan <- function(y, pars, T, steps, berrvec, bveclim) {
33
34 2   out <- matrix(NA, nrow = (T+1), ncol = 4)
35
36 3   R0 <- pars[['R0']]
37 4   r <- pars[['r']]
38 5   N <- pars[['N']]
39 6   eta <- pars[['eta']]
40 7   berr <- pars[['berr']]
41
42 8   S <- y[['S']]
43 9   I <- y[['I']]
44 10  R <- y[['R']]
45
46 11  B0 <- R0 * r / N
47 12  B <- B0
```

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

¹ Appendix E

² S-map and SIRS

³ E.1 SIRS R Function Code

⁴ R code to simulate the outlines SIRS function.

```
5
6 1 StocSIRS <- function(y, pars, T, steps) {
7 2
8 3   out <- matrix(NA, nrow = (T+1), ncol = 4)
9 4
10 5   R0 <- pars[['R0']]
11 6   r <- pars[['r']]
12 7   N <- pars[['N']]
13 8   eta <- pars[['eta']]
14 9   berr <- pars[['berr']]
15 10  re <- pars[['re']]
16 11
17 12  S <- y[['S']]
18 13  I <- y[['I']]
19 14  R <- y[['R']]
20 15
21 16  B0 <- R0 * r / N
22 17  B <- B0
23 18
24 19  out[1,] <- c(S,I,R,B)
25 20
26 21  h <- 1 / steps
27 22
28 23  for ( i in 1:(T*steps) ) {
29 24
30 25    #Bfac <- 1/2 - cos((2*pi/365)*i)/2
31 26    Bfac <- exp(2*cos((2*pi/365)*i) - 2)
32 27
33 28    B <- exp( log(B) + eta*(log(B0) - log(B)) + rnorm(1, 0, berr) )
34 29
```

```

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

```

38 E.2 SMAP Code

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

```

40
41 1 library(pracma)
42 2
43 3 smap <- function(data, E, theta, stepsAhead) {
44 4
45 5   # construct library
46 6   tseries <- as.vector(data)
47 7   liblen <- length(tseries) - E + 1 - stepsAhead

```

```

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

```

```

1 58
2 59     return(predictions)
3 60
4 61 }

```

6 E.3 SMAP Parameter Optimization Code

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

```

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

```

```
1 38##  
2 39Elist ← 1:20  
3 40thetalist ← 10*exp(-(seq(0,9.5,0.5)))  
4 41nTrials ← 100  
5 42  
6 43ssemat ← matrix(NA, 20, 20)  
7 44  
8 45for (i in 1:length(Elist)) {  
9 46  for (j in 1:length(thetalist)) {  
10 47  
11 48    ssemean ← 0  
12 49  
13 50    for (k in 1:nTrials) {  
14 51  
15 52      E ← Elist[i]  
16 53      theta ← thetalist[j]  
17 54  
18 55      ## get true trajectory  
19 56      ##  
20 57      sdeout ← StocSIRS(true_init_cond, true_pars, T, steps)  
21 58  
22 59      ## perturb to get data  
23 60      ##  
24 61      infec_counts_raw ← sdeout[1:(Tlim+1), 'I'] + rnorm(Tlim+1, 0,  
25 62          sigma)  
26 62      infec_counts ← ifelse(infec_counts_raw < 0, 0, infec_counts_  
27 63          raw)  
28 63  
29 64      predictions ← smap(infec_counts, E, theta, 52)  
30 65  
31 66      err ← sdeout[(Tlim+2):dim(sdeout)[1], 'I'] - predictions  
32 67      sse ← sum(err^2)  
33 68  
34 69      ssemean ← ssemean + (sse / nTrials)  
35 70  
36 71    }  
37 72  
38 73    ssemat[i,j] ← ssemean  
39 74  
40 75  
41 76  }  
42 77}  
43 78  
44 79quartz()  
45 80image(-ssemat)  
46 81quartz()  
47 82filled.contour(-ssemat)  
48 83  
49 84# print(ssemat)  
50 85# cms ← colMeans(ssemat)  
51 86# rms ← rowMeans(ssemat)
```

```

1  87
2  88 #Emin <- Elist[which.min(rms)]
3  89 #thetamin <- thetalist[which.min(cms)]
4  90 #print(Emin)
5  91 #print(thetamin)
6  92
7  93 mininds <- which(ssemat==min(ssemat),arr.ind=TRUE)
8  94
9  95 Emin <- Elist[mininds[, 'row']]
10 96 thetamin <- thetalist[mininds[, 'col']]
11 97
12 98 print(Emin)
13 99 print(thetamin)

```

¹⁵ E.4 RStan SIRS Code

¹⁶ This code implements a periodic SIRS model in Rstan.

```

17
18 1 data {
19
20 3     int      <lower=1>    T;      // total integration steps
21 4     real                 y[T];   // observed number of cases
22 5     int      <lower=1>    N;      // population size
23 6     real                 h;      // step size
24
25 8 }
26
27 10 parameters {
28
29 12     real <lower=0, upper=10>      R0;      // R0
30 13     real <lower=0, upper=10>      r;       // recovery rate
31 14     real <lower=0, upper=10>      re;      // resusceptibility rate
32 15     real <lower=0, upper=20>      sigma;   // observation error
33 16     real <lower=0, upper=30>      Iinit;   // initial infected
34 17     real <lower=0, upper=1>       eta;     // geometric walk
35         attraction strength
36 18     real <lower=0, upper=1>       berr;    // beta walk noise
37 19     real <lower=-1.5, upper=1.5> Bnoise[T]; // Beta vector
38
39 21 }
40
41 23 //transformed parameters {
42 24 //     real B0 <- R0 * r / N;
43 25 //}
44
45 27 model {
46
47 29     real S[T];

```

```

1 30    real I[T];
2 31    real R[T];
3 32    real B[T];
4 33    real B0;
5 34
6 35    real pi;
7 36    real Bfac;
8 37
9 38    pi ← 3.1415926535;
10 39
11 40    B0 ← R0 * r / N;
12 41
13 42    B[1] ← B0;
14 43
15 44    S[1] ← N - Iinit;
16 45    I[1] ← Iinit;
17 46    R[1] ← 0.0;
18 47
19 48    for (t in 2:T) {
20 49
21 50        Bnoise[t] ~ normal(0,berr);
22 51        Bfac ← exp(2*cos((2*pi/365)*t) - 2);
23 52        B[t] ← exp( log(B0) + eta * ( log(B[t-1]) - log(B0) ) +
24 53            Bnoise[t] );
25 54
26 54        S[t] ← S[t-1] + h*( - Bfac*B[t]*S[t-1]*I[t-1] + re*R[t-1] );
27 55        I[t] ← I[t-1] + h*( Bfac*B[t]*S[t-1]*I[t-1] - I[t-1]*r );
28 56        R[t] ← R[t-1] + h*( I[t-1]*r - re*R[t-1] );
29 57
30 58        if (y[t] > 0) {
31 59            y[t] ~ normal( I[t], sigma );
32 60        }
33 61
34 62    }
35 63
36 64    R0      ~ lognormal(1,1);
37 65    r       ~ lognormal(1,1);
38 66    sigma   ~ lognormal(1,1);
39 67    re      ~ lognormal(1,1);
40 68    Iinit   ~ normal(y[1], sigma);
41 69
42 70}

```

44 E.5 IF2 SIRS Code

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

46

47 /* Author: Dexter Barrows

```
1  2      Github: dbarrows.github.io
2
3
4  5
5  6 #include <stdio.h>
6  7 #include <math.h>
7  8 #include <sys/time.h>
8  9 #include <time.h>
9 10 #include <stdlib.h>
10 11 #include <string>
11 12 #include <cmath>
12 13 #include <cstdlib>
13 14 #include <fstream>
14
15
15 16 //#include "rand.h"
16 17 //#include "timer.h"
17
18
18 19 #define Treal      100          // time to simulate over
19 20 #define R0true     3.0           // infectiousness
20 21 #define rtrue      0.1           // recovery rate
21 22 #define retrue    0.05          // resusceptibility rate
22 23 #define Nreal      500.0        // population size
23 24 #define etatruem  0.5           // real drift attraction strength
24 25 #define berrtrue   0.5           // real beta drift noise
25 26 #define merr       5.0           // expected measurement error
26 27 #define I0         5.0           // Initial infected individuals
27
28
28 29 #define PSC       0.5           // scale factor for more sensitive
29
30
30 31 #include <Rcpp.h>
31 32 using namespace Rcpp;
32
33
34 34 struct State {
35 35     double S;
36 36     double I;
37 37     double R;
38 38 };
39
40 40 struct Particle {
41 41     double R0;
42 42     double r;
43 43     double re;
44 44     double sigma;
45 45     double eta;
46 46     double berr;
47 47     double B;
48 48     double S;
49 49     double I;
50 50     double R;
51 51     double Sinit;
```

```

1 52     double Iinit;
2 53     double Rinit;
3 54 };
4 55
5 56 struct ParticleInfo {
6 57     double R0mean;      double R0sd;
7 58     double rmean;       double rsd;
8 59     double remean;     double resd;
9 60     double sigmamean;  double sigmasd;
10 61    double etamean;    double etasd;
11 62    double berrmean;   double berrsd;
12 63    double Sinitmean; double Sinitsd;
13 64    double Iinitmean; double Iinitsd;
14 65    double Rinitmean; double Rinitsd;
15 66 };
16 67
17 68
18 69 int timeval_subtract (double *result, struct timeval *x, struct
19      timeval *y);
20 70 int check_double(double x,double y);
21 71 void exp_euler_SIRS(double h, double t0, double tn, int N, Particle *
22      particle);
23 72 void copyParticle(Particle * dst, Particle * src);
24 73 void perturbParticles(Particle * particles, int N, int NP, int
25      passnum, double coolrate);
26 74 void particleDiagnostics(ParticleInfo * partInfo, Particle *
27      particles, int NP);
28 75 void getStateMeans(State * state, Particle* particles, int NP);
29 76 NumericMatrix if2(NumericVector * data, int T, int N);
30 77 double randu();
31 78 double randn();
32 79
33 80 // [[Rcpp::export]]
34 81 Rcpp::List if2_sirs(NumericVector data, int T, int N, int NP, int
35      nPasses, double coolrate) {
36 82
37 83     int npar = 9;
38 84
39 85     NumericMatrix paramdata(NP, npar);
40 86     NumericMatrix means(nPasses, npar);
41 87     NumericMatrix sds(nPasses, npar);
42 88     NumericMatrix statemeans(T, 3);
43 89     NumericMatrix statedata(NP, 4);
44 90
45 91     srand(time(NULL));      // Seed PRNG with system time
46 92
47 93     double w[NP];          // particle weights
48 94
49 95     Particle particles[NP]; // particle estimates for current
50         step
51 96     Particle particles_old[NP]; // intermediate particle states for

```

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

```

1 146      // START PASSES THROUGH DATA
2 147
3 148      printf("Starting filter\n");
4 149      printf("-----\n");
5 150      printf("Pass\n");
6 151
7 152
8 153      for (int pass = 0; pass < nPasses; pass++) {
9 154
10 155      printf("...%d / %d\n", pass, nPasses);
11 156
12 157      // reset particle system evolution states
13 158      for (int n = 0; n < NP; n++) {
14 159
15 160          particles[n].S = particles[n].Sinit;
16 161          particles[n].I = particles[n].Iinit;
17 162          particles[n].R = particles[n].Rinit;
18 163          particles[n].B = (double) particles[n].R0 * particles[n].
19 164              r / N;
20 164
21 165      }
22 166
23 167      if (pass == (nPasses-1)) {
24 168          State sMeans;
25 169          getStateMeans(&sMeans, particles, NP);
26 170          statemeans(0,0) = sMeans.S;
27 171          statemeans(0,1) = sMeans.I;
28 172          statemeans(0,2) = sMeans.R;
29 173      }
30 174
31 175      for (int t = 1; t < T; t++) {
32 176
33 177          // generate individual predictions and weight
34 178          for (int n = 0; n < NP; n++) {
35 179
36 180              exp_euler_SIRS(1.0/7.0, (double) t-1, (double) t, N,
37 181                  &particles[n]);
38 181
39 182              double merr_par = particles[n].sigma;
40 183              double y_diff   = data[t] - particles[n].I;
41 184
42 185              w[n] = 1.0/(merr_par*sqrt(2.0*M_PI)) * exp( - y_diff*
43 186                  y_diff / (2.0*merr_par*merr_par) );
44 186
45 187      }
46 188
47 189      // cumulative sum
48 190      for (int n = 1; n < NP; n++) {
49 191          w[n] += w[n-1];
50 192      }
51 193

```

```

1 194          // save particle states to resample from
2 195          for (int n = 0; n < NP; n++){
3 196              copyParticle(&particles_old[n], &particles[n]);
4 197          }
5 198
6 199          // resampling
7 200          for (int n = 0; n < NP; n++) {
8 201
9 202              double w_r = randu() * w[NP-1];
10 203              int i = 0;
11 204              while (w_r > w[i]) {
12 205                  i++;
13 206              }
14 207
15 208              // i is now the index to copy state from
16 209              copyParticle(&particles[n], &particles_old[i]);
17 210
18 211          }
19 212
20 213          // between-iteration perturbations, not after last time
21 214          step
22 214          if (t < (T-1))
23 215              perturbParticles(particles, N, NP, pass, coolrate);
24 216
25 217          if (pass == (nPASSES-1)) {
26 218              State sMeans;
27 219              getStateMeans(&sMeans, particles, NP);
28 220              statemeans(t,0) = sMeans.S;
29 221              statemeans(t,1) = sMeans.I;
30 222              statemeans(t,2) = sMeans.R;
31 223          }
32 224
33 225      }
34 226
35 227      ParticleInfo pInfo;
36 228      particleDiagnostics(&pInfo, particles, NP);
37 229
38 230      means(pass, 0) = pInfo.R0mean;
39 231      means(pass, 1) = pInfo.rmean;
40 232      means(pass, 2) = pInfo.remean;
41 233      means(pass, 3) = pInfo.sigmaxmean;
42 234      means(pass, 4) = pInfo.etamean;
43 235      means(pass, 5) = pInfo.berrmean;
44 236      means(pass, 6) = pInfo.Sinitmean;
45 237      means(pass, 7) = pInfo.Iinitmean;
46 238      means(pass, 8) = pInfo.Rinitmean;
47 239
48 240      sds(pass, 0) = pInfo.R0sd;
49 241      sds(pass, 1) = pInfo.rsd;
50 242      sds(pass, 2) = pInfo.resd;
51 243      sds(pass, 3) = pInfo.sigmasd;

```

```

1 244     sds(pass, 4) = pInfo.etasd;
2 245     sds(pass, 5) = pInfo.berrsd;
3 246     sds(pass, 6) = pInfo.Sinitsd;
4 247     sds(pass, 7) = pInfo.Iinitsd;
5 248     sds(pass, 8) = pInfo.Rinitsd;
6 249
7 250     // between-pass perturbations, not after last pass
8 251     if (pass < (nPasses + 1))
9 252         perturbParticles(particles, N, NP, pass, coolrate);
10 253
11 254 }
12 255
13 256 ParticleInfo pInfo;
14 257 particleDiagnostics(&pInfo, particles, NP);
15 258
16 259     printf("Parameter results (mean | sd)\n");
17 260     printf("-----\n");
18 261     printf("R0      %f %f\n", pInfo.R0mean, pInfo.R0sd);
19 262     printf("r       %f %f\n", pInfo.rmean, pInfo.rsd);
20 263     printf("re      %f %f\n", pInfo.remean, pInfo.resd);
21 264     printf("sigma   %f %f\n", pInfo.sigmaxmean, pInfo.sigmasd);
22 265     printf("eta     %f %f\n", pInfo.etamean, pInfo.etasd);
23 266     printf("berr    %f %f\n", pInfo.berrmean, pInfo.berrsd);
24 267     printf("S_init  %f %f\n", pInfo.Sinitmean, pInfo.Sinitsd);
25 268     printf("I_init  %f %f\n", pInfo.Iinitmean, pInfo.Iinitsd);
26 269     printf("R_init  %f %f\n", pInfo.Rinitmean, pInfo.Rinitsd);
27 270
28 271     printf("\n");
29 272
30 273
31 274
32 275     // Get particle results to pass back to R
33 276
34 277     for (int n = 0; n < NP; n++) {
35 278
36 279         paramdata(n, 0) = particles[n].R0;
37 280         paramdata(n, 1) = particles[n].r;
38 281         paramdata(n, 2) = particles[n].re;
39 282         paramdata(n, 3) = particles[n].sigma;
40 283         paramdata(n, 4) = particles[n].eta;
41 284         paramdata(n, 5) = particles[n].berr;
42 285         paramdata(n, 6) = particles[n].Sinit;
43 286         paramdata(n, 7) = particles[n].Iinit;
44 287         paramdata(n, 8) = particles[n].Rinit;
45 288
46 289     }
47 290
48 291     for (int n = 0; n < NP; n++) {
49 292
50 293         statedata(n, 0) = particles[n].S;
51 294         statedata(n, 1) = particles[n].I;

```

```

1 295     statedata(n, 2) = particles[n].R;
2 296     statedata(n, 3) = particles[n].B;
3 297
4 298 }
5 299
6 300
7 301
8 302     return Rcpp::List::create( Rcpp::Named("paramdata") = paramdata,
9 303             Rcpp::Named("means") = means,
10 304             Rcpp::Named("statemeans") =
11                 statemeans,
12 305             Rcpp::Named("statedata") = statedata,
13 306             Rcpp::Named("sds") = sds);
14 307
15 308 }
16 309
17 310
18 311 /* Use the Explicit Euler integration scheme to integrate SIR model
19 forward in time
20 312     double h      - time step size
21 313     double t0     - start time
22 314     double tn     - stop time
23 315     double * y    - current system state; a three-component vector
24             representing [S I R], susceptible-infected-recovered
25 316
26 317 */
27 318 void exp_euler_SIRS(double h, double t0, double tn, int N, Particle *
28 particle) {
29 319
30 320     int num_steps = floor( (tn-t0) / h );
31 321
32 322     double S = particle->S;
33 323     double I = particle->I;
34 324     double R = particle->R;
35 325
36 326     double R0    = particle->R0;
37 327     double r     = particle->r;
38 328     double re    = particle->re;
39 329     double B0    = R0 * r / N;
40 330     double eta   = particle->eta;
41 331     double berr  = particle->berr;
42 332
43 333     double B = particle->B;
44 334
45 335     for(int i = 0; i < num_steps; i++) {
46 336
47 337         //double Bfac = 0.5 - 0.95*cos( (2.0*M_PI/365)*(t0*num_steps+
48             i) )/2.0;
49 338         double Bfac = exp(2*cos((2*M_PI/365)*(t0*num_steps+i)) - 2);
50 339         B = exp( log(B) + eta*(log(B0) - log(B)) + berr*randn() );
51 340

```

```

1 341     double BSI = Bfac*B*S*I;
2 342     double rI = r*I;
3 343     double reR = re*R;
4 344
5 345     // get derivatives
6 346     double dS = - BSI + reR;
7 347     double dI = BSI - rI;
8 348     double dR = rI - reR;
9 349
10 350    // step forward by h
11 351    S += h*dS;
12 352    I += h*dI;
13 353    R += h*dR;
14 354
15 355 }
16 356
17 357     particle->S = S;
18 358     particle->I = I;
19 359     particle->R = R;
20 360     particle->B = B;
21 361
22 362 }
23 363
24 364
25 365 /* Particle perturbation function to be run between iterations and
26  passes
27 366 */
28 367
29 368 void perturbParticles(Particle * particles, int N, int NP, int
30  passnum, double coolrate) {
31 369
32 370     //double coolcoef = exp( - (double) passnum / coolrate );
33 371     double coolcoef = pow(coolrate, passnum);
34 372
35 373
36 374     double spreadR0      = coolcoef * R0true / 10.0;
37 375     double spreadr       = coolcoef * rtrue / 10.0;
38 376     double spreadre      = coolcoef * retrue / 10.0;
39 377     double spreadsigma   = coolcoef * merr / 10.0;
40 378     double spreadIinit   = coolcoef * I0 / 10.0;
41 379     double spreadeta     = coolcoef * etatru / 10.0;
42 380     double spreadberr   = coolcoef * berrtrue / 10.0;
43 381
44 382
45 383     double R0can, rcan, recan, sigmacan, Iinitcan, etacan, berrcan;
46 384
47 385     for (int n = 0; n < NP; n++) {
48 386
49 387         do {
50 388             R0can = particles[n].R0 + spreadR0*randn();
51 389             } while (R0can < 0);

```

```

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

```

```
1 441     dst->I      = src->I;
2 442     dst->R      = src->R;
3 443     dst->Sinit   = src->Sinit;
4 444     dst->Iinit   = src->Iinit;
5 445     dst->Rinit   = src->Rinit;
6 446
7 447 }
8 448
9 449 void particleDiagnostics(ParticleInfo * partInfo, Particle *
10    particles, int NP) {
11 450     double R0mean      = 0.0,
12 451         rmean        = 0.0,
13 452         remean       = 0.0,
14 453         sigmamean    = 0.0,
15 454         etamean       = 0.0,
16 455         berrmean     = 0.0,
17 456         Sinitmean    = 0.0,
18 457         Iinitmean    = 0.0,
19 458         Rinitmean    = 0.0;
20 459
21 460     // means
22 461
23 462     for (int n = 0; n < NP; n++) {
24 463
25 464         R0mean      += particles[n].R0;
26 465         rmean        += particles[n].r;
27 466         remean       += particles[n].re;
28 467         etamean      += particles[n].eta,
29 468         berrmean     += particles[n].berr,
30 469         sigmamean    += particles[n].sigma;
31 470         Sinitmean    += particles[n].Sinit;
32 471         Iinitmean    += particles[n].Iinit;
33 472         Rinitmean    += particles[n].Rinit;
34 473
35 474
36 475     }
37 476
38 477     R0mean      /= NP;
39 478     rmean        /= NP;
40 479     remean       /= NP;
41 480     sigmamean    /= NP;
42 481     etamean      /= NP;
43 482     berrmean     /= NP;
44 483     Sinitmean    /= NP;
45 484     Iinitmean    /= NP;
46 485     Rinitmean    /= NP;
47 486
48 487     // standard deviations
49 488
50 489     double R0sd      = 0.0,
51 490         rsd        = 0.0,
```

```

1 491      resd    = 0.0,
2 492      sigmasd = 0.0,
3 493      etasd   = 0.0,
4 494      berrsd  = 0.0,
5 495      Sinitsd = 0.0,
6 496      Iinitsd = 0.0,
7 497      Rinitsd = 0.0;
8 498
9 499      for (int n = 0; n < NP; n++) {
10 500
11 501          R0sd    += ( particles[n].R0 - R0mean ) * ( particles[n].R0 -
12 502              R0mean );
13 503          rsd     += ( particles[n].r - rmean ) * ( particles[n].r -
14 504              rmean );
15 505          resd    += ( particles[n].re - rmean ) * ( particles[n].re -
16 506              rmean );
17 507          sigmasd += ( particles[n].sigma - sigmamean ) * ( particles[n].
18 508              sigma - sigmamean );
19 509          etasd   += ( particles[n].eta - etamean ) * ( particles[n].
20 510              eta - etamean );
21 511          berrsd  += ( particles[n].berr - berrmean ) * ( particles[n].
22 512              berr - berrmean );
23 513          Sinitsd += ( particles[n].Sinit - Sinitmean ) * ( particles[n].
24 514              Sinit - Sinitmean );
25 515          Iinitsd += ( particles[n].Iinit - Iinitmean ) * ( particles[n].
26 516              Iinit - Iinitmean );
27 517          Rinitsd += ( particles[n].Rinit - Rinitmean ) * ( particles[n].
28 518              Rinit - Rinitmean );
29 519
30 520      }
31 521
32 522      R0sd      /= NP;
33 523      rsd       /= NP;
34 524      resd     /= NP;
35 525      sigmasd  /= NP;
36 526      etasd    /= NP;
37 527      berrsd   /= NP;
38 528      Sinitsd  /= NP;
39 529      Iinitsd  /= NP;
40 530      Rinitsd  /= NP;
41 531
42 532      partInfo->R0mean    = R0mean;
43 533      partInfo->R0sd     = R0sd;
44 534      partInfo->rmean    = rmean;
45 535      partInfo->rsd     = rsd;
46 536      partInfo->remean   = remean;
47 537      partInfo->resd    = resd;
48 538      partInfo->sigmamean = sigmamean;
49 539      partInfo->sigmasd  = sigmasd;
50 540      partInfo->etamean   = etamean;
51 541      partInfo->etasd    = etasd;

```

```

1 533     partInfo->berrmean = berrmean;
2 534     partInfo->berrsdp = berrsdp;
3 535     partInfo->Sinitmean = Sinitmean;
4 536     partInfo->Sinitstd = Sinitstd;
5 537     partInfo->Iinitmean = Iinitmean;
6 538     partInfo->Iinitstd = Iinitstd;
7 539     partInfo->Rinitmean = Rinitmean;
8 540     partInfo->Rinitstd = Rinitstd;
9 541 }
10 542 }
11 543
12 544 double randu() {
13 545
14 546     return (double) rand() / (double) RAND_MAX;
15 547 }
16 548 }
17 549
18 550 void getStateMeans(State * state, Particle* particles, int NP) {
19 551
20 552     double Smean = 0, Imean = 0, Rmean = 0;
21 553
22 554     for (int n = 0; n < NP; n++) {
23 555         Smean += particles[n].S;
24 556         Imean += particles[n].I;
25 557         Rmean += particles[n].R;
26 558     }
27 559
28 560     state->S = (double) Smean / NP;
29 561     state->I = (double) Imean / NP;
30 562     state->R = (double) Rmean / NP;
31 563 }
32 564 }
33 565
34 566
35 567 /* Return a normally distributed random number with mean 0 and
36 standard deviation 1
37 568     Uses the polar form of the Box-Muller transformation
38 569     From http://www.design.caltech.edu/erik/Misc/Gaussian.html
39 570 */
40 571 double randn() {
41 572
42 573     double x1, x2, w, y1;
43 574
44 575     do {
45 576         x1 = 2.0 * randu() - 1.0;
46 577         x2 = 2.0 * randu() - 1.0;
47 578         w = x1 * x1 + x2 * x2;
48 579     } while ( w >= 1.0 );
49 580
50 581     w = sqrt( (-2.0 * log( w ) ) / w );
51 582     y1 = x1 * w;

```

```
1 583|
2 584    return y1;
3 585}
4 586|
```

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 ## ymat: Contains the initial conditions where:
7 2 # - rows are locations
8 3 # - columns are S, I, R
9 4 ## pars: Contains the parameters: global values for R0, r, N, eta,
10 berr
11 5 ## T: The stop time. Since 0 is included, there should be T+1
12 time steps in the simulation
13 6 ## neinum: Number of neighbors for each location, in order
14 7 ## neibmat: Contains lists of neighbors for each location
15 8 # - rows are parent locations (nodes)
16 9 # - columns are locations each parent is attached to (edges)
17 10 StocSSIR ← function(ymat, pars, T, steps, neinum, neibmat) {
18
19 12     ## number of locations
20 13     nloc ← dim(ymat)[1]
21
22 15     ## storage
23 16     ## dims are locations, (S,I,R,B), times
24 17     # output array
25 18     out ← array(NA, c(nloc, 4, T+1), dimnames = list(NULL, c("S","I",
26 "R","B"), NULL))
27 19     # temp storage
28 20     BSI ← numeric(nloc)
29 21     rI ← numeric(nloc)
30
31 23     ## extract parameters
32 24     R0 ← pars[['R0']]
33 25     r ← pars[['r']]
34 26     N ← pars[['N']]
```

```

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

```

```

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

```

19 F.2 RStan Spatial SIR Code

20 This code implements a Spatial SIR model in Rstan.

```

21
22 1 data {
23
24 3     int      <lower=1> T;      // total integration steps
25 4     int      <lower=1> nloc;    // number of locations
26 5     real     y[nloc, T];    // observed number of cases
27 6     int      <lower=1> N;      // population size
28 7     real     h;          // step size
29 8     int      <lower=0> neignum[nloc]; // number of neighbors
30 9     each location has
31 10    int           neibmat[nloc, nloc]; // neighbor list for
32 11    each location
33
34 11}
35
36 13 parameters {
37
38 15    real <lower=0, upper=10> R0;      // R0
39 16    real <lower=0, upper=10> r;        // recovery rate
40 17    real <lower=0, upper=20> sigma;    // observation error
41 18    real <lower=0, upper=30> Iinit[nloc]; // initial
42 19    infected for each location
43 20    real <lower=0, upper=1> eta;      // geometric walk
44 21    attraction strength
45 20    real <lower=0, upper=1> berr;     // beta walk noise
46 21    real <lower=-1.5, upper=1.5> Bnoise[nloc,T]; // Beta vector

```

```

1 22    real <lower=0, upper=1>           phi;      // interconnectivity
2     strength
3
4 24 }
5
6 26 model {
7
8 28     real S[nloc, T];
9     real I[nloc, T];
10 30    real R[nloc, T];
11 31    real B[nloc, T];
12 32    real B0;
13
14 34    real BSI[nloc, T];
15 35    real rI[nloc, T];
16 36    int n;
17 37    real sphi;
18 38    real ophi;
19 39    real nBIsum;
20
21 41    B0 ← R0 * r / N;
22
23 43    for (loc in 1:nloc) {
24 44        S[loc, 1] ← N - Iinit[loc];
25 45        I[loc, 1] ← Iinit[loc];
26 46        R[loc, 1] ← 0.0;
27 47        B[loc, 1] ← B0;
28 48    }
29
30 50    for (t in 2:T) {
31 51        for (loc in 1:nloc) {
32
33 53            Bnoise[loc, t] ~ normal(0,berr);
34 54            B[loc, t] ← exp( log(B[loc, t-1]) + eta * ( log(B0) - log
35               (B[loc, t-1]) ) + Bnoise[loc, t] );
36
37 56            n ← neinum[loc];
38 57            sphi ← 1.0 - phi*( n/(n+1.0) );
39 58            ophi ← phi/(n+1.0);
40
41 60            nBIsum ← 0.0;
42 61            for (j in 1:n)
43 62                nBIsum ← nBIsum + B[neibmat[loc, j], t-1] * I[neibmat
44               [loc, j], t-1];
45
46 64            BSI[loc, t] ← S[loc, t-1]*( sphi*B[loc, t-1]*I[loc, t-1]
47               + ophi*nBIsum );
48 65            rI[loc, t] ← r*I[loc, t-1];
49
50 67            S[loc, t] ← S[loc, t-1] + h*( - BSI[loc, t] );
51 68            I[loc, t] ← I[loc, t-1] + h*( BSI[loc, t] - rI[loc, t] );

```

```

1  69      R[loc, t] ← R[loc, t-1] + h*( rI[loc, t] );
2  70
3  71      if (y[loc, t] > 0) {
4  72          y[loc, t] ~ normal( I[loc, t], sigma );
5  73      }
6  74
7  75  }
8  76  }
9  77
10 78  R0 ~ lognormal(1,1);
11 79  r ~ lognormal(1,1);
12 80  sigma ~ lognormal(1,1);
13 81  for (loc in 1:nloc) {
14 82      Iinit[loc] ~ normal(y[loc, 1], sigma);
15 83  }
16 84
17 85 }
```

19 F.3 IF2 Spatial SIR Code

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

```

21
22 1/* Author: Dexter Barrows
23 2 Github: dbarrows.github.io
24 3
25 4 */
26 5
27 6 #include <stdio.h>
28 7 #include <math.h>
29 8 #include <sys/time.h>
30 9 #include <time.h>
31 10 #include <stdlib.h>
32 11 #include <string>
33 12 #include <cmath>
34 13 #include <cstdlib>
35 14 #include <fstream>
36 15
37 16 //#include "rand.h"
38 17 //#include "timer.h"
39 18
40 19 #define Treal 100           // time to simulate over
41 20 #define R0true 3.0          // infectiousness
42 21 #define rtrue 0.1           // recovery rate
43 22 #define Nreal 500.0         // population size
44 23 #define etatru 0.5          // real drift attraction strength
45 24 #define berrtru 0.5          // real beta drift noise
46 25 #define phitru 0.5           // real connectivity strength
47 26 #define merr 10.0            // expected measurement error
```

```

1 27 #define I0          5.0           // Initial infected individuals
2 28
3 29 #define PSC         0.5           // perturbation scale factor for more
4      sensitive parameters
5 30
6 31 #include <Rcpp.h>
7 32 using namespace Rcpp;
8 33
9 34 struct Particle {
10 35     double R0;
11 36     double r;
12 37     double sigma;
13 38     double eta;
14 39     double berr;
15 40     double phi;
16 41     double * S;
17 42     double * I;
18 43     double * R;
19 44     double * B;
20 45     double * Iinit;
21 46 };
22 47
23 48
24 49 int timeval_subtract (double *result, struct timeval *x, struct
25      timeval *y);
26 50 int check_double(double x,double y);
27 51 void initializeParticles(Particle ** particles, int NP, int nloc, int
28      N);
29 52 void exp_euler_SSIR(double h, double t0, double tn, int N, Particle *
30      particle,
31      NumericVector neinum, NumericMatrix neibmat, int
32          nloc) ;
33 54 void copyParticle(Particle * dst, Particle * src, int nloc);
34 55 void perturbParticles(Particle * particles, int N, int NP, int nloc,
35      int passnum, double coolrate);
36 56 double randu();
37 57 double randn();
38 58
39 59 // [[Rcpp::export]]
40 60 Rcpp::List if2_spa(NumericMatrix data, int T, int N, int NP, int
41      nPasses, double coolrate, NumericVector neinum, NumericMatrix
42      neibmat, int nloc) {
43 61
44 62     NumericMatrix paramdata(NP, 6);      // for R0, r, sigma, eta,
45         berr, phi
46 63     NumericMatrix initInfec(nloc, NP);   // for Iinit
47 64     NumericMatrix infecmeans(nloc, T);   // mean infection counts for
48         each location
49 65     NumericMatrix finalstate(nloc, 4);   // SIRB means for each
50         location
51 66

```

```

1  67    srand(time(NULL));           // Seed PRNG with system time
2  68
3  69    double w[NP];             // particle weights
4  70
5  71    // initialize particles
6  72    printf("Initializing particle states\n");
7  73    Particle * particles = NULL;        // particle estimates for
8      current step
9  74    Particle * particles_old = NULL;    // intermediate particle
10     states for resampling
11 75    initializeParticles(&particles, NP, nloc, N);
12 76    initializeParticles(&particles_old, NP, nloc, N);
13 77
14 78    /*
15 79    // copy particle test
16 80    copyParticle(&particles[0], &particles_old[0], nloc);
17 81
18 82    // perturb particle test
19 83    perturbParticles(particles, N, NP, nloc, 1, coolrate);
20 84
21 85    // evolution test
22 86    // reset particle system evolution states
23 87    for (int n = 0; n < NP; n++) {
24 88        for (int loc = 0; loc < nloc; loc++) {
25 89            particles[n].S[loc] = N - particles[n].Iinit[loc];
26 90            particles[n].I[loc] = particles[n].Iinit[loc];
27 91            particles[n].R[loc] = 0.0;
28 92            particles[n].B[loc] = (double) particles[n].R0 *
29                  particles[n].r / N;
30 93        }
31 94    }
32 95    printf("Before S:%f | I:%f | R:%f\n", particles[0].S[0],
33      particles[0].I[0], particles[0].R[0]);
34 96    exp_euler_SSIR(1.0/7.0, 0.0, 1.0, N, &particles[0], neinum,
35      neibmat, nloc);
36 97    printf("After S:%f | I:%f | R:%f\n", particles[0].S[0], particles
37      [0].I[0], particles[0].R[0]);
38 98    */
39 99
40 100   // START PASSES THROUGH DATA
41 101
42 102   printf("Starting filter\n");
43 103   printf("-----\n");
44 104   printf("Pass\n");
45 105
46 106
47 107   for (int pass = 0; pass < nPasses; pass++) {
48 108
49 109       printf("...%d / %d\n", pass, nPasses);
50 110
51 111       // reset particle system evolution states

```

```

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

```

```

1 159          // resampling
2 160          for (int n = 0; n < NP; n++) {
3 161
4 162              double w_r = randu() * w[NP-1];
5 163              int i = 0;
6 164              while (w_r > w[i]) {
7 165                  i++;
8 166              }
9 167
10 168          // i is now the index to copy state from
11 169          copyParticle(&particles[n], &particles_old[i], nloc);
12 170
13 171      }
14 172
15 173          // between-iteration perturbations, not after last time
16 174          step
17 174          if (t < (T-1))
18 175              perturbParticles(particles, N, NP, nloc, pass,
19 175                  coolrate);
20 176
21 177          if (pass == (nPASSES-1)) {
22 178              double means[nloc];
23 179              for (int loc = 0; loc < nloc; loc++) {
24 180                  means[loc] = 0.0;
25 181                  for (int n = 0; n < NP; n++) {
26 182                      means[loc] += particles[n].I[loc] / NP;
27 183                  }
28 184                  infecmeans(loc, t) = means[loc];
29 185              }
30 186          }
31 187
32 188      }
33 189
34 190          // between-pass perturbations, not after last pass
35 191          if (pass < (nPASSES + 1))
36 192              perturbParticles(particles, N, NP, nloc, pass, coolrate);
37 193
38 194      }
39 195
40 196          // pack parameter data (minus initial conditions)
41 197          for (int n = 0; n < NP; n++) {
42 198              paramdata(n, 0) = particles[n].R0;
43 199              paramdata(n, 1) = particles[n].r;
44 200              paramdata(n, 2) = particles[n].sigma;
45 201              paramdata(n, 3) = particles[n].eta;
46 202              paramdata(n, 4) = particles[n].berr;
47 203              paramdata(n, 5) = particles[n].phi;
48 204          }
49 205
50 206          // Pack initial condition data
51 207          for (int n = 0; n < NP; n++) {

```

```

1 208      for (int loc = 0; loc < nloc; loc++) {
2 209          initInfec(loc, n) = particles[n].Iinit[loc];
3 210      }
4 211  }
5 212
6 213 // Pack final state means data
7 214 double Smeans[nloc], Imeans[nloc], Rmeans[nloc], Bmeans[nloc];
8 215 for (int loc = 0; loc < nloc; loc++) {
9 216     Smeans[loc] = 0.0;
10 217     Imeans[loc] = 0.0;
11 218     Rmeans[loc] = 0.0;
12 219     Bmeans[loc] = 0.0;
13 220     for (int n = 0; n < NP; n++) {
14 221         Smeans[loc] += particles[n].S[loc] / NP;
15 222         Imeans[loc] += particles[n].I[loc] / NP;
16 223         Rmeans[loc] += particles[n].R[loc] / NP;
17 224         Bmeans[loc] += particles[n].B[loc] / NP;
18 225     }
19 226     finalstate(loc, 0) = Smeans[loc];
20 227     finalstate(loc, 1) = Imeans[loc];
21 228     finalstate(loc, 2) = Rmeans[loc];
22 229     finalstate(loc, 3) = Bmeans[loc];
23 230 }
24 231
25 232
26 233 return Rcpp::List::create( Rcpp::Named("paramdata") = paramdata,
27 234 Rcpp::Named("initInfec") = initInfec,
28 235 Rcpp::Named("infecmeans") =
29 236         infecmeans,
30 237         Rcpp::Named("finalstate") =
31 238             finalstate);
32 239
33 239
34 239
35 240 }
36 241
37 242
38 243 /* Use the Explicit Euler integration scheme to integrate SIR model
39 forward in time
40 244 double h      - time step size
41 245 double t0     - start time
42 246 double tn     - stop time
43 247 double * y    - current system state; a three-component vector
44 248     representing [S I R], susceptible-infected-recovered
45 248
46 249 */
47 250 void exp_euler_SSIR(double h, double t0, double tn, int N, Particle *
48 251     particle,
49 251         NumericVector neinum, NumericMatrix neibmat, int
50 251         nloc) {
51 252

```

```

1 253     int num_steps = floor( (tn-t0) / h );
2 254
3 255     double * S = particle->S;
4 256     double * I = particle->I;
5 257     double * R = particle->R;
6 258     double * B = particle->B;
7 259
8 260     // create last state vectors
9 261     double S_last[nloc];
10 262    double I_last[nloc];
11 263    double R_last[nloc];
12 264    double B_last[nloc];
13 265
14 266    double R0    = particle->R0;
15 267    double r     = particle->r;
16 268    double B0    = R0 * r / N;
17 269    double eta   = particle->eta;
18 270    double berr  = particle->berr;
19 271    double phi   = particle->phi;
20 272
21 273    //printf("sphi \t\t| ophi \t\t| BSI \t\t| rI \t\t| dS \t\t| dI \t
22 274      \t| dR \t\t| S \t\t| I \t\t| R |\n");
23 275
24 276    for(int t = 0; t < num_steps; t++) {
25 277
26 277        for (int loc = 0; loc < nloc; loc++) {
27 278            S_last[loc] = S[loc];
28 279            I_last[loc] = I[loc];
29 280            R_last[loc] = R[loc];
30 281            B_last[loc] = B[loc];
31 282        }
32 283
33 284        for (int loc = 0; loc < nloc; loc++) {
34 285
35 286            B[loc] = exp( log(B_last[loc]) + eta*(log(B0) - log(
36 287                B_last[loc])) + berr*randn() );
37 288
38 288            int n = neinum[loc];
39 289            double sphi = 1.0 - phi*( (double) n/(n+1.0) );
40 290            double ophi = phi/(n+1.0);
41 291
42 292            double nBIsum = 0.0;
43 293            for (int j = 0; j < n; j++)
44 294                nBIsum += B_last[(int) neibmat(loc, j) - 1] * I_last
45 295                  [(int) neibmat(loc, j) - 1];
46 296
47 296            double BSI = S_last[loc]*( sphi*B_last[loc]*I_last[loc] +
48 297              ophi*nBIsum );
49 297            double rI  = r*I_last[loc];
50 298
51 299            // get derivatives

```

```

1 300      double dS = - BSI;
2 301      double dI = BSI - rI;
3 302      double dR = rI;
4 303
5 304      // step forward by h
6 305      S[loc] += h*dS;
7 306      I[loc] += h*dI;
8 307      R[loc] += h*dR;
9 308
10 309      //if (loc == 1)
11 310      //  printf("%f\t|%f\t|%f\t|%f\t|%f\t|%f\t|%f\t|%f\t|\n",
12 311          //    sphi, ophi, BSI, rI, dS, dI, dR, S[1], I
13 312          //    [1], R[1]);
14 313
15 314 }
16 315
17 316 /*particle->S = S;
18 317 particle->I = I;
19 318 particle->R = R;
20 319 particle->B = B;*/
21 320
22 321 }
23 322
24 323 /* Initializes particles
25 324 */
26 325 void initializeParticles(Particle ** particles, int NP, int nloc, int
27 326 N) {
28 327
29 328     // allocate space for doubles
30 329     *particles = (Particle*) malloc (NP*sizeof(Particle));
31 330
32 331     // allocate space for arrays inside particles
33 332     for (int n = 0; n < NP; n++) {
34 333         (*particles)[n].S = (double*) malloc(nloc*sizeof(double));
35 334         (*particles)[n].I = (double*) malloc(nloc*sizeof(double));
36 335         (*particles)[n].R = (double*) malloc(nloc*sizeof(double));
37 336         (*particles)[n].B = (double*) malloc(nloc*sizeof(double));
38 337         (*particles)[n].Iinit = (double*) malloc(nloc*sizeof(double))
39 338             ;
40 339     }
41 340
42 341     // initialize all all parameters
43 342     for (int n = 0; n < NP; n++) {
44 343
45 344         double R0can, rcan, sigmacan, Iinitcan, etacan, berrcan,
46 345             phican;
47 346
48 347         do {
49 348             R0can = R0true + R0true*randn();
50 349
51 350

```

```

1 346      } while (R0can < 0);
2 347      (*particles)[n].R0 = R0can;
3 348
4 349      do {
5 350          rcan = rtrue + rtrue*randn();
6 351      } while (rcan < 0);
7 352      (*particles)[n].r = rcan;
8 353
9 354      for (int loc = 0; loc < nloc; loc++)
10 355          (*particles)[n].B[loc] = (double) R0can * rcan / N;
11 356
12 357      do {
13 358          sigmacan = merr + merr*randn();
14 359      } while (sigmacan < 0);
15 360      (*particles)[n].sigma = sigmacan;
16 361
17 362      do {
18 363          etacan = etatru + PSC*etatru*randn();
19 364      } while (etacan < 0 || etacan > 1);
20 365      (*particles)[n].eta = etacan;
21 366
22 367      do {
23 368          berrcan = berrtru + PSC*berrtru*randn();
24 369      } while (berrcan < 0);
25 370      (*particles)[n].berr = berrcan;
26 371
27 372      do {
28 373          phican = phitru + PSC*phitru*randn();
29 374      } while (phican <= 0 || phican >= 1);
30 375      (*particles)[n].phi = phican;
31 376
32 377      for (int loc = 0; loc < nloc; loc++) {
33 378          do {
34 379              Iinitcan = I0 + I0*randn();
35 380          } while (Iinitcan < 0 || N < Iinitcan);
36 381          (*particles)[n].Iinit[loc] = Iinitcan;
37 382      }
38 383
39 384  }
40 385
41 386  }
42 387
43 388 /* Particle perturbation function to be run between iterations and
44  passes
45 389 */
46 390
47 391 void perturbParticles(Particle * particles, int N, int NP, int nloc,
48  int passnum, double coolrate) {
49 392
50 393     //double coolcoef = exp( - (double) passnum / coolrate );
51 394     double coolcoef = pow(coolrate, passnum);

```

```

1 395      double spreadR0      = coolcoef * R0true / 10.0;
2 396      double spreadr       = coolcoef * rtrue / 10.0;
3 397      double spreadsigma   = coolcoef * merr / 10.0;
4 398      double spreadIinit   = coolcoef * I0 / 10.0;
5 399      double spreadeta     = coolcoef * etatru / 10.0;
6 400      double spreadberr   = coolcoef * berrtrue / 10.0;
7 401      double spreadphi     = coolcoef * phitru / 10.0;
8 402
9 403
10 404      double R0can, rcan, sigmacan, Iinitcan, etacan, berrcan, phican;
11 405
12 406      for (int n = 0; n < NP; n++) {
13 407
14 408          do {
15 409              R0can = particles[n].R0 + spreadR0*randn();
16 410          } while (R0can < 0);
17 411          particles[n].R0 = R0can;
18 412
19 413          do {
20 414              rcan = particles[n].r + spreadr*randn();
21 415          } while (rcan < 0);
22 416          particles[n].r = rcan;
23 417
24 418          do {
25 419              sigmacan = particles[n].sigma + spreadsigma*randn();
26 420          } while (sigmacan < 0);
27 421          particles[n].sigma = sigmacan;
28 422
29 423          do {
30 424              etacan = particles[n].eta + PSC*spreadeta*randn();
31 425          } while (etacan < 0 || etacan > 1);
32 426          particles[n].eta = etacan;
33 427
34 428          do {
35 429              berrcan = particles[n].berr + PSC*spreadberr*randn();
36 430          } while (berrcan < 0);
37 431          particles[n].berr = berrcan;
38 432
39 433          do {
40 434              phican = particles[n].phi + PSC*spreadphi*randn();
41 435          } while (phican <= 0 || phican >= 1);
42 436          particles[n].phi = phican;
43 437
44 438          for (int loc = 0; loc < nloc; loc++) {
45 439              do {
46 440                  Iinitcan = particles[n].Iinit[loc] + spreadIinit*
47 441                      randn();
48 441              } while (Iinitcan < 0 || Iinitcan > 500);
49 442              particles[n].Iinit[loc] = Iinitcan;
50 443
51 444      }

```

```
1 445 }
2 446 }
3 447
4 448 /* Convenience function for particle resampling process
5 449 */
6 450 void copyParticle(Particle * dst, Particle * src, int nloc) {
7 451
8 452     dst->R0      = src->R0;
9 453     dst->r       = src->r;
10 454    dst->sigma   = src->sigma;
11 455    dst->eta     = src->eta;
12 456    dst->berr    = src->berr;
13 457    dst->phi     = src->phi;
14 458
15 459    for (int n = 0; n < nloc; n++) {
16 460        dst->S[n]     = src->S[n];
17 461        dst->I[n]     = src->I[n];
18 462        dst->R[n]     = src->R[n];
19 463        dst->B[n]     = src->B[n];
20 464        dst->Iinit[n] = src->Iinit[n];
21 465    }
22 466
23 467 }
24 468
25 469
26 470
27 471 double randu() {
28 472
29 473     return (double) rand() / (double) RAND_MAX;
30 474 }
31 475 }
32 476
33 477 /*
34 478 void getStateMeans(State * state, Particle* particles, int NP) {
35 479
36 480     double Smean = 0, Imean = 0, Rmean = 0;
37 481
38 482     for (int n = 0; n < NP; n++) {
39 483         Smean += particles[n].S;
40 484         Imean += particles[n].I;
41 485         Rmean += particles[n].R;
42 486     }
43 487
44 488     state->S = (double) Smean / NP;
45 489     state->I = (double) Imean / NP;
46 490     state->R = (double) Rmean / NP;
47 491
48 492 }
49 493 */
50 494
51 495 /* Return a normally distributed random number with mean 0 and
```

```

1      standard deviation 1
2 496      Uses the polar form of the Box-Muller transformation
3 497      From http://www.design.caltech.edu/erik/Misc/Gaussian.html
4 498      */
5 499 double randn() {
6 500
7 501     double x1, x2, w, y1;
8 502
9 503     do {
10 504         x1 = 2.0 * randu() - 1.0;
11 505         x2 = 2.0 * randu() - 1.0;
12 506         w = x1 * x1 + x2 * x2;
13 507     } while ( w >= 1.0 );
14 508
15 509     w = sqrt( (-2.0 * log( w ) ) / w );
16 510     y1 = x1 * w;
17 511
18 512     return y1;
19 513
20 514 }
```

22 F.4 CUDA IF2 Spatial Fitting Code

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

```

26
27 1 /* Author: Dexter Barrows
28 2 Github: dbarrows.github.io
29 3 */
30 4
31 5 /* Runs a particle filter on synthetic noisy data and attempts to
32 6 reconstruct underlying true state at each time step. Note that
33 7 this program uses gnuplot to plot the data, so an x11
34 8 environment must be present. Also the multiplier of 1024 in the
35 9 definition of NP below should be set to a multiple of the number
36 10 of multiprocessors of your GPU for optimal results.
37 11
38 12 Also, the accompanying "pf.plg" file contains the instructions
39 13 gnuplot will use. It must be present in the same directory as
40 14 the executable generated by compiling this file.
41 15
42 16 Compile with:
43 17
44 18 nvcc -arch=sm_20 -O2 pf_cuda.cu timer.cpp rand.cpp -o pf_cuda.x
45 19
46 20 */
47 21
```

```

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

```

```

1 68     curandState randState; // PRNG state
2 69 } Particle;
3 70
4 71 __host__ std::string getHRmemsize (size_t memsize);
5 72 __host__ std::string getHRtime (float runtime);
6 73
7 74 __device__ void exp_euler_SSIR(float h, float t0, float tn, Particle
8      * particle, int * neinum, int * neibmat, int nloc);
9 75 __device__ void copyParticle(Particle * dst, Particle * src, int nloc
10     );
11 76
12 77
13 78 /* Initialize all PRNG states, get starting state vector using
14    initial distribution
15 */
16 80 __global__ void initializeParticles (Particle * particles, int nloc)
17  {
18 81     int id = blockIdx.x*blockDim.x + threadIdx.x; // global thread
19     ID
20
21 83
22 84     if (id < NP) {
23 85
24 86         // initialize PRNG state
25 87         curandState state;
26 88         curand_init(id, 0, 0, &state);
27 89
28 90         float R0can, rcan, sigmacan, Iinitcan, etacan, berrcan,
29             phican;
30 91
31 92         do {
32 93             R0can = R0true + R0true*curand_normal(&state);
33 94         } while (R0can < 0);
34 95         particles[id].R0 = R0can;
35 96
36 97         do {
37 98             rcan = rtrue + rtrue*curand_normal(&state);
38 99         } while (rcan < 0);
39 100        particles[id].r = rcan;
40 101
41 102        for (int loc = 0; loc < nloc; loc++)
42 103            particles[id].B[loc] = (float) R0can * rcan / N;
43 104
44 105        do {
45 106            sigmacan = merr + merr*curand_normal(&state);
46 107        } while (sigmacan < 0);
47 108        particles[id].sigma = sigmacan;
48 109
49 110        do {
50 111            etacan = etatrue + PSC*etatrue*curand_normal(&state);
51 112        } while (etacan < 0 || etacan > 1);

```

```

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

```

```

1 162      particles[id].sigma = merr;
2 163      particles[id].eta = etatrue;
3 164      particles[id].berr = berrtrue;
4 165      particles[id].phi = phitru;
5 166
6 167      for (int loc = 0; loc < nloc; loc++) {
7 168          particles[id].Iinit[loc] = I0;
8 169      }
9 170
10 171  }
11 172  }
12 173  }
13 174
14 175
15 176 /* Project particles forward, perturb, and save weight based on data
16 177 int t - time step number (1,...,T)
17 178 */
18 179 __global__ void project (Particle * particles, int * neinum, int *
19  neibmat, int nloc) {
20 180
21 181     int id = blockIdx.x*blockDim.x + threadIdx.x; // global id
22 182
23 183     if (id < NP) {
24 184         // project forward
25 185         exp_euler_SSIR(1.0/7.0, 0.0, 1.0, &particles[id], neinum,
26  neibmat, nloc);
27 186     }
28 187
29 188 }
30 189
31 190 __global__ void weight(float * data, Particle * particles, double * w
32  , int t, int T, int nloc) {
33 191
34 192     int id = blockIdx.x*blockDim.x + threadIdx.x; // global id
35 193
36 194     if (id < NP) {
37 195
38 196         float merr_par = particles[id].sigma;
39 197
40 198         // Get weight and save
41 199         double w_local = 1.0;
42 200         for (int loc = 0; loc < nloc; loc++) {
43 201             float y_diff = data[loc*T + t] - particles[id].I[loc];
44 202             w_local *= 1.0/(merr_par*sqrt(2.0*PI)) * exp( - y_diff*
45  y_diff / (2.0*merr_par*merr_par) );
46 203         }
47 204
48 205         w[id] = w_local;
49 206
50 207     }
51 208 }
```

```

1 209 }
2 210
3 211 __global__ void stashParticles (Particle * particles, Particle *
4 212     particles_old, int nloc) {
5 213
6 214     int id = blockIdx.x*blockDim.x + threadIdx.x; // global id
7 215
8 216     if (id < NP) {
9 217         // COPY PARTICLE
10 218         copyParticle(&particles_old[id], &particles[id], nloc);
11 219     }
12 219
13 220 }
14 221
15 222
16 223 /* The 0th thread will perform cumulative sum on the weights.
17 224 There may be a faster way to do this, will investigate.
18 225 */
19 226 __global__ void cumsumWeights (double * w) {
20 227
21 228     int id = blockIdx.x*blockDim.x + threadIdx.x; // global thread
22 229         ID
23 229
24 230     // compute cumulative weights
25 231     if (id == 0) {
26 232         for (int i = 1; i < NP; i++)
27 233             w[i] += w[i-1];
28 234     }
29 235
30 236 }
31 237
32 238
33 239 /* Resample from all particle states within cell
34 240 */
35 241 __global__ void resample (Particle * particles, Particle *
36 242     particles_old, double * w, int nloc) {
37 242
38 243     int id = blockIdx.x*blockDim.x + threadIdx.x;
39 244
40 245     if (id < NP) {
41 246
42 247         // resampling proportional to weights
43 248         double w_r = curand_uniform(&particles[id].randState) * w[NP
44 248             -1];
45 249         int i = 0;
46 250         while (w_r > w[i]) {
47 251             i++;
48 252         }
49 253
50 254         // i is now the index of the particle to copy from
51 255         copyParticle(&particles[id], &particles_old[i], nloc);

```

```
1 256      }
2 257  }
3 258 }
4 259 }
5 260
6 261 // launch this with probably just nloc threads... block structure/
7   size probably not important
8 262 __global__ void reduceStates (Particle * particles, float *
9   countmeans, int t, int T, int nloc) {
10 263     int id = blockIdx.x*blockDim.x + threadIdx.x;
11 264
12 265     if (id < nloc) {
13 266         int loc = id;
14 267
15 268         double countmean_local = 0.0;
16 269         for (int n = 0; n < NP; n++) {
17 270             countmean_local += particles[n].I[loc] / NP;
18 271         }
19 272
20 273         countmeans[loc*T + t] = (float) countmean_local;
21 274
22 275     }
23 276
24 277 }
25 278
26 279 }
27 280
28 281 __global__ void perturbParticles(Particle * particles, int nloc, int
29   passnum, double coolrate) {
30 282
31 283     //double coolcoef = exp( - (double) passnum / coolrate );
32 284     double coolcoef = pow(coolrate, passnum);
33 285
34 286     double spreadR0      = coolcoef * R0true / 10.0;
35 287     double spreadr       = coolcoef * rtrue / 10.0;
36 288     double spreadsigma   = coolcoef * merr / 10.0;
37 289     double spreadInit    = coolcoef * I0 / 10.0;
38 290     double spreadEta    = coolcoef * etatru / 10.0;
39 291     double spreadBerr   = coolcoef * berrtru / 10.0;
40 292     double spreadPhi    = coolcoef * phitru / 10.0;
41 293
42 294     double R0can, rcan, sigmacan, Iinitcan, etacan, berrcan, phican;
43 295
44 296     int id = blockIdx.x*blockDim.x + threadIdx.x;
45 297
46 298     if (id < NP) {
47 299
48 300         do {
49 301             R0can = particles[id].R0 + spreadR0*curand_normal(&
50               particles[id].randState);
51 302         } while (R0can < 0);
```

```

1 303     particles[id].R0 = R0can;
2 304
3 305     do {
4 306         rcan = particles[id].r + spreadr*curand_normal(&particles
5 307             [id].randState);
6 307     } while (rcan < 0);
7 308     particles[id].r = rcan;
8 309
9 310     do {
10 311         sigmacan = particles[id].sigma + spreadsigma*
11 311             curand_normal(&particles[id].randState);
12 312     } while (sigmacan < 0);
13 313     particles[id].sigma = sigmacan;
14 314
15 315     do {
16 316         etacan = particles[id].eta + PSC*spreadeta*curand_normal
17 316             (&particles[id].randState);
18 317     } while (etacan < 0 || etacan > 1);
19 318     particles[id].eta = etacan;
20 319
21 320     do {
22 321         berrcan = particles[id].berr + PSC*spreadberr*
23 321             curand_normal(&particles[id].randState);
24 322     } while (berrcan < 0);
25 323     particles[id].berr = berrcan;
26 324
27 325     do {
28 326         phican = particles[id].phi + PSC*spreadphi*curand_normal
29 326             (&particles[id].randState);
30 327     } while (phican <= 0 || phican >= 1);
31 328     particles[id].phi = phican;
32 329
33 330     for (int loc = 0; loc < nloc; loc++) {
34 331         do {
35 332             Iinitcan = particles[id].Iinit[loc] + spreadIinit*
36 332                 curand_normal(&particles[id].randState);
37 333         } while (Iinitcan < 0 || Iinitcan > 500);
38 334         particles[id].Iinit[loc] = Iinitcan;
39 335     }
40 336
41 337 }
42 338
43 339 }
44 340
45 341
46 342 int main (int argc, char *argv[]) {
47 343
48 344     int T, nloc;
49 345
50 346     double restime;
51 347

```

```
1 348     struct timeval tdr0, tdr1, tdrMaster;
2 349
3 350     // Parse arguments ****
4 351
5 352     if (argc < 4) {
6 353         std::cout << "Not enough arguments" << std::endl;
7 354         return 0;
8 355     }
9 356
10 357     std::string arg1(argv[1]); // infection counts
11 358     std::string arg2(argv[2]); // neighbour counts
12 359     std::string arg3(argv[3]); // neighbour indices
13 360     std::string arg4(argv[4]); // outfile: params + runtime
14 361
15 362     std::cout << "Arguments:" << std::endl;
16 363     std::cout << "Infection data: " << arg1 << std::endl;
17 364     std::cout << "Neighbour counts: " << arg2 << std::endl;
18 365     std::cout << "Neighbour indices: " << arg3 << std::endl;
19 366     std::cout << "Outfile " << arg4 << std::endl;
20 367
21 368     // ****
22 369
23 370
24 371     // Read count data ****
25 372
26 373     std::cout << "Getting count data" << std::endl;
27 374     float * data = getDataFloat(arg1, &T, &nloc);
28 375     size_t datasize = nloc*T*sizeof(float);
29 376
30 377     // ****
31 378
32 379     // Read neinum matrix data ****
33 380
34 381     std::cout << "Getting neighbour count data" << std::endl;
35 382     int * neinum = getDataInt(arg2, NULL, NULL);
36 383     size_t neinumsize = nloc * sizeof(int);
37 384
38 385     // ****
39 386
40 387     // Read neibmat matrix data ****
41 388
42 389     std::cout << "Getting neighbour count data" << std::endl;
43 390     int * neibmat = getDataInt(arg3, NULL, NULL);
44 391     size_t neibmatsize = nloc * nloc * sizeof(int);
45 392
46 393     // ****
47 394
48 395     //
49
50
51 396     ****
```

```

1 397      // start timing
2 398      gettimeofday (&tdr0, NULL);
3 399
4 400      // CUDA data ****
5 401
6 402      std::cout << "Allocating device storage" << std::endl;
7 403
8 404      float      * d_data;           // device copy of data
9 405      Particle   * particles;        // particles
10 406     Particle   * particles_old;    // intermediate particle states
11 407     double     * w;               // weights
12 408     int       * d_neinum;         // device copy of adjacency
13
14 409     matrix
15 410     int       * d_neibmat;        // device copy of neighbour
16 411     counts matrix
17 412     float      * countmeans;      // host copy of reduced infection
18 413     count means from last pass
19 414     float      * d_countmeans;    // device copy of reduced
20 415     infection count means from last pass
21 416
22 417     CUDA_CALL( cudaMalloc( (void**) &d_data           , datasize )
23 418             );
24 419     CUDA_CALL( cudaMalloc( (void**) &particles        , NP*sizeof(
25 420             Particle)) );
26 421     CUDA_CALL( cudaMalloc( (void**) &particles_old    , NP*sizeof(
27 422             Particle)) );
28 423     CUDA_CALL( cudaMalloc( (void**) &w                , NP*sizeof(
29 424             double)) );
30 425     CUDA_CALL( cudaMalloc( (void**) &d_neinum         , neinumsize)
31 426             );
32 427     CUDA_CALL( cudaMalloc( (void**) &d_neibmat        , neibmatsize)
33 428             );
34 429     CUDA_CALL( cudaMalloc( (void**) &d_countmeans     , nloc*T*sizeof(
35 430             float)) );
36 431
37 432     gettimeofday (&tdr1, NULL);
38 433     timeval_subtract (&restime, &tdr1, &tdr0);
39 434
40 435     std::cout << "\t" << getHRtime(restime) << std::endl;
41 436
42 437     size_t avail, total;
43 438     cudaMemGetInfo( &avail, &total );
44 439     size_t used = total - avail;
45 440
46 441     std::cout << "\t[" << getHRmemsize(used) << "] used of [" <<
47 442     getHRmemsize(total) << "]" << std::endl;
48 443
49 444     std::cout << "Copying data to device" << std::endl;
50 445
51 446     gettimeofday (&tdr0, NULL);

```

```

1 436
2 437     CUDA_CALL( cudaMemcpy(d_data      , data       , datasize      ,
3           cudaMemcpyHostToDevice) );
4 438     CUDA_CALL( cudaMemcpy(d_neinum   , neinum    , neinumsize   ,
5           cudaMemcpyHostToDevice) );
6 439     CUDA_CALL( cudaMemcpy(d_neibmat , neibmat  , neibmatsize ,
7           cudaMemcpyHostToDevice) );
8 440
9 441     gettimeofday (&tdr1, NULL);
10 442     timeval_subtract (&restime, &tdr1, &tdr0);
11 443
12 444     std::cout << "\t" << getHRtime(restime) << std::endl;
13 445
14 446 // ****
15 447
16 448
17 449
18 450 // Initialize particles ****
19 451
20 452     std::cout << "Initializing particles" << std::endl;
21 453
22 454 //gettimeofday (&tdr0, NULL);
23 455
24 456     int nThreads    = 32;
25 457     int nBlocks     = ceil( (float) NP / nThreads);
26 458
27 459     initializeParticles <<< nBlocks, nThreads >>> (particles, nloc);
28 460     CUDA_CALL( cudaGetLastError() );
29 461     CUDA_CALL( cudaDeviceSynchronize() );
30 462
31 463     initializeParticles <<< nBlocks, nThreads >>> (particles_old,
32           nloc);
33 464     CUDA_CALL( cudaGetLastError() );
34 465     CUDA_CALL( cudaDeviceSynchronize() );
35 466
36 467 //gettimeofday (&tdr1, NULL);
37 468 //timeval_subtract (&restime, &tdr1, &tdr0);
38 469 //std::cout << "\t" << getHRtime(restime) << std::endl;
39 470
40 471     cudaMemGetInfo( &avail, &total );
41 472     used = total - avail;
42 473     std::cout << "\t[" << getHRmemsize(used) << "] used of [" <<
43           getHRmemsize(total) << "]" << std::endl;
44 474
45 475 // ****
46 476
47 477 // Starting filtering ****
48 478
49 479     for (int pass = 0; pass < 50; pass++) {
50 480
51 481         //std::cout << "pass = " << pass << std::endl;

```

```

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

```

```

1  528          // gettimeofday (&tdr1 , NULL);
2  529          // timeval_subtract (&restime, &tdr1 , &tdr0);
3  530          // std::cout << "\tProjection " << getHRtime(restime) <<
4              std::endl;
5  531      //}
6  532
7  533      // Weighting
8          ****
9  534
10 535          nThreads     = 32;
11 536          nBlocks      = ceil( (float) NP / nThreads);
12 537
13 538          weight <<< nBlocks , nThreads >>>(d_data , particles , w , t ,
14              T , nloc);
15 539          CUDA_CALL( cudaGetLastError() );
16 540          CUDA_CALL( cudaDeviceSynchronize() );
17 541
18 542      // Cumulative sum
19          ****
20 543
21 544          nThreads     = 1;
22 545          nBlocks      = 1;
23 546
24 547          //if (t == 1)
25 548          //    gettimeofday (&tdr0 , NULL);
26 549
27 550          cumsumWeights <<< nBlocks , nThreads >>> (w);
28 551          CUDA_CALL( cudaGetLastError() );
29 552          CUDA_CALL( cudaDeviceSynchronize() );
30 553
31 554          //if (t == 1) {
32 555          //    gettimeofday (&tdr1 , NULL);
33 556          //    timeval_subtract (&restime, &tdr1 , &tdr0);
34 557          //    std::cout << "\tCumulative sum " << getHRtime(
35              restime) << std::endl;
36 558      //}
37 559
38 560      // Save particles for resampling from
39          ****
40 561
41 562          nThreads     = 32;
42 563          nBlocks      = ceil( (float) NP / nThreads);
43 564
44 565          stashParticles <<< nBlocks , nThreads >>> (particles ,
45              particles_old , nloc);
46 566          CUDA_CALL( cudaGetLastError() );
47 567          CUDA_CALL( cudaDeviceSynchronize() );
48 568
49 569
50 570      // Resampling
51          ****

```

```

1 571             nThreads      = 32;
2 572             nBlocks       = ceil( (float) NP/ nThreads);
3 573
4 574             //if (t == 1)
5 575             //  gettimeofday (&tdr0, NULL);
6 576
7 577             resample <<< nBlocks, nThreads >>> (particles,
8 578                 particles_old, w, nloc);
9 579             CUDA_CALL( cudaGetLastError() );
10 580             CUDA_CALL( cudaDeviceSynchronize() );
11 581
12 582             //if (t == 1) {
13 583             //  gettimeofday (&tdr1, NULL);
14 584             //  timeval_subtract (&restime, &tdr1, &tdr0);
15 585             //  std::cout << "\tResampling " << getHRTIME(restime) <<
16 586             //          std::endl;
17 587             //}
18 588
19 589             // Reduction
20 588             ****
21
22 589
23 590             //if (t == (Tlim-1)) {
24 591
25 592             if (pass == 49) {
26 593
27 594                 //if (t == 1)
28 595                 //  gettimeofday (&tdr0, NULL);
29 596
30 597                 nThreads = 1;
31 598                 nBlocks  = 10;
32 599
33 600                 reduceStates <<< nBlocks, nThreads >>> (particles,
34
35 601                     d_countmeans, t, T, nloc);
36 602                 CUDA_CALL( cudaGetLastError() );
37 603                 CUDA_CALL( cudaDeviceSynchronize() );
38 604
39 605                 //if (t == 1) {
40 606                 //  gettimeofday (&tdr1, NULL);
41 607                 //  timeval_subtract (&restime, &tdr1, &tdr0);
42 608                 //  std::cout << "Reduction " << getHRTIME(
43 609                     restime) << std::endl;
44 610                 //}
45 610
46 611
47 612             // Perturb particles
48 612             ****
49 613
50 614             nThreads      = 32;
51 615             nBlocks       = ceil( (float) NP/ nThreads);

```

```
1 616      perturbParticles <<< nBlocks, nThreads >>> (particles,
2 617          nloc, pass, 0.975);
3 618      CUDA_CALL( cudaGetLastError() );
4 619      CUDA_CALL( cudaDeviceSynchronize() );
5 620
6 621      } // end time
7 621
8 622      } // end pass
9 623
10 624      std::cout.precision(10);
11 625
12 626      countmeans = (float*) malloc (nloc*T*sizeof(float));
13 627      cudaMemcpy(countmeans, d_countmeans, nloc*T*sizeof(float),
14 628          cudaMemcpyDeviceToHost);
15 629
16 630      // stop master timer and print
17 631
18 632      gettimeofday (&tdrMaster, NULL);
19 632      timeval_subtract(&restime, &tdrMaster, &tdr0);
20 633      std::cout << "Time: " << getHRTIME(restime) << std::endl;
21 634      std::cout << "Rawtime: " << restime << std::endl;
22 635
23 636
24 637      // Write results out
25 638
26 639
27 640      std::string filename = arg4;
28 641
29 642      std::cout << "Writing results to file " << filename << ", ... "
30 643          << std::endl;
31 643
32 644      std::ofstream outfile;
33 645      outfile.open(filename.c_str());
34 646
35 647      for(int loc = 0; loc < nloc; loc++) {
36 648          for (int t = 0; t < T; t++) {
37 649              outfile << countmeans[loc*T + t] << " ";
38 650          }
39 651          outfile << std::endl;
40 652      }
41 653
42 654      outfile.close();
43 655
44 656      //gettimeofday (&tdr1, NULL);
45 657      //timeval_subtract (&restime, &tdr1, &tdrMaster);
46 658      //std::cout << "Total PF time (excluding setup) " << getHRTIME(
47 659          restime) << std::endl;
48 659
49 660      cudaFree(d_data);
50 661      cudaFree(particles);
51 662      cudaFree(particles_old);
```

```

1 663     cudaFree(w);
2 664     cudaFree(d_neinum);
3 665     cudaFree(d_neibmat);
4 666     cudaFree(d_countmeans);
5 667
6 668     exit (EXIT_SUCCESS);
7 669 }
8 670 }
9 671
10 672
11 673 /* Use the Explicit Euler integration scheme to integrate SIR model
12 forward in time
13 674     float h      - time step size
14 675     float t0     - start time
15 676     float tn      - stop time
16 677     float * y     - current system state; a three-component vector
17             representing [S I R], susceptible-infected-recovered
18 678 */
19 679 __device__ void exp_euler_SSIR(float h, float t0, float tn, Particle
20 * particle, int * neinum, int * neibmat, int nloc) {
21 680
22 681     int num_steps = floor( (tn-t0) / h );
23 682
24 683     float * S = particle->S;
25 684     float * I = particle->I;
26 685     float * R = particle->R;
27 686     float * B = particle->B;
28 687
29 688     // create last state vectors
30 689     float * S_last = (float*) malloc (nloc*sizeof(float));
31 690     float * I_last = (float*) malloc (nloc*sizeof(float));
32 691     float * R_last = (float*) malloc (nloc*sizeof(float));
33 692     float * B_last = (float*) malloc (nloc*sizeof(float));
34 693
35 694     float R0      = particle->R0;
36 695     float r       = particle->r;
37 696     float B0      = R0 * r / N;
38 697     float eta     = particle->eta;
39 698     float berr    = particle->berr;
40 699     float phi     = particle->phi;
41 700
42 701     for(int t = 0; t < num_steps; t++) {
43 702
44 703         for (int loc = 0; loc < nloc; loc++) {
45 704             S_last[loc] = S[loc];
46 705             I_last[loc] = I[loc];
47 706             R_last[loc] = R[loc];
48 707             B_last[loc] = B[loc];
49 708         }
50 709
51 710         for (int loc = 0; loc < nloc; loc++) {

```

```

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

```

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

```

1 808     ss << runtime/mt << " m";
2 809     else if (runtime > ht && runtime <= dt)
3 810         ss << runtime/dt << " h";
4 811     else
5 812         ss << runtime/ht << " d";
6 813
7 814     valstring = ss.str();
8 815
9 816     return valstring;
10 817
11 818 }

```

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

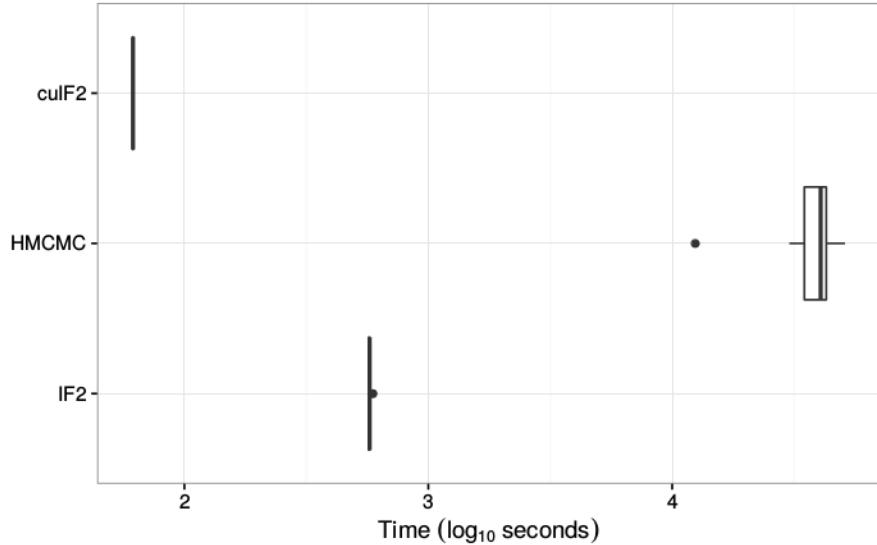


Figure F.1: Running times for fitting the spatial SIR model to data.

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