

ESTIMATION AND INFERENCE OF NONLINEAR
STOCHASTIC TIME SERIES

A COMPARISON OF TECHNIQUES FOR ESTIMATION AND INFERENCE OF
NONLINEAR STOCHASTIC TIME SERIES

By
DEXTER BARROWS, B.Sc.

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

McMaster University ©Copyright by Dexter Barrows, April 2016

MASTER OF SCIENCE (2015)
(Mathematics)

McMaster University
Hamilton, Ontario, Canada

TITLE

A Comparison of Techniques for Es-
timation and Inference of Nonlinear
Stochastic Time Series

AUTHOR

Dexter Barrows B.Sc.

B.Sc. (Honours)

Ryerson University, Toronto, Canada

SUPERVISOR

Dr. Benjamin Bolker

NUMBER OF PAGES

x, 141

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Acknowledgements

There are many people in my life that have provided support, love, thoughts, support, guidance, care, and a great many other things over the last two years.

I am especially thankful to my supervisor Dr. Ben Bolker for his mentorship, advice, direction, and most of all patience.

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	HMCMC Fitting	35
4.7	HMCMC Densities	35
4.8	HMCMC 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	68
A.1	Full R code	68
A.2	Full Stan code	71
B	Iterated Filtering	73
B.1	Full R code	73
B.2	Full C++ code	75
C	Parameter Fitting	86
D	Forecasting Frameworks	87
D.1	IF2 Parametric Bootstrapping Function	87
D.2	RStan Forward Simulator	89
E	S-map and SIRS	91
E.1	SIRS R Function Code	91
E.2	SMAP Code	92
E.3	SMAP Parameter Optimization Code	94
E.4	RStan SIRS Code	96
E.5	IF2 SIRS Code	97

F	Spatial Epidemics	111
F.1	Spatial SIR R Function Code	111
F.2	RStan Spatial SIR Code	113
F.3	IF2 Spatial SIR Code	115
F.4	CUDA IF2 Spatial Fitting Code	126

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. (<i>Andrieu et al., 2003</i>)	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 $\epsilon_{\text{obs}} = \mathcal{N}(0, 10)$, and the grey ribbon is centre 95th quantile from 100 random trajectories.	28
4.4	Fitting errors.	31
4.5	True system trajectory (solid line), observed data (dots), and IF2 estimated real state (dashed line).	31
4.6	The horizontal axis shows the IF2 pass number. The solid black lines show the evolution of the ML estimates, the solid grey lines show the true value, and the dashed grey lines show the mean parameter estimates from the particle swarm after the final pass.	33
4.7	The horizontal axis shows the IF2 pass number and the solid black lines show the evolution of the standard deviations of the particle swarm values.	33
4.8	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 HMC MC bootstrap trajectories. The solid line shows the true states, the dots show the data, the dotted line shows the average system behaviour, the dashed line shows the bootstrap mean, and the grey ribbon shows the centre 95th quantile of the bootstrap trajectories.	37
4.11	IF2 point estimate densities are shown in black and HMC MC point estimate densities are shown in grey. The vertical black lines show the true parameter values.	38
4.12	Fitting times for IF2 and HMC MC, 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 HMC MC / 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 observation noise drawn from $\mathcal{N}(0, 10)$	56
7.3	Average SSE (log scale) across each location and all trials as a function of the number of weeks ahead in the forecast.	58
7.4	Runtimes for producing spatial SIR forecasts. The box shows the middle 50th quantile, the bold line is the median, and the dots are outliers.	58

Chapter 1

Introduction

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

Despite the importance of reliable forecasts, obtaining them remains a challenge from both a theoretical and practical standpoint. Mathematical models can capture the essential drivers in disease dynamics, and extend them past the present into the future. However, different epidemics may present with varying dynamics and require different model parameters to be accurately represented. These parameters can be inferred by using statistical model fitting techniques, but this can become computationally intensive, and the modeller risks “overfitting” by attempting to capture too many drivers with too little data. Thus, The modeller must exercise restraint in model selection and fitting technique.

Securing precise, error-free data in the midst of an outbreak can be difficult if not impossible, thus uncertainty in what we observe in building mathematical models of disease spread must be accounted for from the get-go. Models must differentiate between natural variation in the intensity of disease spread (process error) and error in data collection (observation error) in order to accurately determine the dynamics underlying a data set, adding another layer of complexity. With these caveats and concerns acknowledged, we can turn to a discussion of technique.

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

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

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*.

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

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

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

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

1 without the need to refit the entire model. Normally, MCMC-based machinery must
2 refit the entire model whereas PF-based approaches can sometimes integrate data in
3 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. Additionally, these methods are
10 very-well-studied, and further work showing their viability would likely prove extra-
11 neous in the modern academic landscape.

12 Somewhat frustratingly, there exists no “gold standard” in forecasting. As method-
13 ology varies widely in theoretical justification, implementation, and operation, it is
14 difficult to compare the state of the art in forecasting methods from a first-principles
15 perspective. Further, published work making use of any of these methods to forecast
16 uses different prediction accuracy metrics, such as SSE, peak time/duration/intensity,
17 correlation tests, or RMSE, among others. Thus is is difficult to select the best tool
18 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 location are connected and disease is free to spread
24 from one to another.

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

39 For PF-based methods we have selected IF2 [*Ionides ref*], a very recently developed
40 approach that uses multiple particle filtering rounds to generate Maximum Likelihood

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

11 Finally, from the phenomenological methods we have selected the sequential locally
12 weighted global linear maps (S-map) [*Sugihara ref*], combined with Dewdrop Regres-
13 sion [*Sugihara ref 2*]. These methods stand on their own as a unique take on the
14 forecasting problem, and bear little resemblance to other methodology. The virtues
15 of these techniques have been long-extolled by their developers, but their efficacy when
16 compared to competing methods has not been well-studied. This work will mark one
17 of the first times this has been done.

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

Chapter 2

Hamiltonian MCMC

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

2.1 Markov Chains

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

The transition probabilities can be summarized as a matrix as

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

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

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

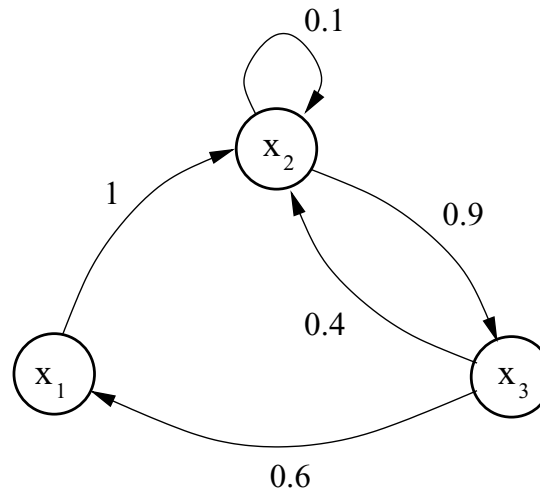


Figure 2.1: A finite state machine. States are shown as graph nodes, and the probability of transitioning from one particular state to another is shown as a weighted graph edge. (Andrieu et al., 2003)

- 1 This indicates that no matter what we pick for the initial probability distribution
- 2 $\mu(x^{(1)})$, the chain will always stabilize at the equilibrium distribution.
- 3 This property holds when the chain satisfies the following conditions
- 4 • *Irreducible* Any state A can be reached from any other state B with non-zero
- 5 probability
- 6 • *Positive Recurrent* The number of steps required for the chain to reach state A
- 7 from state B must be finite
- 8 • *Aperiodic* The chain must be able to explore the parameter space without be-
- 9 coming trapped in a cycle
- 10 Note that MCMC sampling generates a Markov chain $(\theta^{(1)}, \theta^{(2)}, \dots, \theta^{(N)})$ that does
- 11 indeed satisfy these conditions, and uses the chain's equilibrium distribution to ap-
- 12 proximate the posterior distribution of the parameter space.

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

1 order to do this we need a way to evaluate whether or not $M(\theta)$ is a good fit for D ;
 2 this is done by specifying a likelihood function $\mathcal{L}(\theta)$ such that

$$3 \qquad \qquad \qquad \mathcal{L}(\theta) \propto P(D|\theta). \qquad \qquad \qquad (2.3)$$

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

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 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.

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

stand a decent chance of also being accepted, but far enough away that it doesn't get "trapped" in a particular region of the parameter space.

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

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

2.5 Algorithm

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

We will denote the previously discussed quantities as

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

and then define the acceptance ratio, r , as

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

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

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

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

Algorithm [1] shows the Metropolis MCMC algorithm.

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

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 $M + N$.

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.

2.8 Hamiltonian Monte Carlo

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

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

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

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

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

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

The Hamiltonian of the system is defined as

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

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

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

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

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

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. However, some recent algo-
 17 rithms, such as the No U-Turn sampler implemented in RStan, and adaptively select
 18 appropriate values automatically during the sampling process.

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  $(\theta^{(1)}, \theta^{(2)}, \dots, \theta^{(N)})$ 

```

2.9 RStan Fitting

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

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

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

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

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

Figure [2.2] shows the system simulation results.

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

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

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

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

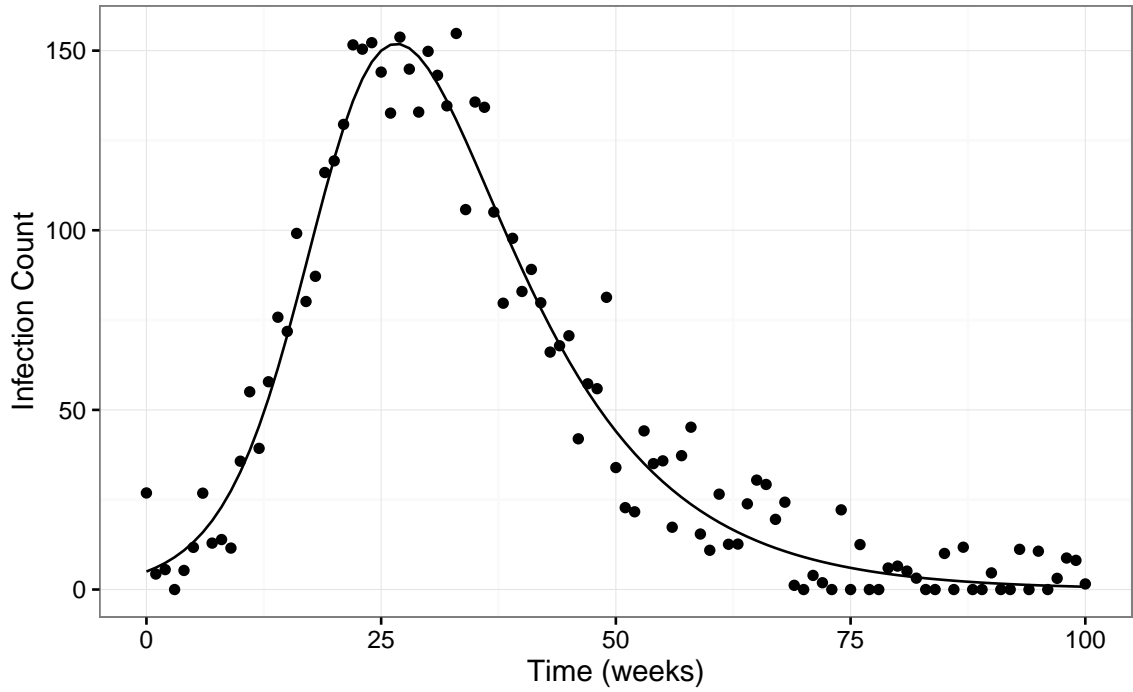


Figure 2.2: True SIR ODE solution infected counts, and with added observation noise.

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

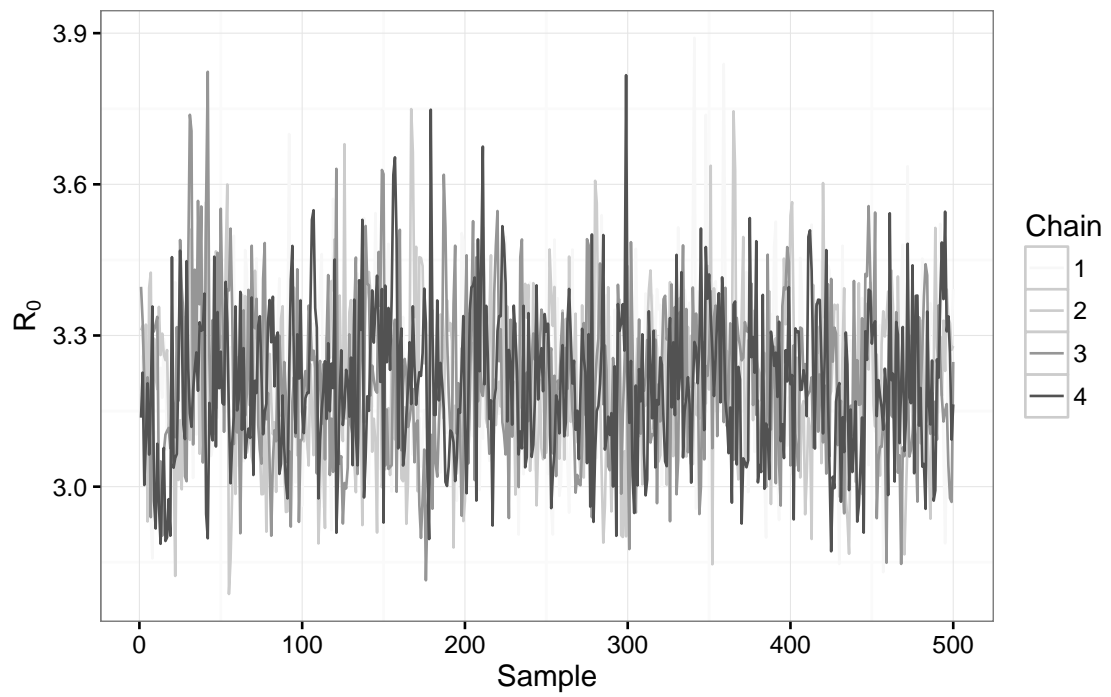


Figure 2.3: Traceplot of samples drawn for parameter R_0 , excluding warmup.

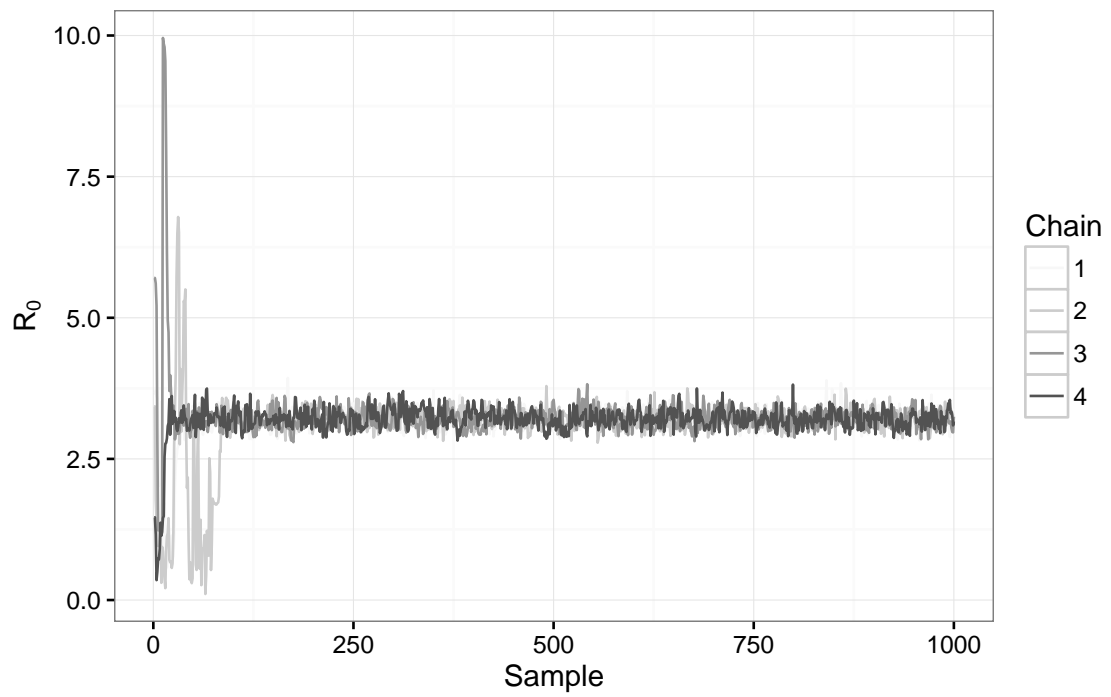


Figure 2.4: Traceplot of samples drawn for parameter R_0 , including warmup.

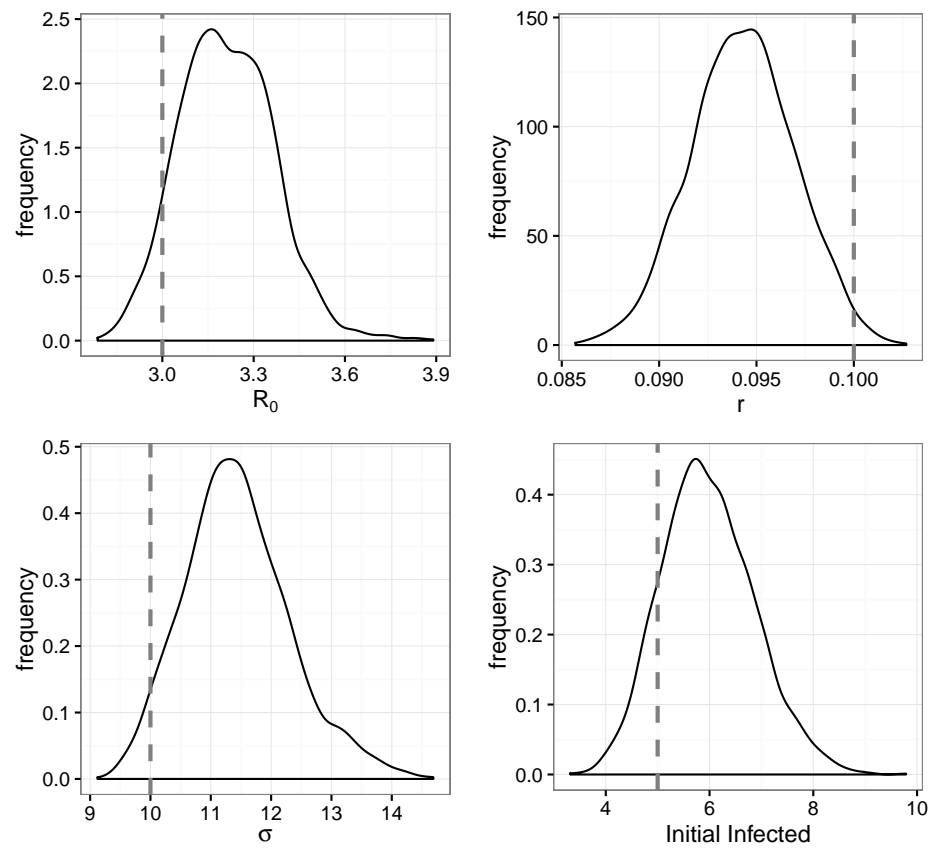


Figure 2.5: Kernel density estimates produced by Stan. Dashed lines show true parameter values.

Chapter 3

Iterated Filtering

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

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

3.1 Formulation

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

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

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

1 And the observation process by

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

3 Components of 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  $(\theta^{(1)}, \theta^{(2)}, \dots, \theta^{(J)})$ 

```

3.3 Particle Collapse

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

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

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

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

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

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

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

3.4 Iterated Filtering and Data Cloning

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

Rigorous proofs have been developed [Ionides ref.] that show that by treating the parameters as stochastic processes instead of fixed values, the multiple passes through

1 the data will indeed force convergence of the process mean toward maximum likeli-
 2 hood, and the process variance toward 0.

3 **3.5 Iterated Filtering 2 (IF2)**

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

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

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

25 The algorithm for IF2 can be seen in Algorithm [4].
 26

Algorithm 4: IF2

```

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

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

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

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

```

1 3.6 IF2 Fitting

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

4 As in the previous section, the model in Equation [1.1] was use 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 lees 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 HMCMC, 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.

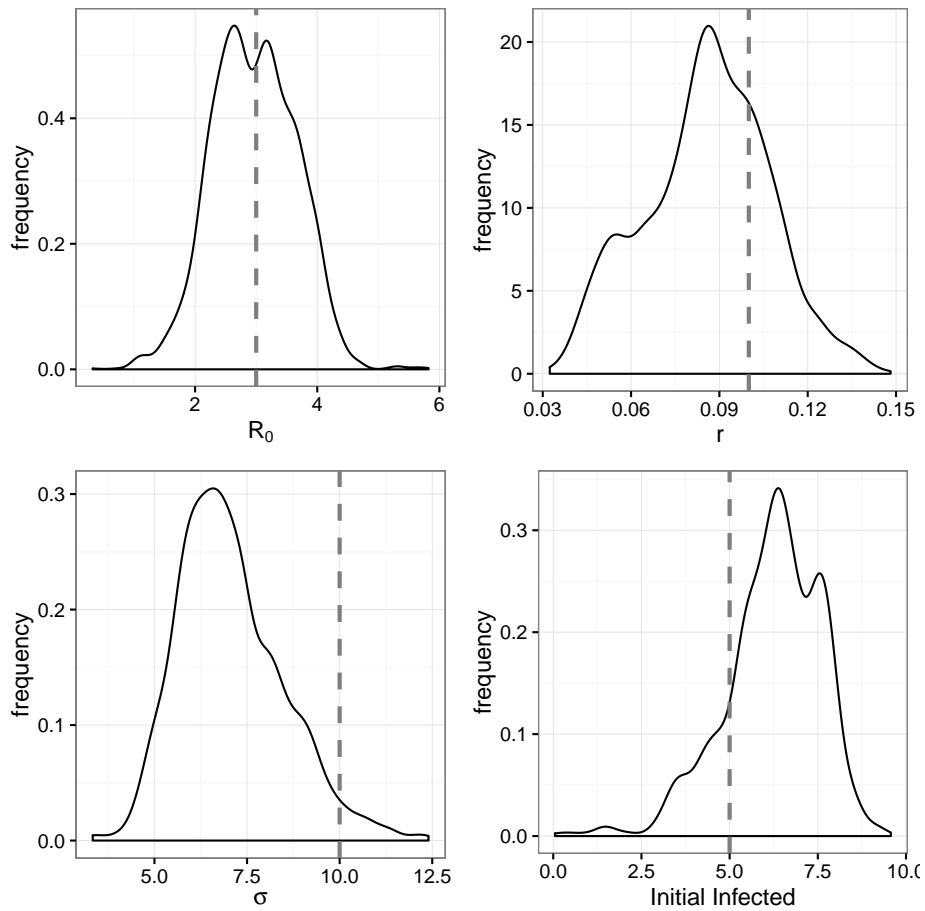


Figure 3.1: Kernel estimates for four essential system parameters. True values are indicated by dashed lines.

Chapter 4

Parameter Fitting

4.1 Fitting Setup

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

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

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

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

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

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

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

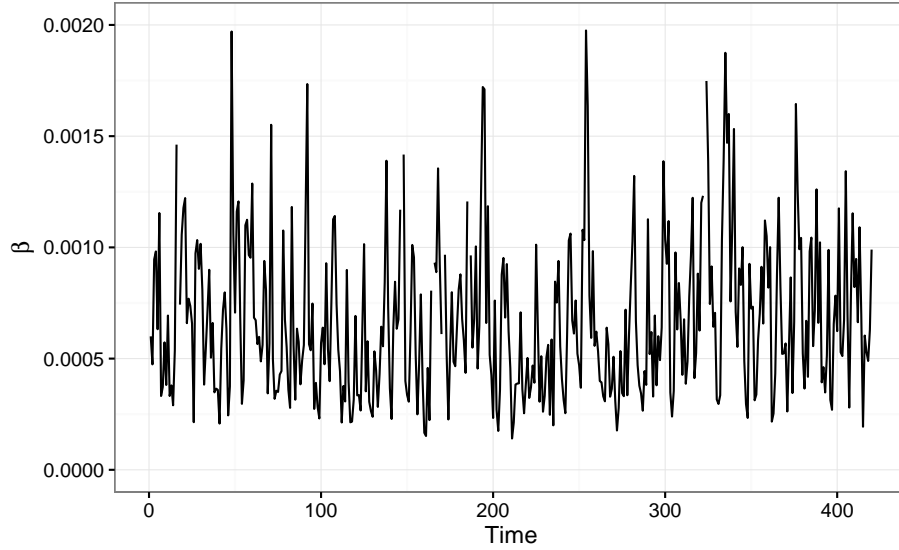


Figure 4.1: Simulated geometric autoregressive process show in Equation [4.1].

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

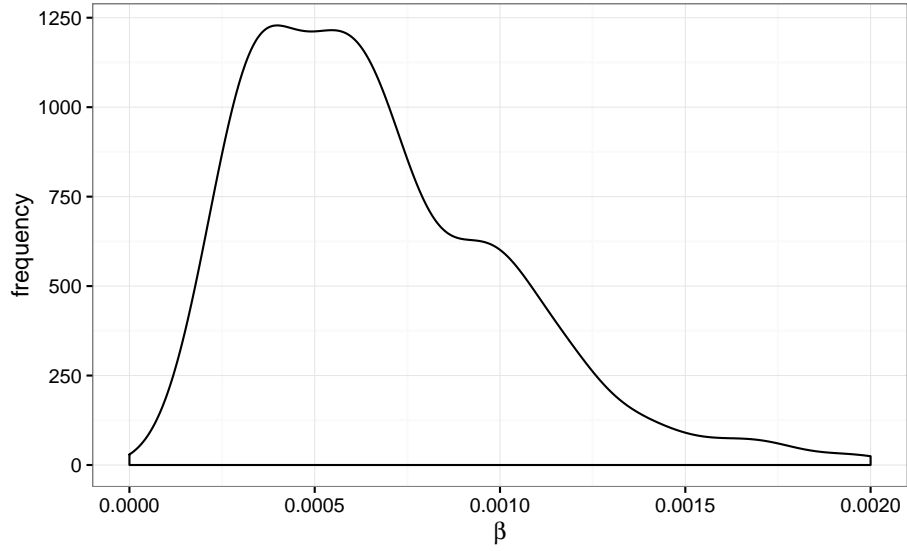


Figure 4.2: Density plot of values shown in Figure 4.1.

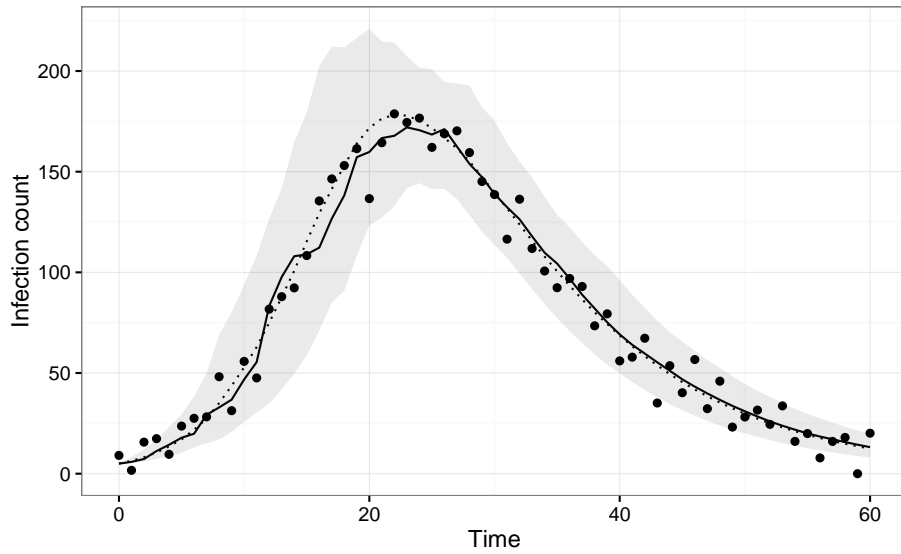


Figure 4.3: Stochastic SIR model simulated using an explicit Euler stepping scheme. The solid line is a single random trajectory, the dots show the data points obtained by adding in observation error defined as $\epsilon_{\text{obs}} = \mathcal{N}(0, 10)$, and the grey ribbon is centre 95th quantile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

1 4.2 Calibrating Samples

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

14 Suppose we are using a Monte-Carlo based method to obtain an estimate $\hat{\mu}_n$ for a
 15 quantity μ , where n is the number of samples. Then the Law of Large Numbers says
 16 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

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

22 which is known as the Monte Carlo Standard Error.

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

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

26 where 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

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

1 where P is the number of particles.

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

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

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

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

Name	True	IF2		HMCMC	
		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.

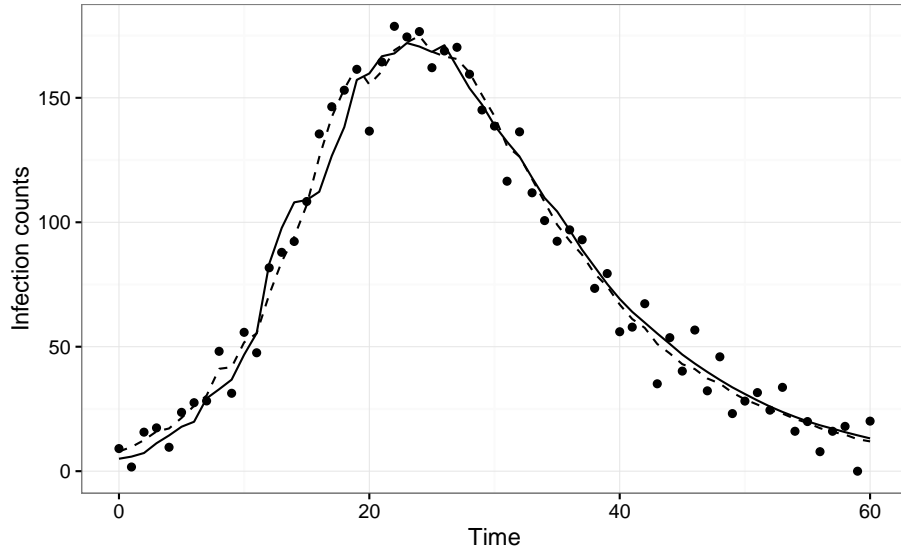


Figure 4.5: True system trajectory (solid line), observed data (dots), and IF2 estimated real state (dashed line).

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 paramete-
10 rs.

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

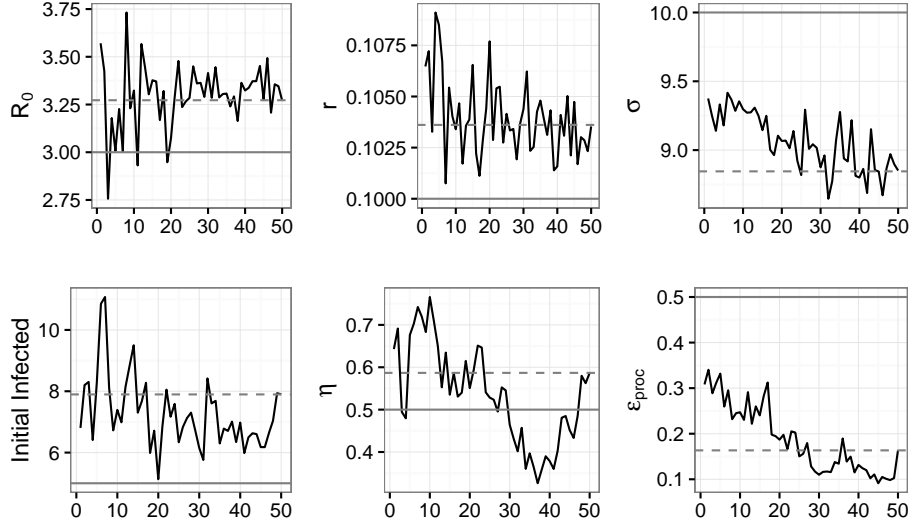


Figure 4.6: The horizontal axis shows the IF2 pass number. The solid black lines show the evolution of the ML estimates, the solid grey lines show the true value, and the dashed grey lines show the mean parameter estimates from the particle swarm after the final pass.

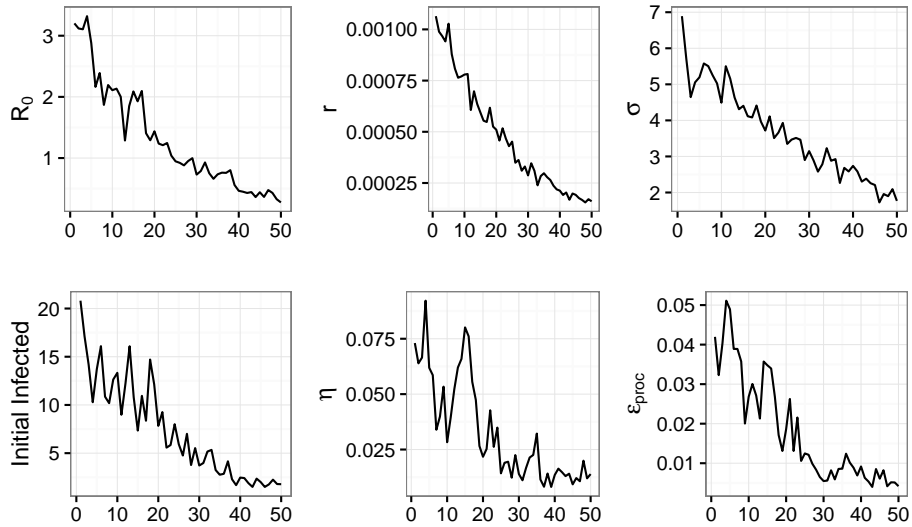


Figure 4.7: The horizontal axis shows the IF2 pass number and the solid black lines show the evolution of the standard deviations of the particle swarm values.

1 4.5 IF2 Densities

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

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

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

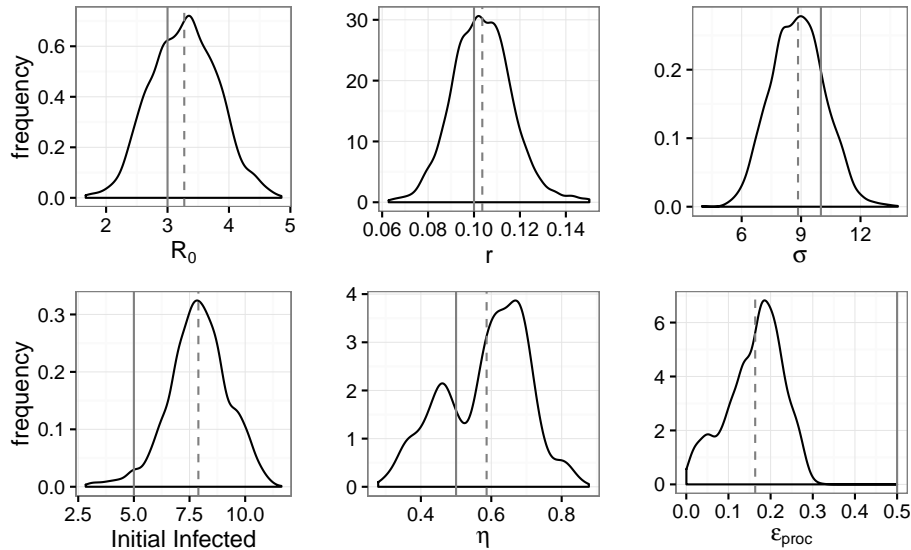


Figure 4.8: As before, the solid grey lines show the true parameter values and the dashed grey lines show the density means.

4.6 HMCMC Fitting

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

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

4.7 HMCMC Densities

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

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

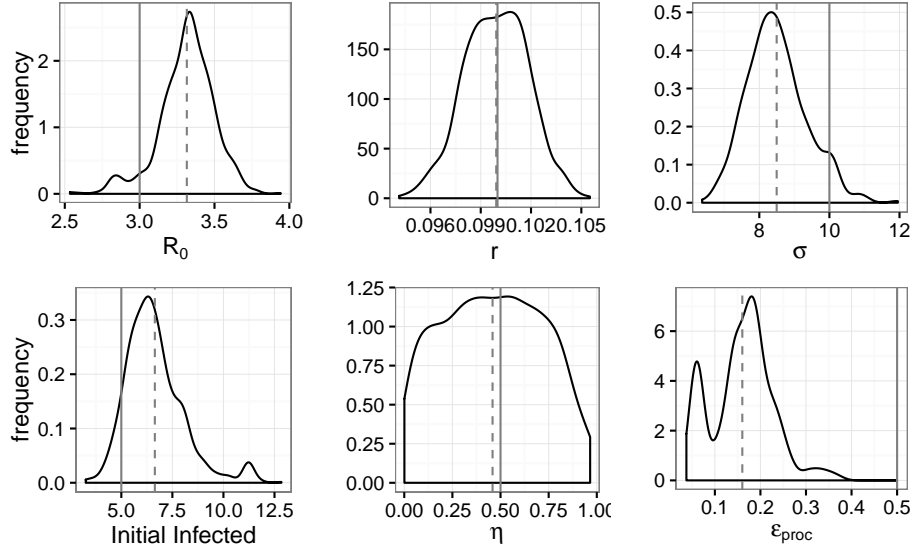


Figure 4.9: As before, the solid grey lines show the true parameter values and the dashed grey lines show the density means.

1 4.8 HMC MC 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 MC samples.

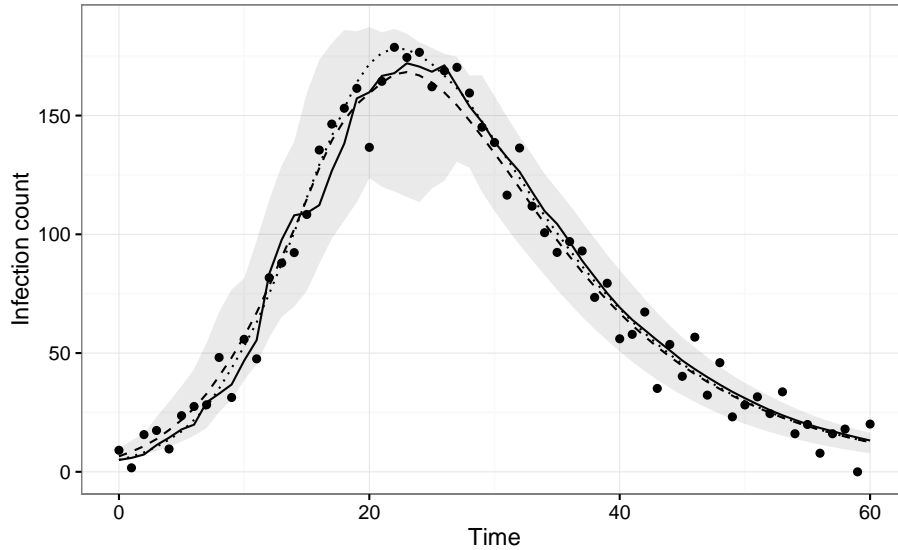


Figure 4.10: Result from 100 HMCBC bootstrap trajectories. The solid line shows the true states, the dots show the data, the dotted line shows the average system behaviour, the dashed line shows the bootstrap mean, and the grey ribbon shows the centre 95th quantile of the bootstrap trajectories.

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 HMCBC 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 HMCBC densities for the same param-
 8 eters.

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

10 The average running times were approximately 45.5 seconds and 257.4 seconds for IF2
 11 and HMCBC respectively, representing a 5.7x speedup for IF2 over HMCBC. While
 12 IF2 may be able to fit the model to data faster than HMCBC, 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 HMCBC are anything but, showing a wide spread of potential times.

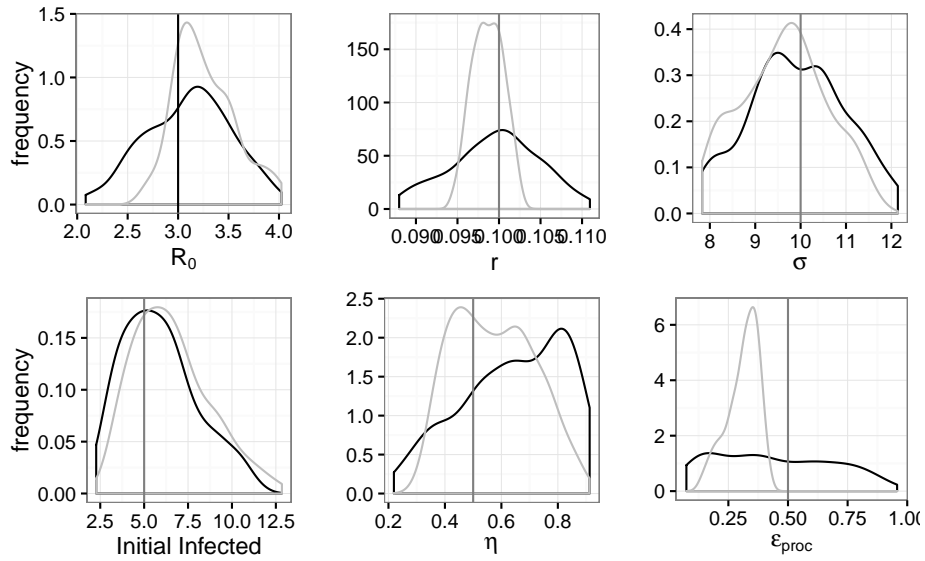


Figure 4.11: IF2 point estimate densities are shown in black and HMCMC point estimate densities are shown in grey. The vertical black lines show the true parameter values.

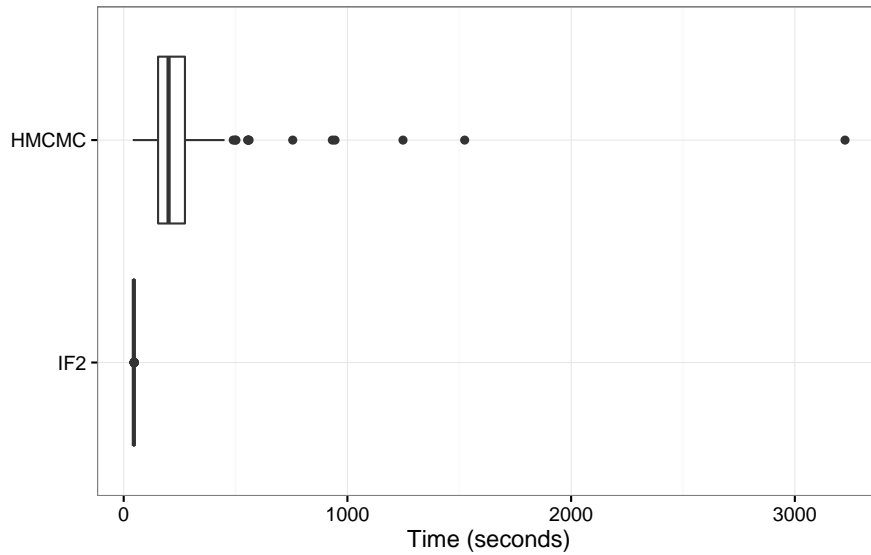


Figure 4.12: Fitting times for IF2 and HMCMC, in seconds. The centre box in each plot shows the centre 50th quantile, with the bold centre line showing the median.

Chapter 5

Forecasting Frameworks

5.1 Data Setup

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

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

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

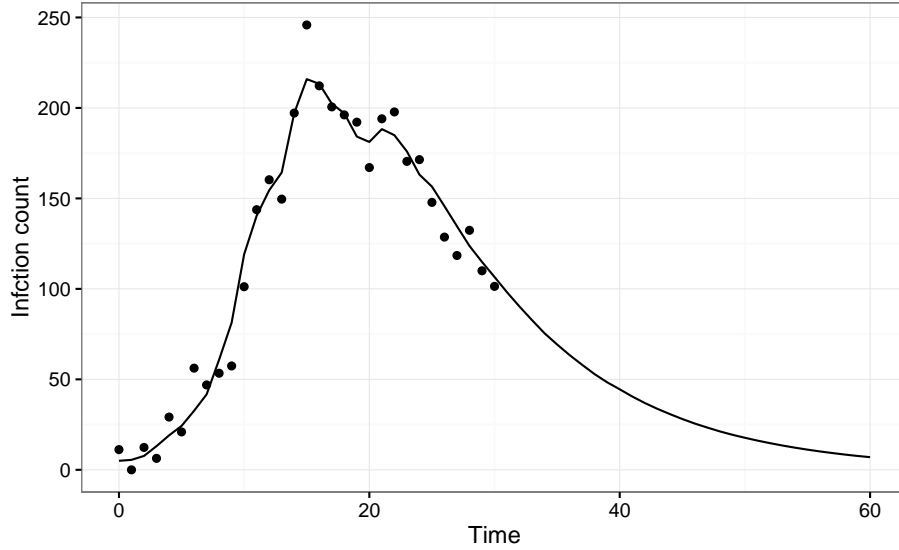


Figure 5.1: Infection count data truncated at $T = 30$. The solid line shows the true underlying system states, and the dots show those states with added observation noise. Parameters used were $R_0 = 3.0$, $r = 0.1$, $\eta = .05$, $\sigma_{proc} = 0.5$, and additive observation noise was drawn from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

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.

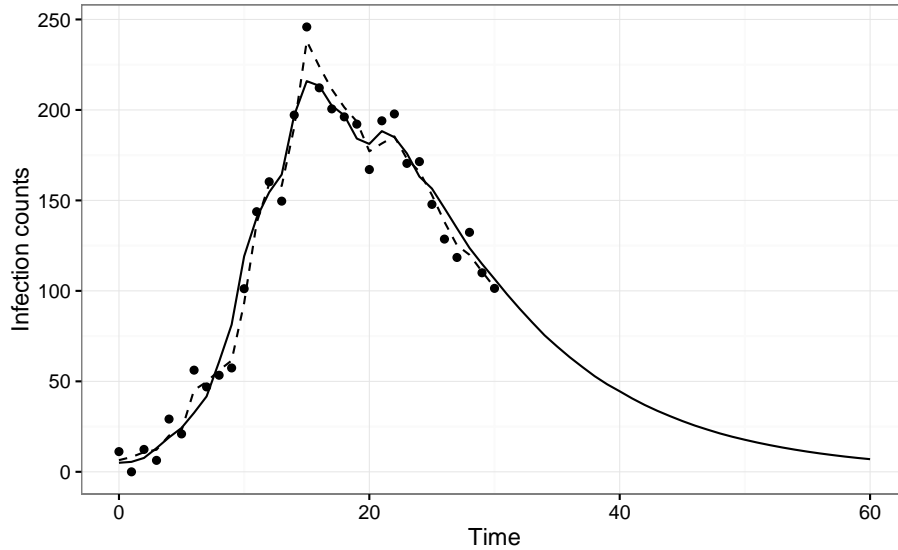


Figure 5.2: Infection count data truncated at $T = 30$ from Figure [5.1]. The dashed line shows IF2's attempt to reconstruct the true underlying state from the observed data points.

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

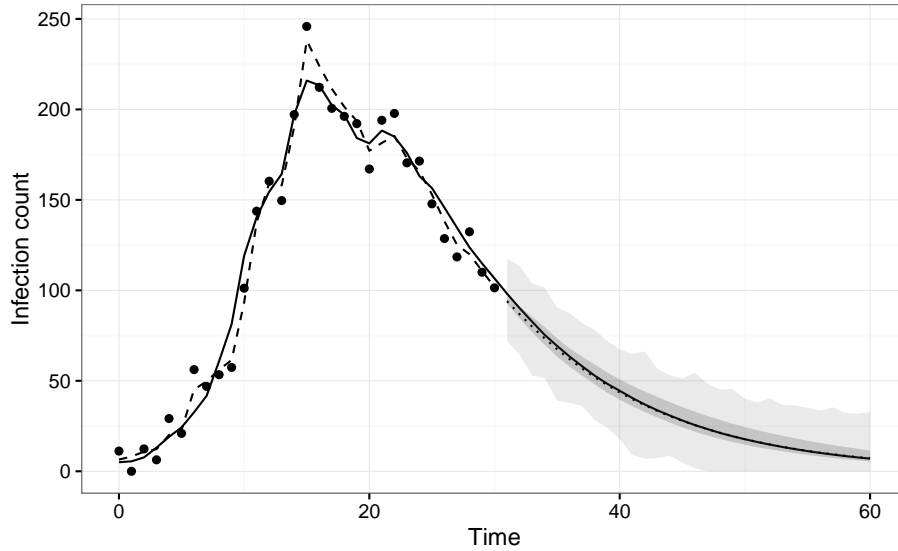


Figure 5.3: Forecast produced by the IF2 / parametric bootstrapping framework. The dotted line shows the mean estimate of the forecasts, the dark grey ribbon shows the 95% confidence interval based on the 0.025 and 0.975 quantiles on the true state estimates, and the lighter grey ribbon shows the same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

7 For HMC MC 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 HMC MC 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 $\overline{SSE} = 20.27$.

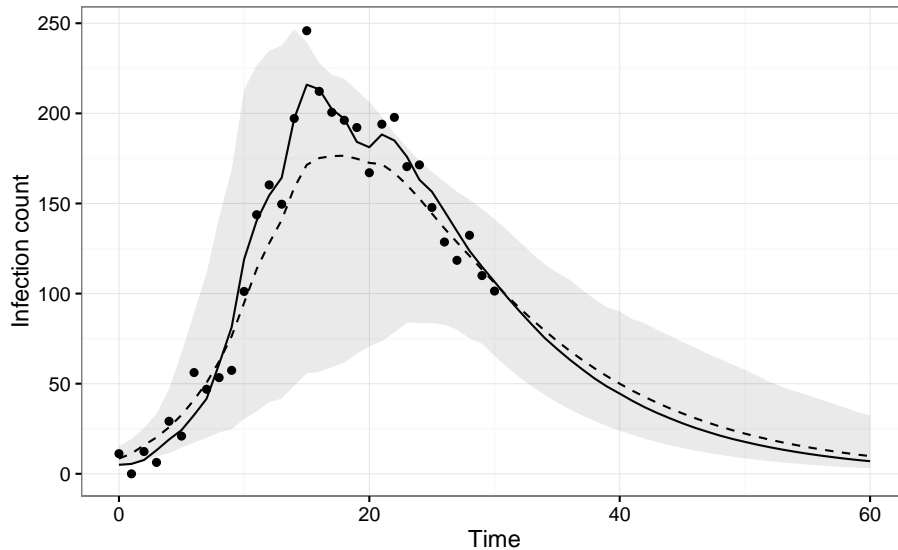


Figure 5.4: Forecast produced by the HMCMC / bootstrapping framework with $M = 200$ trajectories. The dotted line shows the mean estimate of the forecasts, and the grey ribbon shows the 95% confidence interval on the estimated true states as described in Figure [5.3].

1 5.4 Truncation vs. Error

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

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

10 IF2 and HMCMC 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 HMCMC 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.

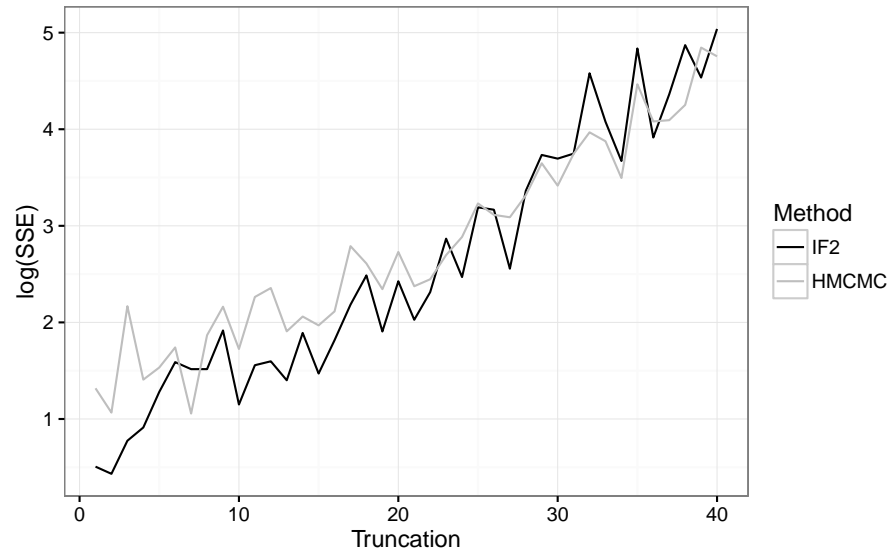


Figure 5.5: Error growth as a function of data truncation amount. Both methods used 200 bootstrap trajectories. Note that the y-axis shows the natural log of the averaged SSE, not the total SSE.

- 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.

Chapter 6

S-map and SIRS

6.1 S-maps

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

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

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

This function is defined as follows:

1 First construct a matrix A and vector b defined as

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

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

6 The weighting function w is defined as

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

8 where d is the euclidean distance between the predictor vector and library vectors in
9 Equation [6.1] and \bar{d} is the average of these distances. We can then see that 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  $\hat{y}_t$ 

```

6.3 SIRS Model

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

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

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

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

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

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

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

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

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

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

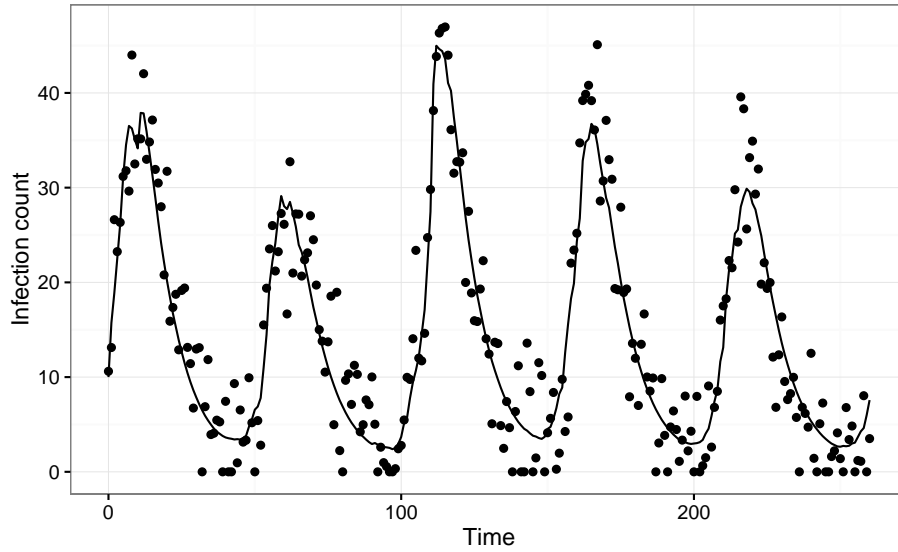


Figure 6.1: Five cycles generated by the SIRS function. The solid line the the true number of cases, dots show case counts with added observation noise. The parameter values were $R_0 = 3.0$, $\gamma = 0.1$, $\eta = 1$, $\sigma = 5$, and 10 initial cases.

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

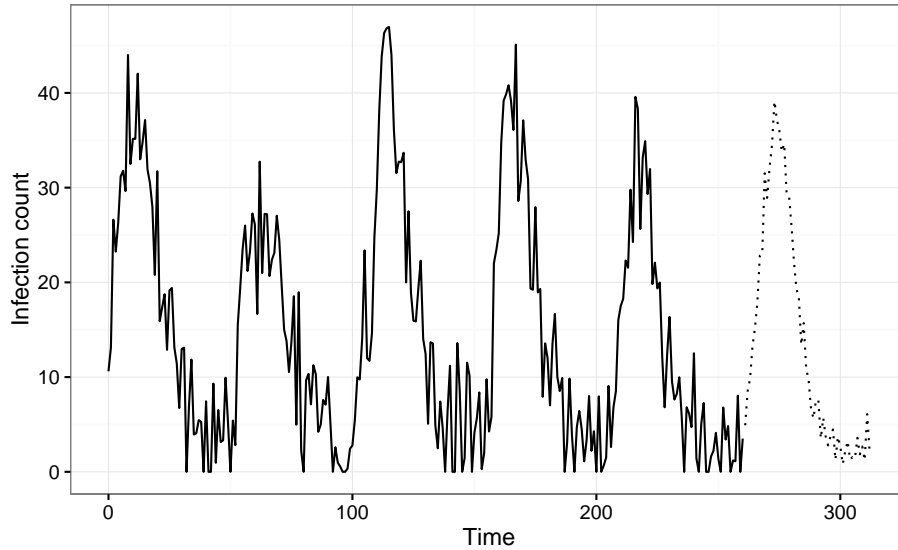


Figure 6.2: S-map applied to the data from the previous figure. The solid line shows the infection counts with observation noise from the previous plot, and the dotted line is the S-map forecast. Parameters chosen were $E = 14$ and $\theta = 3$.

- 1 The parameters used in the S-map algorithm to obtain the forecast used in Figure [6.2]
- 2 were obtained using a grid search of potential parameters outlined in [Sugihara ref].
- 3 The 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

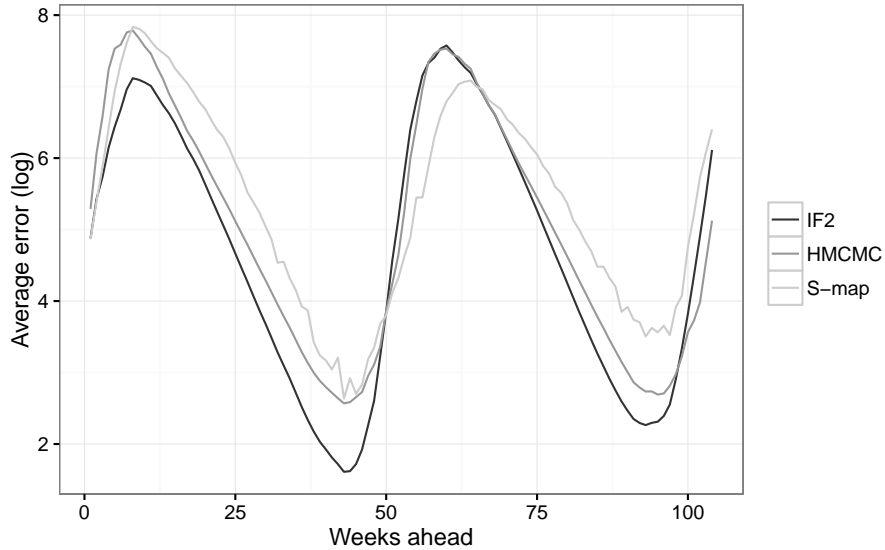


Figure 6.3: Error as a function of forecast length.

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

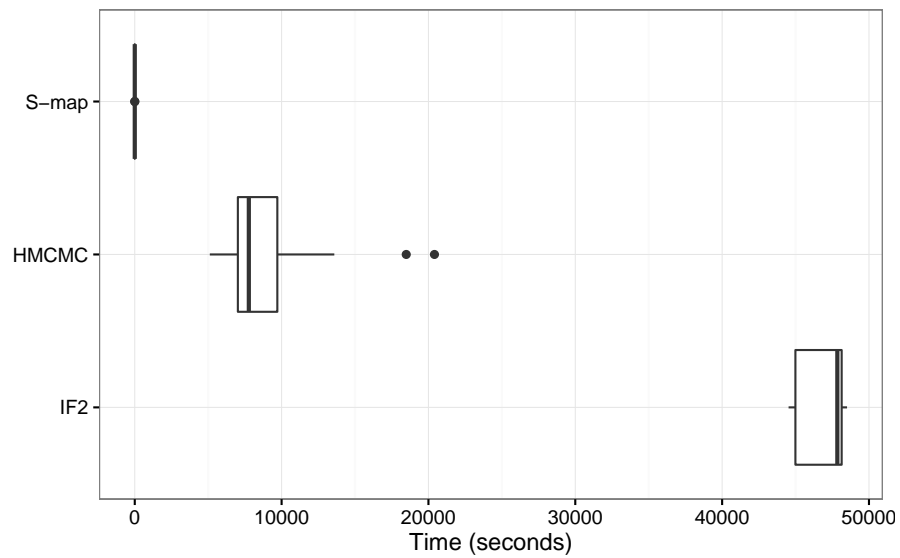


Figure 6.4: Runtimes for producing SIRS forecasts. The box shows the middle 50th quantile, the bold line is the median, and the dots are outliers. Note that these are not “true” outliers, simply ones outside a ranges based on the interquartile range.

Chapter 7

Spatial Epidemics

7.1 Spatial SIR

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

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

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

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

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

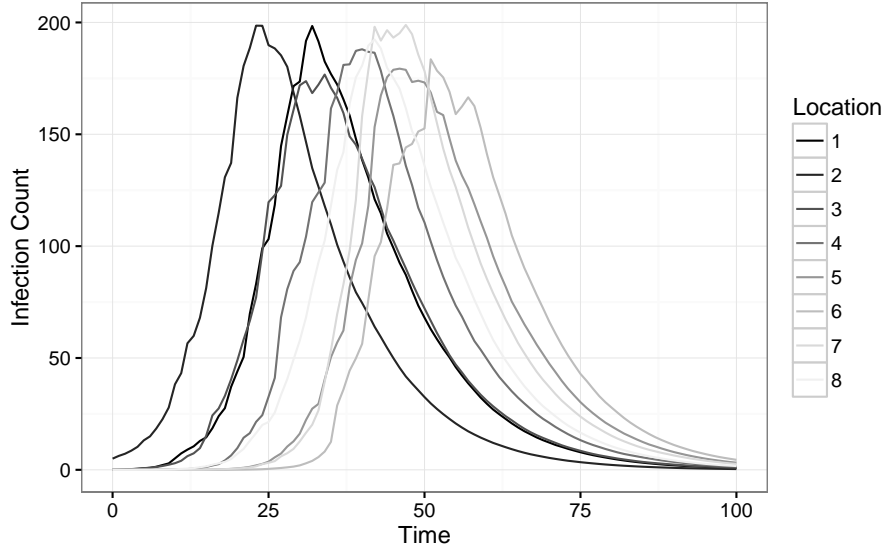


Figure 7.1: Evolution of a spatial epidemic in a ring topology. The outbreak was started with 5 cases in Location 2. Parameters were $R_0 = 3.0$, $\gamma = 0.1$, $\eta = 0.5$, $\sigma_{err} = 0.5$, and $\phi = 0.5$.

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

Note that as β is a state variable, each location has its own stochastic process driving the evolution of its β state.

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

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

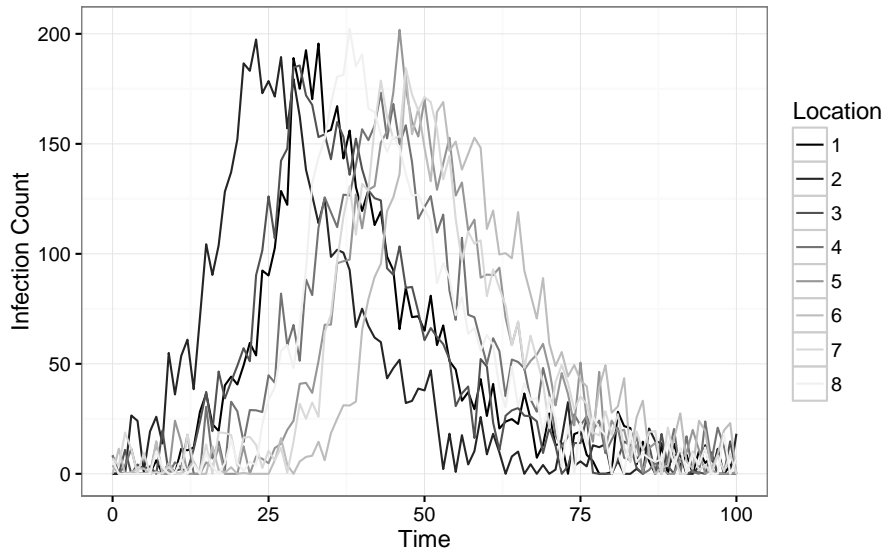


Figure 7.2: Evolution of a spatial epidemic as in Figure [7.1], with added observation noise drawn from $\mathcal{N}(0, 10)$.

7.2 Dewdrop Regression

Dewdrop regression (references) aims to overcome the primary disadvantage suffered by methods such as the S-map or its cousin Simplex Projection: the requirement of long time series from which to build a library. Suggested by Sugihara’s group in 2008, Dewdrop Regression works by stitching together shorter, related, time series, in order to give the S-map or similar methods enough data to operate on. The underlying idea is that as long as the underlying dynamics of the time series display similar behaviour (such as potentially collapsing to the same attractor), they can be treated as part of the same overarching system.

It is not enough to simply concatenate the shorter time series together – several procedures must be carried out and a few caveats observed. First, as the individual time series can be or drastically differing scales and breadths, they all must be rescaled to unit mean and variance. Then the library is constructed as before with an embedding dimension E , but any library vectors that span any of the seams joining the time series are discarded. Further, and predictions stemming from a library vector must stay within the time series from which they originated. In this way we are allowing the “shadow” of the underlying dynamics of the separate time series to infer the forecasts for segments of other time series. Once the library has been constructed, S-mapping can be carried out as previously specified.

This procedure is especially well-suited to the spatial model we are using. While the dynamics are stochastic, they still display very similar means and variances.

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

5 **7.3 Spatial Model Forecasting**

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

13 The results are shown in Figure [7.3].

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

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

20 As before, the S-map with Dewdrop Regression runs faster than the other two methods
 21 with a huge margin. It is again hard to see exactly how large the margin is from the
 22 figure due to the scale, but we can examine the average values: the average running
 23 time for S-mapping with Dewdrop Regression was about 249 seconds, whereas the
 24 average times for IF2 and HMC MC 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 HMC MC.

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

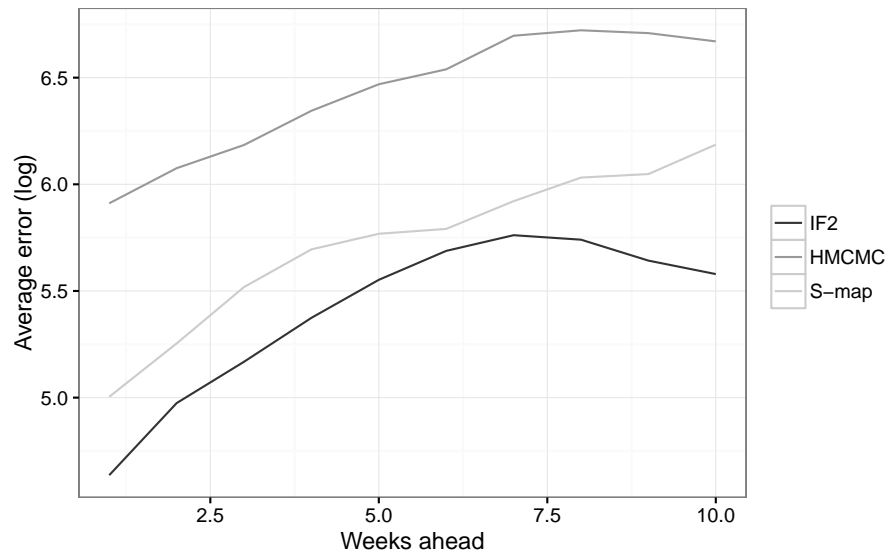


Figure 7.3: Average SSE (log scale) across each location and all trials as a function of the number of weeks ahead in the forecast.

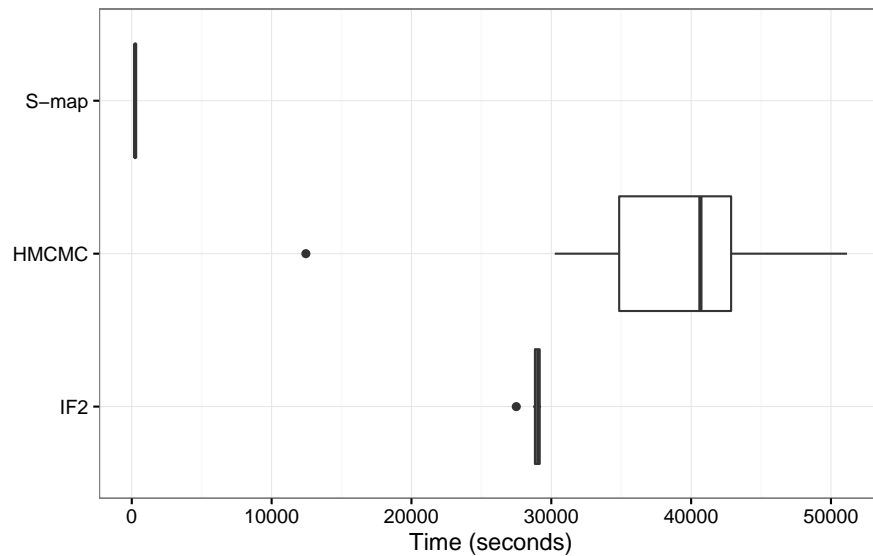


Figure 7.4: Runtimes for producing spatial SIR forecasts. The box shows the middle 50th quantile, the bold line is the median, and the dots are outliers.

Chapter 8

Discussion and Future Directions

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

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

8.1 Parallel and Distributed Computing

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

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 dissipa-
12 tion 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 must
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

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.

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-

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

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

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

8.2 IF2, Bootstrapping, and Forecasting Methodology

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

A recent paper utilising IF2 for forecasting [King reference] 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, HMC MC, 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.

Bibliography

- [1] Mohammad Ali et al. “Time Series Analysis of Cholera in Matlab, Bangladesh, during 1988-2001”. In: *Journal of health, population, ...* 31.1 (2013), pp. 11–19. URL: <http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3702354/>.
- [2] Christophe Andrieu et al. “An introduction to MCMC for machine learning”. In: *Machine Learning* 50.1-2 (2003), pp. 5–43. ISSN: 08856125. DOI: 10.1023/A:1020281327116. arXiv: 1109.4435v1.
- [3] M.S. Arulampalam et al. “A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking”. In: *IEEE Transactions on Signal Processing* 50.2 (2002), pp. 174–188. ISSN: 1053587X. DOI: 10.1109/78.978374. URL: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=978374>.
- [4] Jacob Bock Axelsen et al. “Multiannual forecasting of seasonal influenza dynamics reveals climatic and evolutionary drivers.” In: *Proceedings of the National Academy of Sciences of the United States of America* 111.26 (July 2014), pp. 9538–42. ISSN: 1091-6490. DOI: 10.1073/pnas.1321656111. URL: <http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=4084473%7B%5C%7Dtool=pmcentrez%7B%5C%7Drendertype=abstract>.
- [5] a. Camacho et al. “Explaining rapid reinfections in multiple-wave influenza outbreaks: Tristan da Cunha 1971 epidemic as a case study”. In: *Proceedings of the Royal Society B: Biological Sciences* 278 (2011), pp. 3635–3643. ISSN: 0962-8452. DOI: 10.1098/rspb.2011.0300.
- [6] Jean-Paul Chretien et al. “Influenza forecasting in human populations: a scoping review.” In: *PloS one* 9.4 (Jan. 2014), e94130. ISSN: 1932-6203. DOI: 10.1371/journal.pone.0094130. URL: <http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3979760%7B%5C%7Dtool=pmcentrez%7B%5C%7Drendertype=abstract>.
- [7] 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.

- 1 [8] Andrea Freyer Dugas et al. “Influenza forecasting with Google Flu Trends.” In:
2 *PloS one* 8.2 (Jan. 2013), e56176. ISSN: 1932-6203. DOI: 10.1371/journal.pone.
3 0056176. URL: [http://www.pubmedcentral.nih.gov/articlerender.fcgi?](http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3572967%7B%5C%7Dtool=pmcentrez%7B%5C%7Drendertype=abstract)
4 [artid=3572967%7B%5C%7Dtool=pmcentrez%7B%5C%7Drendertype=abstract](http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3572967%7B%5C%7Dtool=pmcentrez%7B%5C%7Drendertype=abstract).
- 5 [9] Sarah M. Glaser, Hao Ye, and George Sugihara. “A nonlinear, low data re-
6 quirement model for producing spatially explicit fishery forecasts”. In: *Fisheries*
7 *Oceanography* 23 (2014), pp. 45–53. ISSN: 10546006. DOI: 10.1111/fog.12042.
- 8 [10] Andrea L Graham et al. “Explaining rapid reinfections in multiple- wave in-
9 fluenza outbreaks : Tristan da Cunha 1971 epidemic as a case study”. In: *Proc.*
10 *R. Soc. B* (2016). DOI: 10.1098/rspb.2011.0300.
- 11 [11] Florian Hartig et al. “Statistical inference for stochastic simulation models -
12 theory and application”. In: *Ecology Letters* 14.8 (2011), pp. 816–827. ISSN:
13 1461023X. DOI: 10.1111/j.1461-0248.2011.01640.x.
- 14 [12] Chih-hao Hsieh, Christian Anderson, and George Sugihara. “Extending nonlin-
15 ear analysis to short ecological time series.” In: *The American naturalist* 171.1
16 (2008), pp. 71–80. ISSN: 0003-0147. DOI: 10.1086/524202.
- 17 [13] E L Ionides, C Bret, and a a King. “Inference for nonlinear dynamical systems.”
18 In: *Proceedings of the National Academy of Sciences of the United States of*
19 *America* 103.49 (Dec. 2006), pp. 18438–43. ISSN: 0027-8424. DOI: 10.1073/pnas.
20 0603181103. URL: [http://www.pubmedcentral.nih.gov/articlerender.fcgi?](http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3020138%7B%5C%7Dtool=pmcentrez%7B%5C%7Drendertype=abstract)
21 [artid=3020138%7B%5C%7Dtool=pmcentrez%7B%5C%7Drendertype=abstract](http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3020138%7B%5C%7Dtool=pmcentrez%7B%5C%7Drendertype=abstract).
- 22 [14] Edward L. Ionides et al. “Inference for dynamic and latent variable models
23 via iterated, perturbed Bayes maps”. In: *Proceedings of the National Academy*
24 *of Sciences* 112.3 (2015), pp. 719–724. ISSN: 0027-8424. DOI: 10.1073/pnas.
25 1410597112. URL: [http://www.pnas.org/lookup/doi/10.1073/pnas.](http://www.pnas.org/lookup/doi/10.1073/pnas.1410597112)
26 [1410597112](http://www.pnas.org/lookup/doi/10.1073/pnas.1410597112).
- 27 [15] Aaron A King et al. “Avoidable errors in the modelling of outbreaks of emerg-
28 ing pathogens, with special reference to Ebola”. In: *Proceedings of the Royal*
29 *Society B: Biological Sciences* 282.1806 (2015), pp. 20150347–20150347. ISSN:
30 0962-8452. DOI: 10.1098/rspb.2015.0347. arXiv: 1412.0968. URL: [http:](http://rspb.royalsocietypublishing.org/cgi/doi/10.1098/rspb.2015.0347)
31 [//rspb.royalsocietypublishing.org/cgi/doi/10.1098/rspb.2015.0347](http://rspb.royalsocietypublishing.org/cgi/doi/10.1098/rspb.2015.0347).
- 32 [16] Kanti V. Mardia et al. “The Kriged Kalman filter”. In: *Test* 7.2 (1998), pp. 217–
33 282. ISSN: 11330686. DOI: 10.1007/BF02565111.
- 34 [17] Radford M Neal. “Handbook of Markov Chain Monte Carlo”. In: *Handbook of*
35 *Markov Chain Monte Carlo* 20116022 (2011), pp. 113–162. ISSN: jnull. DOI:
36 10.1201/b10905. arXiv: arXiv:1206.1901v1. URL: [http://www.crcnetbase.](http://www.crcnetbase.com/doi/book/10.1201/b10905)
37 [com/doi/book/10.1201/b10905](http://www.crcnetbase.com/doi/book/10.1201/b10905).

- 1 [18] Elaine O Nsoesie et al. “A systematic review of studies on forecasting the dy-
2 namics of influenza outbreaks.” In: *Influenza and other respiratory viruses* 8.3
3 (May 2014), pp. 309–16. ISSN: 1750-2659. DOI: 10.1111/irv.12226. URL: [http:
4 //www.pubmedcentral.nih.gov/articlerender.fcgi?artid=4181479%7B%5C%
5 %7Dtool=pmcentrez%7B%5C%7Drendertype=abstract](http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=4181479%7B%5C%7Dtool=pmcentrez%7B%5C%7Drendertype=abstract).
- 6 [19] Elaine Nsoesie, Madhav Mararthe, and John Brownstein. “Forecasting peaks of
7 seasonal influenza epidemics.” In: *PLoS currents* 5 (Jan. 2013), pp. 1–14. ISSN:
8 2157-3999. DOI: 10.1371/currents.outbreaks.bb1e879a23137022ea79a8c508b030bc.
9 URL: [http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=
10 3712489%7B%5C%7Dtool=pmcentrez%7B%5C%7Drendertype=abstract](http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3712489%7B%5C%7Dtool=pmcentrez%7B%5C%7Drendertype=abstract).
- 11 [20] Robert C Reiner et al. “Highly localized sensitivity to climate forcing drives
12 endemic cholera in a megacity.” In: *Proceedings of the National Academy of
13 Sciences of the United States of America* 109.6 (Feb. 2012), pp. 2033–6. ISSN:
14 1091-6490. DOI: 10.1073/pnas.1108438109. URL: [http://www.pubmedcentral.
15 nih.gov/articlerender.fcgi?artid=3277579%7B%5C%7Dtool=pmcentrez%
16 7B%5C%7Drendertype=abstract](http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3277579%7B%5C%7Dtool=pmcentrez%7B%5C%7Drendertype=abstract).
- 17 [21] Sujit K Sahu. “A Bayesian Kriged-Kalman model for short-term fore- casting of
18 air pollution levels”. In: *Appl. Statist* 54 (2005), pp. 223–244. ISSN: 0035-9254.
19 DOI: 10.1111/j.1467-9876.2005.00480.x.
- 20 [22] Jacques Sau et al. “Particle filter-based real-time estimation and prediction of
21 traffic conditions Modeling framework”. In: *Information Systems* (1918), pp. 1–
22 8.
- 23 [23] Jeffrey Shaman and Alicia Karspeck. “Forecasting seasonal outbreaks of in-
24 fluenza.” In: *Proceedings of the National Academy of Sciences of the United
25 States of America* 109.50 (Dec. 2012), pp. 20425–30. ISSN: 1091-6490. DOI:
26 10.1073/pnas.1208772109. URL: [http://www.pubmedcentral.nih.gov/
27 articlerender.fcgi?artid=3528592%7B%5C%7Dtool=pmcentrez%7B%5C%
28 %7Drendertype=abstract](http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3528592%7B%5C%7Dtool=pmcentrez%7B%5C%7Drendertype=abstract).
- 29 [24] G Sugihara and R M May. *Nonlinear forecasting as a way of distinguishing
30 chaos from measurement error in time series*. 1990. DOI: 10.1038/344734a0.
- 31 [25] George Sugihara. “Nonlinear Forecasting for the Classification of Natural Time
32 Series”. In: *Philosophical Transactions of the Royal Society A: Mathematical,
33 Physical and Engineering Sciences* 348 (1994), pp. 477–495. ISSN: 1364-503X.
34 DOI: 10.1098/rsta.1994.0106.
- 35 [26] Carmen L Vidal Rodeiro and Andrew B Lawson. “Online updating of space-
36 time disease surveillance models via particle filters.” In: *Statistical methods in
37 medical research* 15.5 (2006), pp. 423–444. ISSN: 0962-2802. DOI: 10.1177/
38 0962280206071640.

- 1 [27] 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. URL: [http://www.pubmedcentral.nih.gov/
5 articlerender.fcgi?artid=3998879%7B%5C%7Dtool=pmcentrez%7B%5C%
6 %7Drendertype=abstract](http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3998879%7B%5C%7Dtool=pmcentrez%7B%5C%7Drendertype=abstract).
- 7 [28] Xingyu Zhang et al. “Comparative study of four time series methods in fore-
8 casting typhoid fever incidence in China.” In: *PloS one* 8.5 (Jan. 2013), e63116.
9 ISSN: 1932-6203. DOI: 10.1371/journal.pone.0063116. URL: [http://www.
10 pubmedcentral.nih.gov/articlerender.fcgi?artid=3641111%7B%5C%
11 %7Dtool=pmcentrez%7B%5C%7Drendertype=abstract](http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3641111%7B%5C%7Dtool=pmcentrez%7B%5C%7Drendertype=abstract).

1 Appendix A

2 Hamiltonian MCMC

3 A.1 Full R code

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

```
5  
6 1 ## Dexter Barrows  
7 2 ## 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 60                  y = standata,    # infection count data
34 61                  N = 500,         # population size
35 62                  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 67                      iter  = 2000,    # iterations per chain
41 68                      warmup = 1000,    # warmup iterations
42 69                      thin   = 2)       # thinning number
43 70 fit <- stan(file = "d_sirole_euler.stan",
44 71            data   = sir_data,
45 72            chains  = stan_options$chains,
46 73            iter    = stan_options$iter,
47 74            warmup  = stan_options$warmup,
48 75            thin    = stan_options$thin )
49 76
50 77 exfit <- extract(fit, permuted = TRUE, inc_warmup = FALSE)
51 78

```



```

1 79 R0points <- exfit$R0
2 80 R0kernel <- qplot(R0points, geom = "density", xlab = expression(R[0]),
3      ylab = "frequency") +
4 81     geom_vline(aes(xintercept=R0), linetype="dashed", size=1,
5      color="grey50") +
6 82     theme_bw()
7 83
8 84 print(R0kernel)
9 85 ggsave(R0kernel, filename="kernelR0.pdf", height=3, width=3.25)
10 86
11 87 rpoinst <- exfit$r
12 88 rkernel <- qplot(rpoinst, 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,
45 115         aes(x = iterations,
46 116             y = value,
47 117             color = factor(chains, labels = 1:stan_
48 118                 options$chains))) +
49 118     labs(x = "Sample", y = expression(R[0]), color = "Chain
50 119         ") +
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)
25

```

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 4
33 5 data {
34 6
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 11
40 12 }
41 13
42 14 parameters {
43 15
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 trajecory 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 54         geom_point(aes(y = infec_counts)) +
26 55         theme_bw()
27 56
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 67
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 72                 ylab = "frequency") +
46 73                 geom_vline(aes(xintercept=true_pars[["R0"]]), linetype="
47 74                 dashed", size=1, color="grey50") +
48 75                 theme_bw()
49 76
50 75 print(R0kernel)
51 76 ggsave(R0kernel, filename="kernelR0.pdf", height=3, width=3.25)

```

```

1 77
2 78 rpoints <- paramdata$r
3 79 rkernell <- qplot(rpoints, geom = "density", xlab = "r", ylab = "
4     frequency") +
5 80     geom_vline(aes(xintercept=true_pars[["r"]]), linetype="dashed
6     ", size=1, color="grey50") +
7 81     theme_bw()
8 82
9 83 print(rkernell)
10 84 ggsave(rkernell, filename="kernelr.pdf", height=3, width=3.25)
11 85
12 86 sigmapoints <- paramdata$sigma
13 87 sigmakernell <- qplot(sigmapoints, geom = "density", xlab = expression(
14     sigma), ylab = "frequency") +
15 88     geom_vline(aes(xintercept=sigma), linetype="dashed", size=1,
16     color="grey50") +
17 89     theme_bw()
18 90
19 91 print(sigmakernell)
20 92 ggsave(sigmakernell, filename="kernelsigma.pdf", height=3, width=3.25)
21 93
22 94 infecpoints <- paramdata$Iinit
23 95 infeckernell <- qplot(infecpoints, geom = "density", xlab = "Initial
24     Infected", ylab = "frequency") +
25 96     geom_vline(aes(xintercept=true_init_cond[["I"]]), linetype="
26     dashed", size=1, color="grey50") +
27 97     theme_bw()
28 98
29 99 print(infeckernell)
30 100 ggsave(infeckernell, filename="kernelinfec.pdf", height=3, width=3.25)
31 101
32 102 # show grid
33 103 grid.arrange(R0kernell, rkernell, sigmakernell, infeckernell, ncol = 2,
34     nrow = 2)
35 104
36 105 pdf("if2kernels.pdf", height = 6.5, width = 6.5)
37 106 grid.arrange(R0kernell, rkernell, sigmakernell, infeckernell, ncol = 2,
38     nrow = 2)
39 107 dev.off()
40 108 #ggsave(filename="if2kernels.pdf", g2, height=6.5, width=6.5)
41

```

42 B.2 Full C++ code

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

```

44
45 1 /* Author: Dexter Barrows
46 2   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 63 // [[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   = I0;
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             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.sigmean, pInfo.sigmasd);
40 189 printf("S_init  %f %f\n", pInfo.Sinitmean, pInfo.Sinit_sd);
41 190 printf("I_init  %f %f\n", pInfo.Iinitmean, pInfo.Iinit_sd);
42 191 printf("R_init  %f %f\n", pInfo.Rinitmean, pInfo.Rinit_sd);
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 double h      - time step size
18 double t0     - start time
19 double tn     - stop time
20 double * y    - current system state; a three-component vector
21                representing [S I R], susceptible-infected-recovered
22 220
23 221     */
24 222 void exp_euler_SIR(double h, double t0, double tn, int N, Particle *
25 particle) {
26 223
27 224     int num_steps = floor( (tn-t0) / h );
28 225
29 226     double S = particle->S;
30 227     double I = particle->I;
31 228     double R = particle->R;
32 229
33 230     double R0   = particle->R0;
34 231     double r    = particle->r;
35 232     double B    = R0 * r / N;
36 233
37 234     for(int i = 0; i < num_steps; i++) {
38 235         // get derivatives
39 236         double dS = - B*S*I;
40 237         double dI = B*S*I - r*I;
41 238         double dR = r*I;
42 239         // step forward by h
43 240         S += h*dS;
44 241         I += h*dI;
45 242         R += h*dR;
46 243     }
47 244
48 245     particle->S = S;
49 246     particle->I = I;
50 247     particle->R = R;
51 248

```

```

1 249 }
2 250
3 251
4 252 /* Particle pertubation function to be run between iterations and
5      passes
6 253
7 254 */
8 255 void perturbParticles(Particle * particles, int N, int NP, int
9      passnum, double coolrate) {
10 256
11 257     double coolcoef = pow(coolrate, passnum);
12 258
13 259     double spreadR0      = coolcoef * R0true / 10.0;
14 260     double spreadr       = coolcoef * rtrue / 10.0;
15 261     double spreadsigma   = coolcoef * merr / 10.0;
16 262     double spreadIinit   = coolcoef * I0 / 10.0;
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 /* Convinience function for particle resampling process
49 295
50 296 */
51 297 void copyParticle(Particle * dst, Particle * src) {

```

```

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

```

```

1 345     for (int n = 0; n < NP; n++) {
2 346
3 347         R0sd      += ( particles[n].R0 - R0mean ) * ( particles[n].R0 -
4           R0mean );
5 348         rsd       += ( particles[n].r - rmean ) * ( particles[n].r -
6           rmean );
7 349         sigmasd  += ( particles[n].sigma - sigmamean ) * ( particles[n]
8           ].sigma - sigmamean );
9 350         Sinitd   += ( particles[n].Sinit - Sinitmean ) * ( particles[n]
10          ].Sinit - Sinitmean );
11 351         Iinitd   += ( particles[n].Iinit - Iinitmean ) * ( particles[n]
12          ].Iinit - Iinitmean );
13 352         Rinitd   += ( particles[n].Rinit - Rinitmean ) * ( particles[n]
14          ].Rinit - Rinitmean );
15 353
16 354     }
17 355
18 356     R0sd      /= NP;
19 357     rsd       /= NP;
20 358     sigmasd   /= NP;
21 359     Sinitd    /= NP;
22 360     Iinitd    /= NP;
23 361     Rinitd    /= 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   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             Iinitd    = 0.0,
20 408             Rinitd    = 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                    R0mean );
26 413         rsd       += ( particles[n].r - rmean ) * ( particles[n].r -
27                    rmean );
28 414         sigmasd += ( particles[n].sigma - sigmamean ) * ( particles[n]
29                    ].sigma - sigmamean );
30 415         Sinitsd += ( particles[n].Sinit - Sinitmean ) * ( particles[n]
31                    ].Sinit - Sinitmean );
32 416         Iinitd  += ( particles[n].Iinit - Iinitmean ) * ( particles[n]
33                    ].Iinit - Iinitmean );
34 417         Rinitd  += ( particles[n].Rinit - Rinitmean ) * ( particles[n]
35                    ].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     Iinitd        /= NP;
44 426     Rinitd        /= 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->Sinitstd  = Sinitstd;
3 436     partInfo->Iinitmean = Iinitmean;
4 437     partInfo->Iinitstd  = Iinitstd;
5 438     partInfo->Rinitmean = Rinitmean;
6 439     partInfo->Rinitstd  = Rinitstd;
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
18 451    standard deviation 1
19 452    Uses the polar form of the Box-Muller transformation
20 453    From http://www.design.caltech.edu/erik/Misc/Gaussian.html
21 454    */
22 455 double randn() {
23 456
24 457     double x1, x2, w, y1;
25 458
26 459     do {
27 460         x1 = 2.0 * randu() - 1.0;
28 461         x2 = 2.0 * randu() - 1.0;
29 462         w = x1 * x1 + x2 * x2;
30 463     } while ( w >= 1.0 );
31 464
32 465     w = sqrt( (-2.0 * log( w ) ) / w );
33 466     y1 = x1 * w;
34 467
35 468     return y1;
36 469 }
37 470
38 471 }
```


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 1 # Dexter Barrows
6 2 #
7 3 # IF2 parametric bootstrapping function
8 4
9 5 library(foreach)
10 6 library(parallel)
11 7 library(doParallel)
12 8 library(Rcpp)
13 9
14 10 if2_paraboot ← function(if2data_parent, T, Tlim, steps, N, nTrials,
15 11 if2file, if2_s_file, stoc_sir_file, NP, nPasses, coolrate) {
16 12
17 13   source(stoc_sir_file)
18 14
19 15   if (nTrials < 2)
20 16     ntrials ← 2
21 17
22 18   # unpack if2 first fit data
23 19   # ...parameters
24 20   paramdata_parent ← data.frame( if2data_parent$paramdata )
25 21   names(paramdata_parent) ← c("R0", "r", "sigma", "eta", "berr", "
26 22     Sinit", "Iinit", "Rinit")
27 23   parmeans_parent ← colMeans(paramdata_parent)
28 24   names(parmmeans_parent) ← c("R0", "r", "sigma", "eta", "berr", "
29 25     Sinit", "Iinit", "Rinit")
30 26   # ...states
31 27   statedata_parent ← data.frame( if2data_parent$statedata )
32 28   names(statedata_parent) ← c("S", "I", "R", "B")
33 29   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 40               c(R0 = R0,
16 41                 r = r,
17 42                 N = N,
18 43                 eta = eta,
19 44                 berr = berr) )
20 45
21 46   init_cond ← with( as.list(parmeans_parent),
22 47                    c(S = Sinit,
23 48                      I = Iinit,
24 49                      R = Rinit) )
25 50
26 51   # generate trajectory
27 52   sdeout ← StocSIR(init_cond, pars, Tlim + 1, steps)
28 53   colnames(sdeout) ← c('S', 'I', 'R', 'B')
29 54
30 55   # add noise
31 56   counts_raw ← sdeout[, 'I'] + rnorm(dim(sdeout)[1], 0, parmeans_
32   parent[['sigma']])
33 57   counts      ← ifelse(counts_raw < 0, 0, counts_raw)
34 58
35 59   ## refit using new data
36 60   ##
37 61
38 62   rm(if2) # because stupid things get done in packages
39 63   sourceCpp(if2file)
40 64   if2time ← system.time( if2data ← if2(counts, Tlim+1, N, NP,
41   nPasses, coolrate) )
42 65
43 66   paramdata ← data.frame( if2data$paramdata )
44 67   names(paramdata) ← c("R0", "r", "sigma", "eta", "berr", "Sinit",
45   "Iinit", "Rinit")
46 68   parmeans ← colMeans(paramdata)
47 69   names(parmeans) ← c("R0", "r", "sigma", "eta", "berr", "Sinit", "
48   Iinit", "Rinit")
49 70
50 71   ## generate the rest of the trajectory
51 72   ##

```

```

1  73
2  74   # pack new parameter estimates
3  75   pars ← with( as.list(parmmeans),
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 1 StocSIRstan ← function(y, pars, T, steps, berrvec, bveclim) {
32 2
33 3   out ← matrix(NA, nrow = (T+1), ncol = 4)
34 4
35 5   R0 ← pars[['R0']]
36 6   r ← pars[['r']]
37 7   N ← pars[['N']]
38 8   eta ← pars[['eta']]
39 9   berr ← pars[['berr']]
40 10
41 11   S ← y[['S']]
42 12   I ← y[['I']]
43 13   R ← y[['R']]
44 14
45 15   B0 ← R0 * r / N
46 16   B ← B0

```

```

1 17
2 18 out[1,] ← c(S,I,R,B)
3 19
4 20 h ← 1 / steps
5 21
6 22 for ( i in 1:(T*steps) ) {
7 23
8 24     if (i <= bveclim) {
9 25         B ← exp( log(B) + eta*(log(B0) - log(B)) + berrvec[i])
10 26     } else {
11 27         B ← exp( log(B) + eta*(log(B0) - log(B)) + rnorm(1, 0, berr
12 28         ))
13 29     }
14 30
15 31     BSI ← B*S*I
16 32     rI ← r*I
17 33
18 34     dS ← -BSI
19 35     dI ← BSI - rI
20 36     dR ← rI
21 37
22 38     S ← S + h*dS #newInf
23 39     I ← I + h*dI #newInf - h*dR
24 40     R ← R + h*dR #h*dR
25 41
26 42     if (i %% steps == 0)
27 43         out[i/steps+1,] ← c(S,I,R,B)
28 44
29 45 }
30 46
31 47 return(out)
32 48
33 49 }

```

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) # resuceptibility 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 37             ), 1, E+1)
31 38         B ← as.matrix(preds * weights)
32 39
33 40         # solve system for C
34 41
35 42         Asvd ← svd(A)
36 43         C ← Asvd$v %*% diag(1/Asvd$d) %*% t(Asvd$u) %*% B
37 44
38 45         # get prediction
39 46
40 47         predsum ← sum(C * c(1,predictee))
41 48
42 49         # save
43 50
44 51         predictions[i] ← predsum
45 52
46 53         # next predictee
47 54
48 55         #predictee ← c( predsum, predictee[-E] )
49 56         #allPredictees[i,] ← predictee
50 57
51 58     }

```



```

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 27               r = 0.1, # recovery rate
38 28               N = 500, # population size
39 29               eta = 0.5, # geometric random walk
40 30               berr = 0.5, # Beta geometric walk noise
41 31               re = 1) # resusceptibility rate
42 32
43 33 true_init_cond ← c(S = N - i_infec,
44 34                  I = i_infec,
45 35                  R = 0)
46 36
47 37 ## trial parameter values to check.options

```

```

1 38 ##
2 39 Elist ← 1:20
3 40 thetalist ← 10*exp(-(seq(0,9.5,0.5)))
4 41 nTrials ← 100
5 42
6 43 ssemat ← matrix(NA, 20, 20)
7 44
8 45 for (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 79 quartz()
45 80 image(-ssemat)
46 81 quartz()
47 82 filled.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 1
18 2 data {
19 3
20 4   int      <lower=1>    T;      // total integration steps
21 5   real     y[T];       // observed number of cases
22 6   int      <lower=1>    N;      // population size
23 7   real     h;          // step size
24 8
25 9 }
26 10
27 11 parameters {
28 12
29 13   real <lower=0, upper=10>    R0;      // R0
30 14   real <lower=0, upper=10>    r;       // recovery rate
31 15   real <lower=0, upper=10>    re;      // resusceptibility rate
32 16   real <lower=0, upper=20>    sigma;   // observation error
33 17   real <lower=0, upper=30>    Iinit;   // initial infected
34 18   real <lower=0, upper=1>     eta;     // geometric walk
35 19   attraction strength
36 20   real <lower=0, upper=1>     berr;    // beta walk noise
37 21   real <lower=-1.5, upper=1.5> Bnoise[T]; // Beta vector
38 22
39 23 }
40 24
41 25 //transformed parameters {
42 26 //   real B0 ← R0 * r / N;
43 27 //}
44 28
45 29 model {
46 30
47 31   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 1 /* Author: Dexter Barrows

```

```

1  2      Github: dbarrows.github.io
2  3
3  4      */
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 etatrue    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      parameters
30 31
31 32 #include <Rcpp.h>
32 33 using namespace Rcpp;
33 34
34 35 struct State {
35 36     double S;
36 37     double I;
37 38     double R;
38 39 };
39 40
40 41 struct Particle {
41 42     double R0;
42 43     double r;
43 44     double re;
44 45     double sigma;
45 46     double eta;
46 47     double berr;
47 48     double B;
48 49     double S;
49 50     double I;
50 51     double R;
51 52     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 Sinitd;
13 64     double Iinitmean;   double Iinitd;
14 65     double Rinitmean;   double Rinitd;
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
10 104             berrcan;
11 105
12 106         do {
13 107             R0can = R0true + R0true*randn();
14 108         } while (R0can < 0);
15 109         particles[n].R0 = R0can;
16 110
17 111         do {
18 112             rcan = rtrue + rtrue*randn();
19 113         } while (rcan < 0);
20 114         particles[n].r = rcan;
21 115
22 116         do {
23 117             recan = retrue + retrue*randn();
24 118         } while (recan < 0);
25 119         particles[n].re = recan;
26 120
27 121         particles[n].B = (double) R0can * rcan / N;
28 122
29 123         do {
30 124             sigmacan = merr + merr*randn();
31 125         } while (sigmacan < 0);
32 126         particles[n].sigma = sigmacan;
33 127
34 128         do {
35 129             etacan = etatrue + PSC*etatrue*randn();
36 130         } while (etacan < 0 || etacan > 1);
37 131         particles[n].eta = etacan;
38 132
39 133         do {
40 134             berrcan = berrtrue + PSC*berrtrue*randn();
41 135         } while (berrcan < 0);
42 136         particles[n].berr = berrcan;
43 137
44 138         do {
45 139             Iinitcan = I0 + I0*randn();
46 140         } while (Iinitcan < 0 || N < Iinitcan);
47 141         particles[n].Sinit = N - Iinitcan;
48 142         particles[n].Iinit = Iinitcan;
49 143         particles[n].Rinit = 0.0;
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;
20 165
21 166     }
22 167
23 168     if (pass == (nPasses-1)) {
24 169         State sMeans;
25 170         getStateMeans(&sMeans, particles, NP);
26 171         statemeans(0,0) = sMeans.S;
27 172         statemeans(0,1) = sMeans.I;
28 173         statemeans(0,2) = sMeans.R;
29 174     }
30 175
31 176     for (int t = 1; t < T; t++) {
32 177
33 178         // generate individual predictions and weight
34 179         for (int n = 0; n < NP; n++) {
35 180
36 181             exp_euler_SIRS(1.0/7.0, (double) t-1, (double) t, N,
37 182                 &particles[n]);
38 183
39 184             double merr_par = particles[n].sigma;
40 185             double y_diff = data[t] - particles[n].I;
41 186
42 187             w[n] = 1.0/(merr_par*sqrt(2.0*M_PI)) * exp( - y_diff*
43 188                 y_diff / (2.0*merr_par*merr_par) );
44 189
45 190         }
46 191
47 192         // cumulative sum
48 193         for (int n = 1; n < NP; n++) {
49 194             w[n] += w[n-1];
50 195         }
51 196

```



```

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.sigamean;
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.Sinitd;
4 247         sds(pass, 7) = pInfo.Iinitd;
5 248         sds(pass, 8) = pInfo.Rinitd;
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.sigamean, 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.Sinitd);
25 268     printf("I_init      %f %f\n", pInfo.Iinitmean, pInfo.Iinitd);
26 269     printf("R_init      %f %f\n", pInfo.Rinitmean, pInfo.Rinitd);
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 double h      - time step size
21 double t0     - start time
22 double tn     - stop time
23 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 pertubation function to be run between iterations and
26 366 passes
27 366
28 367 */
29 368 void perturbParticles(Particle * particles, int N, int NP, int
30 369 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 * etatrue / 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 /* Convenience 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 *
10 450     particles, int NP) {
11 451
12 451     double   R0mean      = 0.0,
13 452             rmean       = 0.0,
14 453             remean       = 0.0,
15 454             sigmamean    = 0.0,
16 455             etamean      = 0.0,
17 456             berrmean     = 0.0,
18 457             Sinitmean    = 0.0,
19 458             Iinitmean    = 0.0,
20 459             Rinitmean    = 0.0;
21 460
22 461     // means
23 462
24 463     for (int n = 0; n < NP; n++) {
25 464
26 465         R0mean      += particles[n].R0;
27 466         rmean       += particles[n].r;
28 467         remean      += particles[n].re;
29 468         etamean     += particles[n].eta,
30 469         berrmean    += particles[n].berr,
31 470         sigmamean   += particles[n].sigma;
32 471         Sinitmean   += particles[n].Sinit;
33 472         Iinitmean   += particles[n].Iinit;
34 473         Rinitmean   += particles[n].Rinit;
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         Iinitd     = 0.0,
7 497         Rinitd     = 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                    R0mean );
13 502         rsd       += ( particles[n].r - rmean ) * ( particles[n].r -
14                    rmean );
15 503         resd      += ( particles[n].re - rmean ) * ( particles[n].re -
16                    rmean );
17 504         sigmasd   += ( particles[n].sigma - sigmamean ) * ( particles[n]
18                    ].sigma - sigmamean );
19 505         etasd     += ( particles[n].eta - etamean ) * ( particles[n].
20                    eta - etamean );
21 506         berrsd    += ( particles[n].berr - berrmean ) * ( particles[n].
22                    berr - berrmean );
23 507         Sinitsd   += ( particles[n].Sinit - Sinitmean ) * ( particles[n]
24                    ].Sinit - Sinitmean );
25 508         Iinitd    += ( particles[n].Iinit - Iinitmean ) * ( particles[n]
26                    ].Iinit - Iinitmean );
27 509         Rinitd    += ( particles[n].Rinit - Rinitmean ) * ( particles[n]
28                    ].Rinit - Rinitmean );
29 510
30 511     }
31 512
32 513     R0sd        /= NP;
33 514     rsd          /= NP;
34 515     resd         /= NP;
35 516     sigmasd      /= NP;
36 517     etasd        /= NP;
37 518     berrsd       /= NP;
38 519     Sinitsd      /= NP;
39 520     Iinitd       /= NP;
40 521     Rinitd       /= NP;
41 522
42 523     partInfo->R0mean    = R0mean;
43 524     partInfo->R0sd      = R0sd;
44 525     partInfo->rmean     = rmean;
45 526     partInfo->rsd       = rsd;
46 527     partInfo->remean    = remean;
47 528     partInfo->resd      = resd;
48 529     partInfo->sigmamean = sigmamean;
49 530     partInfo->sigmasd   = sigmasd;
50 531     partInfo->etamean   = etamean;
51 532     partInfo->etasd     = etasd;

```

```

1 533     partInfo->berrmean  = berrmean;
2 534     partInfo->berrsd   = berrsd;
3 535     partInfo->Sinitmean = Sinitmean;
4 536     partInfo->Sinitd    = Sinitd;
5 537     partInfo->Iinitmean = Iinitmean;
6 538     partInfo->Iinitd    = Iinitd;
7 539     partInfo->Rinitmean = Rinitmean;
8 540     partInfo->Rinitd    = Rinitd;
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
36 568    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 11
19 12     ## number of locations
20 13     nloc ← dim(ymat)[1]
21 14
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 22
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 47         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         nBIsu ← B[neibmat[loc, 1:n]] %*% I[neibmat[loc, 1:n]]
28 53         BSI[loc] ← S[loc]*( sphi*B[loc]*I[loc] + ophi*nBIsu )
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 1 | data {
22 2 |
23 3 |   int      <lower=1>    T;      // total integration steps
24 4 |   int      <lower=1>    nloc;   // number of locations
25 5 |   real     y[nloc, T];  // observed number of cases
26 6 |   int      <lower=1>    N;      // population size
27 7 |   real     h;           // step size
28 8 |   int      <lower=0>    neinum[nloc]; // number of neighbors
29 9 |   each location has
30 10 |   int      neibmat[nloc, nloc]; // neighbor list for
31 11 |   each location
32 12 | }
33 13 |
34 14 | parameters {
35 15 |
36 16 |   real <lower=0, upper=10>    R0;      // R0
37 17 |   real <lower=0, upper=10>    r;       // recovery rate
38 18 |   real <lower=0, upper=20>    sigma;   // observation error
39 19 |   real <lower=0, upper=30>    linit[nloc]; // initial
40 20 |   infected for each location
41 21 |   real <lower=0, upper=1>    eta;      // geometric walk
42 22 |   attraction strength
43 23 |   real <lower=0, upper=1>    berr;     // beta walk noise
44 24 |   real <lower=-1.5, upper=1.5> Bnoise[nloc, T]; // Beta vector

```

```

1 22      real <lower=0, upper=1>          phi;    // interconnectivity
2      strength
3 23
4 24 }
5 25
6 26 model {
7 27
8 28     real S[nloc, T];
9 29     real I[nloc, T];
10 30     real R[nloc, T];
11 31     real B[nloc, T];
12 32     real B0;
13 33
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 nBIsun;
20 40
21 41     B0 ← R0 * r / N;
22 42
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 49
30 50     for (t in 2:T) {
31 51         for (loc in 1:nloc) {
32 52
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 55
37 56             n ← neinum[loc];
38 57             sphi ← 1.0 - phi*( n/(n+1.0) );
39 58             ophi ← phi/(n+1.0);
40 59
41 60             nBIsun ← 0.0;
42 61             for (j in 1:n)
43 62                 nBIsun ← nBIsun + B[neibmat[loc, j], t-1] * I[neibmat
44                 [loc, j], t-1];
45 63
46 64             BSI[loc, t] ← S[loc, t-1]*( sphi*B[loc, t-1]*I[loc, t-1]
47             + ophi*nBIsun );
48 65             rI[loc, t] ← r*I[loc, t-1];
49 66
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 etatrue     0.5          // real drift attraction strength
45 24 #define berrtrue    0.5          // real beta drift noise
46 25 #define phitrue     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
17 175 if (t < (T-1))
18 176     perturbParticles(particles, N, NP, nloc, pass,
19 177                     coolrate);
20 178
21 179 if (pass == (nPasses-1)) {
22 180     double means[nloc];
23 181     for (int loc = 0; loc < nloc; loc++) {
24 182         means[loc] = 0.0;
25 183         for (int n = 0; n < NP; n++) {
26 184             means[loc] += particles[n].I[loc] / NP;
27 185         }
28 186         infecmeans(loc, t) = means[loc];
29 187     }
30 188 }
31 189
32 190 // between-pass perturbations, not after last pass
33 191 if (pass < (nPasses + 1))
34 192     perturbParticles(particles, N, NP, nloc, pass, coolrate);
35 193
36 194 }
37 195
38 196 // pack parameter data (minus initial conditions)
39 197 for (int n = 0; n < NP; n++) {
40 198     paramdata(n, 0) = particles[n].R0;
41 199     paramdata(n, 1) = particles[n].r;
42 200     paramdata(n, 2) = particles[n].sigma;
43 201     paramdata(n, 3) = particles[n].eta;
44 202     paramdata(n, 4) = particles[n].berr;
45 203     paramdata(n, 5) = particles[n].phi;
46 204 }
47 205
48 206 // Pack initial condition data
49 207 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], Imean[nloc], Rmeans[nloc], Bmeans[nloc];
8 215 for (int loc = 0; loc < nloc; loc++) {
9 216     Smeans[loc] = 0.0;
10 217     Imean[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         Imean[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) = Imean[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                               infecmeans,
30 236                           Rcpp::Named("finalstate") =
31                               finalstate);
32 237
33 238
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                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 particle,
49 NumericVector neinum, NumericMatrix neibmat, int
50 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\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
26 278         for (int loc = 0; loc < nloc; loc++) {
27 279             S_last[loc] = S[loc];
28 280             I_last[loc] = I[loc];
29 281             R_last[loc] = R[loc];
30 282             B_last[loc] = B[loc];
31 283         }
32 284
33 285         for (int loc = 0; loc < nloc; loc++) {
34 286
35 287             B[loc] = exp( log(B_last[loc]) + eta*(log(B0) - log(
36 288                 B_last[loc])) + berr*randn() );
37 289
38 290             int n = neinum[loc];
39 291             double sphi = 1.0 - phi*( (double) n/(n+1.0) );
40 292             double ophi = phi/(n+1.0);
41 293
42 294             double nBIsum = 0.0;
43 295             for (int j = 0; j < n; j++)
44 296                 nBIsum += B_last[(int) neibmat(loc, j) - 1] * I_last
45 297                     [(int) neibmat(loc, j) - 1];
46 298
47 299             double BSI = S_last[loc]*( sphi*B_last[loc]*I_last[loc] +
48 300                 ophi*nBIsum );
49 301             double rI  = r*I_last[loc];
50 302
51 303             // 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%f\t%f\t"
12         //         "%f\t\n", sphi, ophi, BSI, rI, dS, dI, dR, S[1], I
13         //         [1], R[1]);
14 311
15 312     }
16 313
17 314 }
18 315
19 316 /*particle->S = S;
20 317 particle->I = I;
21 318 particle->R = R;
22 319 particle->B = B;*/
23 320
24 321 }
25 322
26 323 /* Initializes particles
27 324 */
28 325 void initializeParticles(Particle ** particles, int NP, int nloc, int
29 N) {
30 326
31 327     // allocate space for doubles
32 328     *particles = (Particle*) malloc (NP*sizeof(Particle));
33 329
34 330     // allocate space for arrays inside particles
35 331     for (int n = 0; n < NP; n++) {
36 332         (*particles)[n].S = (double*) malloc(nloc*sizeof(double));
37 333         (*particles)[n].I = (double*) malloc(nloc*sizeof(double));
38 334         (*particles)[n].R = (double*) malloc(nloc*sizeof(double));
39 335         (*particles)[n].B = (double*) malloc(nloc*sizeof(double));
40 336         (*particles)[n].Iinit = (double*) malloc(nloc*sizeof(double))
41         ;
42 337     }
43 338
44 339     // initialize all all parameters
45 340     for (int n = 0; n < NP; n++) {
46 341
47 342         double R0can, rcan, sigmacan, Iinitcan, etacan, berrcan,
48         phican;
49 343
50 344         do {
51 345             R0can = R0true + R0true*randn();

```

```

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         sigma = merr + merr*randn();
14 359     } while (sigma < 0);
15 360     (*particles)[n].sigma = sigma;
16 361
17 362     do {
18 363         etacan = etatrue + PSC*etatrue*randn();
19 364     } while (etacan < 0 || etacan > 1);
20 365     (*particles)[n].eta = etacan;
21 366
22 367     do {
23 368         berrcan = berrtrue + PSC*berrtrue*randn();
24 369     } while (berrcan < 0);
25 370     (*particles)[n].berr = berrcan;
26 371
27 372     do {
28 373         phican = phitrue + PSC*phitrue*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 pertubation 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
2 396 double spreadR0      = coolcoef * R0true / 10.0;
3 397 double spreadr       = coolcoef * rtrue / 10.0;
4 398 double spreadsigma    = coolcoef * merr / 10.0;
5 399 double spreadIinit    = coolcoef * I0 / 10.0;
6 400 double spreadeta      = coolcoef * etatrue / 10.0;
7 401 double spreadberr     = coolcoef * berrtrue / 10.0;
8 402 double spreadphi      = coolcoef * phitrue / 10.0;
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.

```

24
25 1  /* Author: Dexter Barrows
26 2  Github: dbarrows.github.io
27 3  */
28 4
29 5  /* Runs a particle filter on synthetic noisy data and attempts to
30 6  reconstruct underlying true state at each time step. Note that
31 7  this program uses gnuplot to plot the data, so an x11
32 8  environment must be present. Also the multiplier of 1024 in the
33 9  definition of NP below should be set to a multiple of the number
34 10 of multiprocessors of your GPU for optimal results.
35 11
36 12 Also, the accompanying "pf.plg" file contains the instructions
37 13 gnuplot will use. It must be present in the same directory as
38 14 the executable generated by compiling this file.
39 15
40 16 Compile with:
41 17
42 18 nvcc -arch=sm_20 -O2 pf_cuda.cu timer.cpp rand.cpp -o pf_cuda.x
43 19
44 20 */
45 21
46 22 #include <cuda.h>
47 23 #include <iostream>

```

```

1 24 #include <fstream>
2 25 #include <curand.h>
3 26 #include <curand_kernel.h>
4 27 #include <string>
5 28 #include <sstream>
6 29 #include <cmath>
7 30
8 31 #include "timer.h"
9 32 #include "rand.h"
10 33 #include "readdata.h"
11 34
12 35 #define NP          (2*2500)    // number of particles
13 36 #define N           500.0      // population size
14 37 #define R0true      3.0        // infectiousness
15 38 #define rtrue       0.1        // recovery rate
16 39 #define etatrue     0.5        // real drift attraction strength
17 40 #define berrtrue    0.5        // real beta drift noise
18 41 #define phitrue     0.5        // real connectivity strength
19 42 #define merr        10.0       // expected measurement error
20 43 #define I0          5.0        // Initial infected individuals
21 44 #define PSC         0.5        // sensitive parameter perturbation
22    scaling
23 45 #define NLOC        10
24 46
25 47 #define PI           3.141592654f
26 48
27 49 // Wrapper for CUDA calls, from CUDA API
28 50 // Modified to also print the error code and string
29 51 # define CUDA_CALL(x) do { if ((x) != cudaSuccess ) {
30    \
31 52    std::cout << " Error at " << __FILE__ << ":" << __LINE__ << std::
32    endl;    \
33 53    std::cout << " Error was " << x << " " << cudaGetErrorString(x)
34    << std::endl;    \
35 54    return EXIT_FAILURE ;}} while (0)
36    \
37 55
38 56 typedef struct {
39 57     float R0;
40 58     float r;
41 59     float sigma;
42 60     float eta;
43 61     float berr;
44 62     float phi;
45 63     /*
46 64     float * S;
47 65     float * I;
48 66     float * R;
49 67     float * B;
50 68     float * Iinit;
51 69     */

```

```

1  70     float S[NLOC];
2  71     float I[NLOC];
3  72     float R[NLOC];
4  73     float B[NLOC];
5  74     float Iinit[NLOC];
6  75     curandState randState; // PRNG state
7  76 } Particle;
8  77
9  78 __host__ std::string getHRmemsize (size_t memsize);
10 79 __host__ std::string getHRtime (float runtime);
11 80
12 81 __device__ void exp_euler_SSIR(float h, float t0, float tn, Particle
13     * particle, int * neinum, int * neibmat, int nloc);
14 82 __device__ void copyParticle(Particle * dst, Particle * src, int nloc
15     );
16 83
17 84
18 85 /* Initialize all PRNG states, get starting state vector using
19     initial distribution
20 86 */
21 87 __global__ void initializeParticles (Particle * particles, int nloc)
22     {
23 88
24 89     int id = blockIdx.x*blockDim.x + threadIdx.x; // global thread
25         ID
26 90
27 91     if (id < NP) {
28 92
29 93         // initialize PRNG state
30 94         curandState state;
31 95         curand_init(id, 0, 0, &state);
32 96
33 97         // allocate space for arrays inside particle
34 98         //particles[id].S = (float*) malloc(nloc*sizeof(float));
35 99         //particles[id].I = (float*) malloc(nloc*sizeof(float));
36 100        //particles[id].R = (float*) malloc(nloc*sizeof(float));
37 101        //particles[id].B = (float*) malloc(nloc*sizeof(float));
38 102        //particles[id].Iinit = (float*) malloc(nloc*sizeof(float));
39 103
40 104        // initialize all parameters
41 105
42 106        float R0can, rcan, sigmacan, Iinitcan, etacan, berrcan,
43            phican;
44 107
45 108        do {
46 109            R0can = R0true + R0true*curand_normal(&state);
47 110        } while (R0can < 0);
48 111        particles[id].R0 = R0can;
49 112
50 113        do {
51 114            rcan = rtrue + rtrue*curand_normal(&state);

```

```

1 115     } while (rcan < 0);
2 116     particles[id].r = rcan;
3 117
4 118     for (int loc = 0; loc < nloc; loc++)
5 119         particles[id].B[loc] = (float) R0can * rcan / N;
6 120
7 121     do {
8 122         sigmacan = merr + merr*curand_normal(&state);
9 123     } while (sigmacan < 0);
10 124     particles[id].sigma = sigmacan;
11 125
12 126     do {
13 127         etacan = etatrue + PSC*etatrue*curand_normal(&state);
14 128     } while (etacan < 0 || etacan > 1);
15 129     particles[id].eta = etacan;
16 130
17 131     do {
18 132         berrcan = berrtrue + PSC*berrtrue*curand_normal(&state);
19 133     } while (berrcan < 0);
20 134     particles[id].berr = berrcan;
21 135
22 136     do {
23 137         phican = phitrue + PSC*phitrue*curand_normal(&state);
24 138     } while (phican <= 0 || phican >= 1);
25 139     particles[id].phi = phican;
26 140
27 141     for (int loc = 0; loc < nloc; loc++) {
28 142         do {
29 143             Iinitcan = I0 + I0*curand_normal(&state);
30 144         } while (Iinitcan < 0 || N < Iinitcan);
31 145         particles[id].Iinit[loc] = Iinitcan;
32 146     }
33 147
34 148     particles[id].randState = state;
35 149
36 150 }
37 151
38 152 }
39 153
40 154 __global__ void resetStates (Particle * particles, int nloc) {
41 155
42 156     int id = blockIdx.x*blockDim.x + threadIdx.x; // global thread
43 157     ID
44 157
45 158     for (int loc = 0; loc < nloc; loc++) {
46 159         particles[id].S[loc] = N - particles[id].Iinit[loc];
47 160         particles[id].I[loc] = particles[id].Iinit[loc];
48 161         particles[id].R[loc] = 0.0;
49 162     }
50 163
51 164 }

```

```

1 165
2 166 __global__ void clobberParams (Particle * particles, int nloc) {
3 167
4 168     int id = blockIdx.x*blockDim.x + threadIdx.x; // global thread
5 169     ID
6 169
7 170     particles[id].R0 = R0true;
8 171     particles[id].r = rtrue;
9 172     particles[id].sigma = merr;
10 173     particles[id].eta = etatrue;
11 174     particles[id].berr = berrtrue;
12 175     particles[id].phi = phitrue;
13 176
14 177     for (int loc = 0; loc < nloc; loc++) {
15 178         particles[id].Iinit[loc] = I0;
16 179     }
17 180
18 181 }
19 182 }
20 183
21 184
22 185 /* Project particles forward, perturb, and save weight based on data
23 186    int t - time step number (1,...,T)
24 187    */
25 188 __global__ void project (Particle * particles, int * neinum, int *
26 189     neibmat, int nloc) {
27 189
28 190     int id = blockIdx.x*blockDim.x + threadIdx.x; // global id
29 191
30 192     if (id < NP) {
31 193         // project forward
32 194         exp_euler_SSIR(1.0/7.0, 0.0, 1.0, &particles[id], neinum,
33 195         neibmat, nloc);
34 195     }
35 196
36 197 }
37 198
38 199 __global__ void weight(float * data, Particle * particles, double * w
39 200     , int t, int T, int nloc) {
40 200
41 201     int id = blockIdx.x*blockDim.x + threadIdx.x; // global id
42 202
43 203     if (id < NP) {
44 204
45 205         float merr_par = particles[id].sigma;
46 206
47 207         // Get weight and save
48 208         double w_local = 1.0;
49 209         for (int loc = 0; loc < nloc; loc++) {
50 210             float y_diff = data[loc*T + t] - particles[id].I[loc];
51 211             w_local *= 1.0/(merr_par*sqrt(2.0*PI)) * exp( - y_diff*

```

```

1          y_diff / (2.0*merr_par*merr_par) );
2 212      }
3 213
4 214      w[id] = w_local;
5 215
6 216      }
7 217
8 218 }
9 219
10 220 __global__ void stashParticles (Particle * particles, Particle *
11      particles_old, int nloc) {
12 221
13 222     int id = blockIdx.x*blockDim.x + threadIdx.x;    // global id
14 223
15 224     if (id < NP) {
16 225         // COPY PARTICLE
17 226         copyParticle(&particles_old[id], &particles[id], nloc);
18 227     }
19 228
20 229 }
21 230
22 231
23 232 /* The 0th thread will perform cumulative sum on the weights.
24 233    There may be a faster way to do this, will investigate.
25 234    */
26 235 __global__ void cumsumWeights (double * w) {
27 236
28 237     int id  = blockIdx.x*blockDim.x + threadIdx.x;    // global thread
29      ID
30 238
31 239     // compute cumulative weights
32 240     if (id == 0) {
33 241         for (int i = 1; i < NP; i++)
34 242             w[i] += w[i-1];
35 243     }
36 244
37 245 }
38 246
39 247
40 248 /* Resample from all particle states within cell
41 249    */
42 250 __global__ void resample (Particle * particles, Particle *
43      particles_old, double * w, int nloc) {
44 251
45 252     int id  = blockIdx.x*blockDim.x + threadIdx.x;
46 253
47 254     if (id < NP) {
48 255
49 256         // resampling proportional to weights
50 257         double w_r = curand_uniform(&particles[id].randState) * w[NP
51      -1];

```

```

1 258         int i = 0;
2 259         while (w_r > w[i]) {
3 260             i++;
4 261         }
5 262
6 263         // i is now the index of the particle to copy from
7 264         copyParticle(&particles[id], &particles_old[i], nloc);
8 265
9 266     }
10 267
11 268 }
12 269
13 270 // launch this with probably just nloc threads... block structure/
14 271 // size probably not important
15 271 __global__ void reduceStates (Particle * particles, float *
16 272 countmeans, int t, int T, int nloc) {
17 272
18 273     int id = blockIdx.x*blockDim.x + threadIdx.x;
19 274
20 275     if (id < nloc) {
21 276
22 277         int loc = id;
23 278
24 279         double countmean_local = 0.0;
25 280         for (int n = 0; n < NP; n++) {
26 281             countmean_local += particles[n].I[loc] / NP;
27 282         }
28 283
29 284         countmeans[loc*T + t] = (float) countmean_local;
30 285
31 286     }
32 287
33 288 }
34 289
35 290 __global__ void perturbParticles(Particle * particles, int nloc, int
36 291 passnum, double coolrate) {
37 291
38 292     //double coolcoef = exp( - (double) passnum / coolrate );
39 293     double coolcoef = pow(coolrate, passnum);
40 294
41 295     double spreadR0 = coolcoef * R0true / 10.0;
42 296     double spreadr = coolcoef * rtrue / 10.0;
43 297     double spreadsigma = coolcoef * merr / 10.0;
44 298     double spreadIinit = coolcoef * I0 / 10.0;
45 299     double spreadeta = coolcoef * etatrue / 10.0;
46 300     double spreadberr = coolcoef * berrtrue / 10.0;
47 301     double spreadphi = coolcoef * phitrue / 10.0;
48 302
49 303     double R0can, rcan, sigmacan, Iinitcan, etacan, berrcan, phican;
50 304
51 305     int id = blockIdx.x*blockDim.x + threadIdx.x;

```

```

1 306
2 307   if (id < NP) {
3 308
4 309       do {
5 310           R0can = particles[id].R0 + spreadR0*curand_normal(&
6           particles[id].randState);
7 311       } while (R0can < 0);
8 312       particles[id].R0 = R0can;
9 313
10 314       do {
11 315           rcan = particles[id].r + spreadr*curand_normal(&particles
12           [id].randState);
13 316       } while (rcan < 0);
14 317       particles[id].r = rcan;
15 318
16 319       do {
17 320           sigmacan = particles[id].sigma + spreadsigma*
18           curand_normal(&particles[id].randState);
19 321       } while (sigmacan < 0);
20 322       particles[id].sigma = sigmacan;
21 323
22 324       do {
23 325           etacan = particles[id].eta + PSC*spreadeta*curand_normal
24           (&particles[id].randState);
25 326       } while (etacan < 0 || etacan > 1);
26 327       particles[id].eta = etacan;
27 328
28 329       do {
29 330           berrcan = particles[id].berr + PSC*spreadberr*
30           curand_normal(&particles[id].randState);
31 331       } while (berrcan < 0);
32 332       particles[id].berr = berrcan;
33 333
34 334       do {
35 335           phican = particles[id].phi + PSC*spreadphi*curand_normal
36           (&particles[id].randState);
37 336       } while (phican <= 0 || phican >= 1);
38 337       particles[id].phi = phican;
39 338
40 339       for (int loc = 0; loc < nloc; loc++) {
41 340           do {
42 341               Iinitcan = particles[id].Iinit[loc] + spreadIinit*
43               curand_normal(&particles[id].randState);
44 342           } while (Iinitcan < 0 || Iinitcan > 500);
45 343           particles[id].Iinit[loc] = Iinitcan;
46 344       }
47 345
48 346   }
49 347
50 348 }
51 349

```



```

1 350
2 351 int main (int argc, char *argv[]) {
3 352
4 353
5 354     int T, nloc;
6 355
7 356     double restime;
8 357     struct timeval tdr0, tdr1, tdrMaster;
9 358
10 359     gettimeofday (&tdr0, NULL);
11 360
12 361
13 362     // Parse arguments *****
14 363
15 364     if (argc < 4) {
16 365         std::cout << "Not enough arguments" << std::endl;
17 366         return 0;
18 367     }
19 368
20 369     std::string arg1(argv[1]); // infection counts
21 370     std::string arg2(argv[2]); // neighbour counts
22 371     std::string arg3(argv[3]); // neighbour indices
23 372
24 373     std::cout << "Arguments:" << std::endl;
25 374     std::cout << "Infection data: " << arg1 << std::endl;
26 375     std::cout << "Neighbour counts: " << arg2 << std::endl;
27 376     std::cout << "Neighbour indices: " << arg3 << std::endl;
28 377
29 378     // *****
30 379
31 380
32 381     // Read count data *****
33 382
34 383     std::cout << "Getting count data" << std::endl;
35 384     float * data = getDataFloat(arg1, &T, &nloc);
36 385     size_t datasize = nloc*T*sizeof(float);
37 386
38 387     // *****
39 388
40 389     // Read neinum matrix data *****
41 390
42 391     std::cout << "Getting neighbour count data" << std::endl;
43 392     int * neinum = getDataInt(arg2, NULL, NULL);
44 393     size_t neinumsize = nloc * sizeof(int);
45 394
46 395     // *****
47 396
48 397     // Read neibmat matrix data *****
49 398
50 399     std::cout << "Getting neighbour count data" << std::endl;
51 400     int * neibmat = getDataInt(arg3, NULL, NULL);

```

```

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

```

```

1 439
2 440     std::cout << "\t" << getHRtime(restime) << std::endl;
3 441
4 442     size_t avail, total;
5 443     cudaMemGetInfo( &avail, &total );
6 444     size_t used = total - avail;
7 445
8 446     std::cout << "\t[" << getHRmemsize(used) << "]" used of [" <<
9         getHRmemsize(total) << "]" <<std::endl;
10 447
11 448     std::cout << "Copying data to device" << std::endl;
12 449
13 450     gettimeofday (&tdr0, NULL);
14 451
15 452     CUDA_CALL( cudaMemcpy(d_data      , data      , datasize      ,
16         cudaMemcpyHostToDevice) );
17 453     CUDA_CALL( cudaMemcpy(d_neinum    , neinum     , neinumsize    ,
18         cudaMemcpyHostToDevice) );
19 454     CUDA_CALL( cudaMemcpy(d_neibmat   , neibmat    , neibmatsize    ,
20         cudaMemcpyHostToDevice) );
21 455
22 456     gettimeofday (&tdr1, NULL);
23 457     timeval_subtract (&restime, &tdr1, &tdr0);
24 458
25 459     std::cout << "\t" << getHRtime(restime) << std::endl;
26 460
27 461     // *****
28 462
29 463
30 464
31 465     // Initialize particles *****
32 466
33 467     std::cout << "Initializing particles" << std::endl;
34 468
35 469     gettimeofday (&tdr0, NULL);
36 470
37 471     int nThreads      = 32;
38 472     int nBlocks       = ceil( (float) NP / nThreads);
39 473
40 474     initializeParticles <<< nBlocks, nThreads >>> (particles, nloc);
41 475     CUDA_CALL( cudaGetLastError() );
42 476     CUDA_CALL( cudaDeviceSynchronize() );
43 477
44 478     initializeParticles <<< nBlocks, nThreads >>> (particles_old,
45         nloc);
46 479     CUDA_CALL( cudaGetLastError() );
47 480     CUDA_CALL( cudaDeviceSynchronize() );
48 481
49 482     gettimeofday (&tdr1, NULL);
50 483     timeval_subtract (&restime, &tdr1, &tdr0);
51 484

```

```

1 485     std::cout << "\t" << getHRtime(restime) << std::endl;
2 486
3 487     cudaMemGetInfo( &avail, &total );
4 488     used = total - avail;
5 489     std::cout << "\t[" << getHRmemsize(used) << "]" used of [" <<
6         getHRmemsize(total) << "]" <<std::endl;
7 490
8 491     // *****
9 492
10 493     // Starting filtering *****
11 494
12 495     for (int pass = 0; pass < 50; pass++) {
13 496
14 497         std::cout << "pass = " << pass << std::endl;
15 498
16 499         // ** TEMP **
17 500         //clobberParams <<< nBlocks, nThreads >>> (particles, nloc);
18 501         // ** TEMP **
19 502
20 503         nThreads      = 32;
21 504         nBlocks       = ceil( (float) NP / nThreads);
22 505
23 506         resetStates <<< nBlocks, nThreads >>> (particles, nloc);
24 507         CUDA_CALL( cudaGetLastError() );
25 508         CUDA_CALL( cudaDeviceSynchronize() );
26 509
27 510         std::cout << "Filtering over [1," << Tlim << "]"<< std::endl;
28 511
29 512         gettimeofday (&tdrMaster, NULL);
30 513
31 514         gettimeofday (&tdr0, NULL);
32 515
33 516         nThreads = 1;
34 517         nBlocks  = 10;
35 518
36 519         if (pass == 49) {
37 520             reduceStates <<< nBlocks, nThreads >>> (particles,
38                 d_countmeans, 0, T, nloc);
39 521             CUDA_CALL( cudaGetLastError() );
40 522             CUDA_CALL( cudaDeviceSynchronize() );
41 523         }
42 524
43 525         gettimeofday (&tdr1, NULL);
44 526         timeval_subtract (&restime, &tdr1, &tdr0);
45 527         std::cout << "Reduction          " << getHRtime(restime) << std
46             ::endl;
47 528
48 529         int Tlim = T;
49 530
50 531         for (int t = 1; t < Tlim; t++) {
51 532

```

```

1 533 // Projection
2
3 534 *****
4 535 nThreads    = 32;
5 536 nBlocks     = ceil( (float) NP / nThreads);
6 537
7 538 //if (t == 1)
8 539 //  gettimeofday (&tdr0, NULL);
9 540
10 541 project <<< nBlocks, nThreads >>> (particles, d_neinum,
11      d_neibmat, nloc);
12 542 CUDA_CALL( cudaGetLastError() );
13 543 CUDA_CALL( cudaDeviceSynchronize() );
14 544
15 545 //if (t == 1) {
16 546 //  gettimeofday (&tdr1, NULL);
17 547 //  timeval_subtract (&restime, &tdr1, &tdr0);
18 548 //  std::cout << "\tProjection " << getHRtime(restime) <<
19      std::endl;
20 549 //}
21 550
22 551 // Weighting
23      *****
24 552
25 553 nThreads    = 32;
26 554 nBlocks     = ceil( (float) NP / nThreads);
27 555
28 556 weight <<< nBlocks, nThreads >>>(d_data, particles, w, t,
29      T, nloc);
30 557 CUDA_CALL( cudaGetLastError() );
31 558 CUDA_CALL( cudaDeviceSynchronize() );
32 559
33 560 // Cumulative sum
34      *****
35 561
36 562 nThreads    = 1;
37 563 nBlocks     = 1;
38 564
39 565 if (t == 1)
40 566     gettimeofday (&tdr0, NULL);
41 567
42 568 cumsumWeights <<< nBlocks, nThreads >>> (w);
43 569 CUDA_CALL( cudaGetLastError() );
44 570 CUDA_CALL( cudaDeviceSynchronize() );
45 571
46 572 if (t == 1) {
47 573     gettimeofday (&tdr1, NULL);
48 574     timeval_subtract (&restime, &tdr1, &tdr0);
49 575     std::cout << "Cumulative sum " << getHRtime(restime)
50         << std::endl;
51 576 }

```

```

1 577
2 578 // Save particles for resampling from
3      *****
4 579
5 580 nThreads    = 32;
6 581 nBlocks     = ceil( (float) NP / nThreads);
7 582
8 583 stashParticles <<< nBlocks, nThreads >>> (particles,
9      particles_old, nloc);
10 584 CUDA_CALL( cudaGetLastError() );
11 585 CUDA_CALL( cudaDeviceSynchronize() );
12 586
13 587
14 588 // Resampling
15      *****
16 589
17 590 nThreads    = 32;
18 591 nBlocks     = ceil( (float) NP/ nThreads);
19 592
20 593 if (t == 1)
21 594     gettimeofday (&tdr0, NULL);
22 595
23 596 resample <<< nBlocks, nThreads >>> (particles,
24     particles_old, w, nloc);
25 597 CUDA_CALL( cudaGetLastError() );
26 598 CUDA_CALL( cudaDeviceSynchronize() );
27 599
28 600 if (t == 1) {
29 601     gettimeofday (&tdr1, NULL);
30 602     timeval_subtract (&restime, &tdr1, &tdr0);
31 603     std::cout << "\tResampling " << getHRtime(restime) <<
32     std::endl;
33 604 }
34 605
35 606 // Reduction
36      *****
37 607
38 608 //if (t == (Tlim-1)) {
39 609
40 610 if (pass == 49) {
41 611
42 612     if (t == 1)
43 613         gettimeofday (&tdr0, NULL);
44 614
45 615     nThreads = 1;
46 616     nBlocks  = 10;
47 617
48 618     reduceStates <<< nBlocks, nThreads >>> (particles,
49     d_countmeans, t, T, nloc);
50 619 CUDA_CALL( cudaGetLastError() );
51 620 CUDA_CALL( cudaDeviceSynchronize() );

```

```

1 621
2 622         if (t == 1) {
3 623             gettimeofday (&tdr1, NULL);
4 624             timeval_subtract (&restime, &tdr1, &tdr0);
5 625             std::cout << "Reduction" << getHRtime(
6             restime) << std::endl;
7 626         }
8 627
9 628     }
10 629
11 630     // Perturb particles
12     *****
13 631
14 632     nThreads      = 32;
15 633     nBlocks       = ceil( (float) NP/ nThreads);
16 634
17 635     perturbParticles <<< nBlocks, nThreads >>> (particles,
18         nloc, pass, 0.975);
19 636     CUDA_CALL( cudaGetLastError() );
20 637     CUDA_CALL( cudaDeviceSynchronize() );
21 638
22 639     //}
23 640     /*
24 641     nThreads      = RB_DIM;
25 642     nBlocks       = nCells;
26 643
27 644
28 645
29 646     reduce <<< nBlocks, nThreads >>> (d_E, t, particles,
30         Beta_last, nCells);
31 647     CUDA_CALL( cudaGetLastError() );
32 648     CUDA_CALL( cudaDeviceSynchronize() );
33 649
34 650     if (t == 1) {
35 651         gettimeofday (&tdr1, NULL);
36 652         timeval_subtract (&restime, &tdr1, &tdr0);
37 653         std::cout << "Reduction" << getHRtime(restime
38             ) << std::endl;
39 654     }
40 655     */
41 656
42 657
43 658     } // end time
44 659
45 660 } // end pass
46 661
47 662 std::cout.precision(10);
48 663
49 664 countmeans = (float*) malloc (nloc*T*sizeof(float));
50 665 cudaMemcpy(countmeans, d_countmeans, nloc*T*sizeof(float),
51     cudaMemcpyDeviceToHost);

```

```

1 666
2 667     std::string filename = "cuIF2states.dat";
3 668
4 669     std::cout << "Writing results to file '" << filename << "' ..."
5     << std::endl;
6 670
7 671     std::ofstream outfile;
8 672     outfile.open(filename.c_str());
9 673
10 674     for(int loc = 0; loc < nloc; loc++) {
11 675         for (int t = 0; t < T; t++) {
12 676             outfile << countmeans[loc*T + t] << " ";
13 677         }
14 678         outfile << std::endl;
15 679     }
16 680
17 681     /*
18 682     double * h_w = (double*) malloc (NP*sizeof(double));
19 683     cudaMemcpy(h_w, w, NP*sizeof(double), cudaMemcpyDeviceToHost);
20 684
21 685     for (int n = 0; n < NP; n++) {
22 686         std::cout << h_w[n] << " ";
23 687     }
24 688     */
25 689
26 690     /*
27 691     for (int i = 0; i < nCells; i++) {
28 692         outfile << trueCounts[t*nCells + i];
29 693         if (i % dim == 0)
30 694             outfile << std::endl;
31 695         else
32 696             outfile << " ";
33 697     }
34 698     */
35 699
36 700     outfile.close();
37 701
38 702     gettimeofday (&tdr1, NULL);
39 703     timeval_subtract (&restime, &tdr1, &tdrMaster);
40 704     std::cout << "Total PF time (excluding setup) " << getHRtime(
41     restime) << std::endl;
42 705
43 706     cudaFree(d_data);
44 707     cudaFree(particles);
45 708     cudaFree(particles_old);
46 709     cudaFree(w);
47 710     cudaFree(d_neinum);
48 711     cudaFree(d_neibmat);
49 712     cudaFree(d_countmeans);
50 713
51 714     exit (EXIT_SUCCESS);

```



```

1 715
2 716 }
3 717
4 718
5 719 /* Use the Explicit Euler integration scheme to integrate SIR model
6      forward in time
7 720     float h      - time step size
8 721     float t0     - start time
9 722     float tn     - stop time
10 723     float * y    - current system state; a three-component vector
11                  representing [S I R], susceptible-infected-recovered
12 724 */
13 725 __device__ void exp_euler_SSIR(float h, float t0, float tn, Particle
14      * particle, int * neinum, int * neibmat, int nloc) {
15 726
16 727     int num_steps = floor( (tn-t0) / h );
17 728
18 729     float * S = particle->S;
19 730     float * I = particle->I;
20 731     float * R = particle->R;
21 732     float * B = particle->B;
22 733
23 734     // create last state vectors
24 735     float * S_last = (float*) malloc (nloc*sizeof(float));
25 736     float * I_last = (float*) malloc (nloc*sizeof(float));
26 737     float * R_last = (float*) malloc (nloc*sizeof(float));
27 738     float * B_last = (float*) malloc (nloc*sizeof(float));
28 739
29 740     float R0      = particle->R0;
30 741     float r       = particle->r;
31 742     float B0      = R0 * r / N;
32 743     float eta     = particle->eta;
33 744     float berr    = particle->berr;
34 745     float phi     = particle->phi;
35 746
36 747     for(int t = 0; t < num_steps; t++) {
37 748
38 749         for (int loc = 0; loc < nloc; loc++) {
39 750             S_last[loc] = S[loc];
40 751             I_last[loc] = I[loc];
41 752             R_last[loc] = R[loc];
42 753             B_last[loc] = B[loc];
43 754         }
44 755
45 756         for (int loc = 0; loc < nloc; loc++) {
46 757
47 758             B[loc] = exp( log(B_last[loc]) + eta*(log(B0) - log(
48                  B_last[loc])) + berr*curand_normal(&(amp;particle->
49                  randState)) );
50 759
51 760             int n = neinum[loc];

```

```

1 761         float sph = 1.0 - phi*( (float) n/(n+1.0) );
2 762         float ophi = phi/(n+1.0);
3 763
4 764         float nBisum = 0.0;
5 765         for (int j = 0; j < n; j++)
6 766             nBisum += B_last[neibmat[nloc*loc + j]-1] * I_last[
7 767                 neibmat[nloc*loc + j]-1];
8 768
9 769         float BSI = S_last[loc]*( sph*B_last[loc]*I_last[loc] +
10 770             ophi*nBisum );
11 771         float rI = r*I_last[loc];
12 772
13 773         // get derivatives
14 774         float dS = - BSI;
15 775         float dI = BSI - rI;
16 776         float dR = rI;
17 777
18 778         // step forward by h
19 779         S[loc] += h*dS;
20 780         I[loc] += h*dI;
21 781         R[loc] += h*dR;
22 782
23 783     }
24 784
25 785 }
26 786
27 787 free(S_last);
28 788 free(I_last);
29 789 free(R_last);
30 790 free(B_last);
31 791
32 792 }
33 793
34 794 /* Convenience function for particle resampling process
35 795 */
36 796 __device__ void copyParticle(Particle * dst, Particle * src, int nloc
37 797 ) {
38 798
39 799     dst->R0 = src->R0;
40 800     dst->r = src->r;
41 801     dst->sigma = src->sigma;
42 802     dst->eta = src->eta;
43 803     dst->berr = src->berr;
44 804     dst->phi = src->phi;
45 805
46 806     for (int n = 0; n < nloc; n++) {
47 807         dst->S[n] = src->S[n];
48 808         dst->I[n] = src->I[n];
49 809         dst->R[n] = src->R[n];
50 810         dst->B[n] = src->B[n];
51 811         dst->Iinit[n] = src->Iinit[n];

```

```

1 809     }
2 810
3 811 }
4 812
5 813 /* Convert memory size in bytes to human-readable format
6 814 */
7 815 std::string getHRmemsize (size_t memsize) {
8 816
9 817     std::stringstream ss;
10 818     std::string valstring;
11 819
12 820     int kb = 1024;
13 821     int mb = kb*1024;
14 822     int gb = mb*1024;
15 823
16 824     if (memsize <= kb)
17 825         ss << memsize << " B";
18 826     else if (memsize > kb && memsize <= mb)
19 827         ss << (float) memsize/ kb << " KB";
20 828     else if (memsize > mb && memsize <= gb)
21 829         ss << (float) memsize/ mb << " MB";
22 830     else
23 831         ss << (float) memsize/ gb << " GB";
24 832
25 833     valstring = ss.str();
26 834
27 835     return valstring;
28 836
29 837 }
30 838
31 839
32 840 /* Convert time in seconds to human readable format
33 841 */
34 842 std::string getHRtime (float runtime) {
35 843
36 844     std::stringstream ss;
37 845     std::string valstring;
38 846
39 847     int mt = 60;
40 848     int ht = mt*60;
41 849     int dt = ht*24;
42 850
43 851     if (runtime <= mt)
44 852         ss << runtime << " s";
45 853     else if (runtime > mt && runtime <= ht)
46 854         ss << runtime/mt << " m";
47 855     else if (runtime > ht && runtime <= dt)
48 856         ss << runtime/dt << " h";
49 857     else
50 858         ss << runtime/ht << " d";
51 859

```

```
1 860     valstring = ss.str();  
2 861  
3 862     return valstring;  
4 863  
5 864 }
```

7 The parameter estimation means as compared to IF2 and HMCMC are shown in
8 Figure [].

9 The running times for parameter fitting as compared to IF2 and HMCMC are shown
10 in Figure[].