

FUN WITH FORECASTING USING STOCHASTIC NON-LINEAR DYNAMICS

Dexter Barrows

Supervisor: Dr. Benjamin Bolker

A thesis presented for the degree of
Master of Science

Department of Mathematics and Statistics
McMaster University

Canada

March 22, 2016

1 Abstract

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

Dedication

1

To Mom and Dad

2

¹ Acknowledgements

² Sooooooooo many people

Contents

1

1	Introduction	1	2
2	Hamiltonian MCMC	4	3
2.1	Intro	4	4
2.2	Markov Chains	4	5
2.3	Likelihood	6	6
2.4	Prior distribution	6	7
2.5	Proposal distribution	6	8
2.6	Algorithm	7	9
2.7	Burn-in	8	10
2.8	Thinning	8	11
2.9	Hamiltonian Monte Carlo	9	12
2.10	Fitting	12	13
3	Iterated Filtering	20	14
3.1	Intro	20	15
3.2	Formulation	20	16
3.3	Algorithm	21	17
3.4	Particle Collapse	23	18
3.5	Iterated Filtering and Data Cloning	23	19
3.6	IF2	24	20
3.7	Fitting	26	21
4	Parameter Fitting	30	22
4.1	Fitting Setup	30	23
4.2	Calibrating Samples	33	24
4.3	IF2 Fitting	34	25
4.4	IF2 Convergence	36	26
4.5	IF2 Densities	38	27
4.6	HMCMC Fitting	39	28
4.7	HMCMC Densities	39	29
4.8	HMCMC and Bootstrapping	40	30

1	4.9	Multi-trajectory Parameter Estimation	41
2	5	Forecasting Frameworks	43
3	5.1	Data Setup	43
4	5.2	IF2	44
5	5.2.1	Parametric Bootstrapping	45
6	5.2.2	IF2 Forecasts	45
7	5.3	HMCMC	47
8	5.4	Truncation vs. Error	48
9	6	S-map and SIRS	50
10	6.1	S-maps	50
11	6.2	S-map Algorithm	51
12	6.3	SIRS Model	53
13	6.4	SIRS Model Forecasting	55
14	7	Spatial Epidemics	57
15	7.1	Spatial SIR	57
16	7.2	Dewdrop Regression	59
17	7.3	Spatial Model Forecasting	60
18	8	Discussion and Future Directions	62
19	8.1	Parallel and Distributed Computing	62
20	8.2	IF2, Bootstrapping, and Forecasting Methodology	65
21	A	Hamiltonian MCMC	66
22	A.1	Full R code	66
23	A.2	Full Stan code	69
24	B	Iterated Filtering	71
25	B.1	Full R code	71
26	B.2	Full C++ code	74
27	C	Parameter Fitting	85
28	D	Forecasting Frameworks	86
29	D.1	IF2 Parametric Bootstrapping Function	86
30	D.2	RStan Forward Simulator	88
31	E	S-map and SIRS	90
32	E.1	SIRS R Function Code	90
33	E.2	SMAP Code	91
34	E.3	SMAP Parameter Optimization Code	93

E.4	RStan SIRS Code	95	1
E.5	IF2 SIRS Code	97	2
F	Spatial Epidemics	110	3
F.1	Spatial SIR R Function Code	110	4
F.2	RStan Spatial SIR Code	112	5
F.3	IF2 Spatial SIR Code	114	6
F.4	CUDA IF2 Spatial Fitting Code	125	7

1 List of Figures

2	2.1	Finite state machine. (<i>Andrieu et al., 2003</i>)	5
3	2.2	True SIR ODE solution infected counts, and with added observation	
4		noise	14
5	2.3	Traceplot of samples drawn for parameter R_0 , excluding warmup . .	17
6	2.4	Traceplot of samples drawn for parameter R_0 , including warmup. . .	18
7	2.5	Kernel density estimates produced by Stan	19
8	3.1	True SIR ODE solution infected counts, and with added observation	
9		noise.	28
10	3.2	Kernel estimates for four essential system parameters. True values	
11		are indicated by solid vertical lines, sample means by dashed lines. .	29
12	4.1	Simulated geometric autoregressive process show in Equation [4.2]. .	31
13	4.2	Density plot of values shown if Figure[4.1].	32
14	4.3	Stochastic SIR model simulated using an explicit Euler stepping scheme.	
15		The solid line is a single random trajectory, the dots show the data	
16		points obtained by adding in observation error defined as $\epsilon_{obs} =$	
17		$\mathcal{N}(0, 10)$, and the grey ribbon is centre 95th quantile from 100 ran-	
18		dom trajectories.	32
19	4.5	True system trajectory (solid line), observed data (dots), and IF2	
20		estimated real state (dashed line).	35
21	4.4	Fitting errors.	35
22	4.6	The horizontal axis shows the IF2 pass number. The solid black lines	
23		show the evolution of the ML estimates, the solid grey lines show	
24		the true value, and the dashed grey lines show the mean parameter	
25		estimates from the particle swarm after the final pass.	37
26	4.7	The horizontal axis shows the IF2 pass number and the solid black	
27		lines show the evolution of the standard deviations of the particle	
28		swarm values.	37
29	4.8	As before, the solid grey lines show the true parameter values and	
30		the dashed grey lines show the density means.	39

4.9	As before, the solid grey lines show the true parameter values and the dashed grey lines show the density means.	40	1	2
4.10	Result from 100 HMCMC bootstrap trajectories. The solid line shows the true states, the dots show the data, the dotted line shows the average system behaviour, the dashed line shows the bootstrap mean, and the grey ribbon shows the centre 95th quantile of the bootstrap trajectories.	41	3	4
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.	42	5	6
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.	42	7	8
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 $\mathcal{N}(0, 10)$	44	9	10
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.	45	11	12
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 centre 95th quantile of the true state estimates, and the lighter grey ribbon shows the centre 95th quantile of the true state estimates with added observation noise drawn from $\mathcal{N}(0, \sigma)$	46	13	14
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 centre 95th quantile.	48	15	16
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.	49	17	18
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.	54	19	20
6.2	S-map applied to the data from the previous figure. The solid line shows the infection counts with observation noise form the previous plot, and the dotted line is the S-map forecast. Parameters chosen were $E = 14$ and $\theta = 3$	54	21	22

1	6.3	Error as a function of forecast length.	56
2	6.4	Runtimes for producing SIRS forecasts. The box shows the middle	
3		50th quantile, the bold line is the median, and the dots are outliers.	56
4	7.1	Evolution of a spatial epidemic in a ring topology. The outbreak	
5		was started with 5 cases in Location 2. Parameters were $R_0 = 3.0$,	
6		$\gamma = 0.1$, $\eta = 0.5$, $\sigma_{err} = 0.5$, and $\phi = 0.5$	58
7	7.2	Evolution of a spatial epidemic as in Figure [7.1], with added obser-	
8		vation noise drawn from $\mathcal{N}(0, 10)$	59
9	7.3	Average SSE (log scale) across each location and all trials as a func-	
10		tion of the number of weeks ahead in the forecast.	61
11	7.4	Runtimes for producing spatial SIR forecasts. The box shows the	
12		middle 50th quantile, the bold line is the median, and the dots are	
13		outliers.	61

Chapter 1

1

Introduction

2

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 both from a theoretical and practical standpoint. Mathematical models can capture the essential drivers in disease dynamics, and extended 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, so uncertainty in what we observe in building mathematical models of disease spread must be accounted for from the get-go. Further, 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.

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 distri-
 2 butions. It uses three parameters representing the degree of autoregression (p),
 3 integration (trend removal) (d), and the moving average (q), where the orders of
 4 the autoregression and the moving average are determined through the use of an
 5 autocorrelation function (ACF) and partial autocorrelation function (PACF), re-
 6 spectively, applied to the the data *a priori*.

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

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

15 where S , I , and R are the number of individuals in each compartment, β is the
 16 “force” of infection acting on the susceptible population, and γ is a recovery rate.
 17 As an outbreak progresses, individuals transition from the susceptible compartment,
 18 through the infectious compartment, then finish in the removed compartment where
 19 they no longer impact the system dynamics. Many extensions of the SIR model exist
 20 are are commonly used, such as the SEIR model in which susceptible individuals pass
 21 through an exposed class where they have been infected but are not yet themselves
 22 infectious, and the SIRS model in which individuals become susceptible again after
 23 their immunity wanes.
 24

25 Combining the phenomenological and mechanistic approaches are the semi-mechanistic
 26 techniques. These methods use a model to define the expected underlying dynamics
 27 of the system, but integrate data into the model in order to refine estimates of the
 28 model parameters and produce more accurate forecasts. Typically the first step in
 29 implementing such a technique is fitting the desired model to existing data. There
 30 are many ways to do this, most of which fall into two main categories: particle
 31 filter-based (PF) methods, and Markov chain Monte Carlo-based (MCMC) meth-
 32 ods. From there data can either be integrated into the model by refitting the model
 33 to the new longer data set, or in an “on-line” fashion in which data points can be di-
 34 rectly integrated without the need to refit the entire model. Normally, MCMC-based

machinery must refit the entire model whereas PF-based approaches can sometimes
integrate data in an on-line fashion.

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

Somewhat frustratingly, there exists no “gold standard” in forecasting. As method-
ology varies widely in theoretical justification, implementation, and operation, it is
difficult to compare the state of the art in forecasting methods from a first-principles
perspective. Further, published work using any of these methods to forecast uses
different prediction accuracy metrics, such as SSE, peak time/duration/intensity,
correlation tests, or RMSE, among others. Thus is is difficult to select the best tool
for the job when faced with a forecasting problem.

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

For techniques we have the following: from MCMC-based methods we have selected
Hamiltonian MCMC [*ref*], a less recent but nonetheless highly effective technique,
from PF-based methods we have selected IF2 [*Ionides ref*], a newer approach that
uses multiple particle filtering rounds to generate MLEs, and from the phenom-
ological methods we have selected the sequential locally weighted global linear maps
(S-map) [*Sugihara ref*].

Chapter 2

Hamiltonian MCMC

2.1 Intro

Markov Chain Monte Carlo (MCMC) is part of 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 or equilibrium distribution is used to approximate the desired posterior distribution.

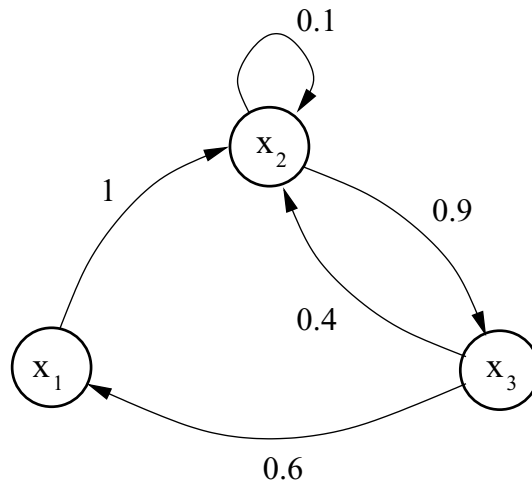
2.2 Markov Chains

Consider a finite state machine with 3 states $S = \{x_1, x_2, x_3\}$, where the probability of transitioning from one particular state to another is shown as a transition graph in Figure [2.1].

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

Figure 2.1: Finite state machine. (*Andrieu et al., 2003*)

$\mu(x^{(1)})T$, 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

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

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

Note that this property holds when the chain satisfies the following conditions

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

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

2.3 Likelihood

MCMC and similar methods hinge on the idea that the weight or support bestowed upon a particular set of parameters θ should be proportional to the probability of observing the data D given the model output using that set of parameters $M(\theta)$. In order to do this we need a way to evaluate whether or not $M(\theta)$ is a good fit for D ; this is done by specifying a likelihood function $\mathcal{L}(\theta)$ such that

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

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

2.4 Prior distribution

Another significant component of MCMC is the user-specified prior distribution for θ or distributions for the individual components of θ (Priors). Priors serve as a way for us to tell the MCMC algorithm what we think consist of good values for the parameters.

Note that if very little is known about the parameters, or we are worried about biasing our estimate of the posterior, we can simply use a wide uniform distribution. However, this handicaps the algorithm in two ways: convergence of the chain may become exceedingly slow, and more pressure is put on the likelihood function to be as good as possible – it will now be the only thing informing the algorithm of what constitutes a “good” set of parameters, and what should be considered poor.

2.5 Proposal distribution

As part of the MCMC algorithm, when we find a state in the parameter space that is accepted as part of the Markov chain construction process, we need a good way of generating a good next step to try. Unlike basic rejection sampling in which we would just randomly sample from our prior distribution, MCMC attempts to optimise our choices by choosing a step that is close enough to the last accepted step so as to stand a decent chance of also being accepted, but far enough away that it doesn’t get “trapped” in a particular region of the parameter space.

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

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

2.6 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.

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

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

Thus, the MCMC algorithm shown in Algorithm [1].

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-Hastings 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

2.7 Burn-in

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

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

10

2.8 Thinning

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

space), but require less room to store.

2.9 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 decrease in overall computational complexity.

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 stan-
 11 dard Metropolis-Hastings MCMC. Now, however, the acceptance ratio r is defined
 12 as

$$13 \quad r = \exp [H(\theta, r) - H(\theta^*, r^*)], \quad (2.10)$$

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

15 Together, we have Algorithm [2].

16 Note that the parameters ε and L have to be tuned in order to maintain stability
 17 and maximize efficiency, a sometimes non-trivial 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  $(\theta^{(1)}, \theta^{(2)}, \dots, \theta^{(N)})$ 

```

2.10 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

$$\begin{aligned}\frac{dS}{dt} &= -\beta IS \\ \frac{dI}{dt} &= \beta IS - rI \\ \frac{dR}{dt} &= rI\end{aligned}\tag{2.11}$$

where S is the number of individuals susceptible to infection, I is the number of infectious individuals, R is the number of recovered individuals, $\beta = R_0 r / N$ is the force of infection, R_0 is the number of secondary cases per infected individual, r is the recovery rate, and N is the population size.

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

```

1  SIR <- function(Time, State, Pars) {
2
3      with(as.list(c(State, Pars)), {
4
5          B <- R0*r/N      # calculate Beta
6          BSI <- B*S*I     # save product
7          rI <- r*I        # save product
8
9          dS = -BSI        # change in Susceptible people
10         dI = BSI - rI     # change in Infected people
11         dR = rI          # change in Removed (recovered people)
12
13         return(list(c(dS, dI, dR)))
14     })
15 }
16
17 
```

The true parameter values were set to $R_0 = 3.0$, $r = 0.1$, $N = 500$ by

```

1 pars <- c(R0 <- 3.0, # new infected people per infected person
2           r <- 0.1, # recovery rate
3           N <- 500) # population size

```

The system was integrated over $[0, 100]$ with infected counts drawn at each integer time step. These timings were set using

```

1 T <- 100 # total integration time
2 times <- seq(0, T, by = 1) # times to draw solution
   values

```

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. These were set using

```

1 y_ini <- c(S = 495, I = 5, R = 0) # initial conditions

```

The `ode()` function is called as

```

1 odeout <- ode(y_ini, times, SIR, pars)

```

where `odeout` is a $(T + 1) \times 4$ matrix where the rows correspond to solutions at the given times (the first row is the initial condition), and the columns correspond to the solution times and S-I-R counts at those times.

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

These “true” values were perturbed to mimic observation error by

```

1 set.seed(1001) # set RNG seed for reproducibility
2 sigma <- 5 # observation error standard deviation
3 infec_counts_raw <- odeout[,3] + rnorm(101, 0, sigma)
4 infec_counts <- ifelse(infec_counts_raw < 0, 0, infec_counts)

```

where the last two lines simply set negative observations (impossible) to 0.

Plotting the data using the `ggplot2` package by

```

1 g <- qplot(0:T, odeout[,3], geom = "line", xlab = "Time (weeks)",
2           ylab = "Infection Count") +
3   geom_point(aes(y = infec_counts)) +
4   theme_bw()
5 print(g)

```

we obtain Figure [2.2].

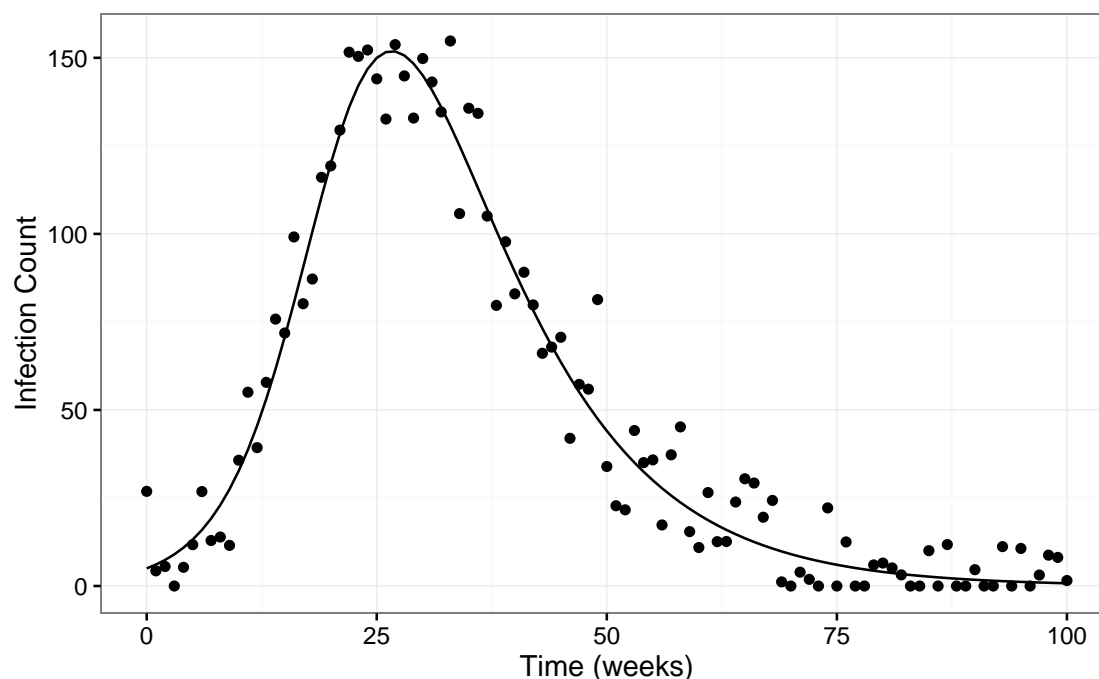


Figure 2.2: True SIR ODE solution infected counts, and with added observation noise

1 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>)
 2
 3 was used to ease implementation.
 4

5 In order to use an Explicit Euler-like stepping method in the later Stan model (both
 6 for speed and for integration method homogeneity with other methods against which
 7 HMC/MC was compared), the synthetic observation counts were treated as weekly
 8 observations in which the counts on the other six days of the week were unobserved.
 9 For computational and organizational simplicity, these values were set to -1 (all valid
 10 observations are non-negative). This is done in R using

```

11 1 sPw <- 7 # steps per week
12 2 datlen <- (T-1)*7 + 1 # size of sparse data vector
13 3
14 4 data <- matrix(data = -1, nrow = T+1, ncol = sPw)
15 5 data[,1] <- infec_counts
16 6 standata <- as.vector(t(data))[1:datlen]
17

```

19 The data to be fed into the R Stan interface is packed as

```

20 1 sir_data <- list( T = datlen, # simulation time
21

```



```

2          y = standata, # infection count data      1
3          N = 500,      # population size           2
4          h = 1/sPw )   # step size per day         3

```

For efficiency we allow Stan to save compiled code to avoid recompilation, and allow multiple chains to be run simultaneously on separate CPU cores

```

1  rstan_options(auto_write = TRUE)      7
2  options(mc.cores = parallel::detectCores()) 8

```

Now we call the Stan fitting function

```

1  stan_options <- list( chains = 4, # number of chains      11
2                        iter  = 2000, # iterations per chain 12
3                        warmup = 1000, # warmup iterations    13
4                        thin   = 2 ) # thinning number        14
5  fit <- stan( file    = "d_siode_euler.stan",               15
6              data     = sir_data,                          16
7              chains    = stan_options$chains,              17
8              iter      = stan_options$iter,                18
9              warmup    = stan_options$warmup,              19
10             thin      = stan_options$thin )                20

```

which fits the model in the file `d_siode_euler.stan` to the data passed in through `sir_data`. The options here specify that 10 chains will be run, each with a burn in period of 1000 steps, with 5000 steps to sample over, and only sampling every 10th step. Options are saved so they can be accessed later.

The Stan file contains three blocks that together specify the model. First, the data block specifies the information the model expects to be given. Here, this is

```

1  data {
2
3      int      <lower=1>    T;      // total integration steps
4      real     y[T];        // observed number of cases
5      int      <lower=1>    N;      // population size
6      real     h;           // step size
7
8  }

```

where each of the data variables correspond to data passed in through the previously shown R code.

Next the parameters block specifies what Stan is expected to estimate. Here this is

```

1  parameters {
2
3      real <lower=0, upper=10> sigma; // observation error

```

```

1  4      real <lower=0, upper=10>    R0;      // R0
2  5      real <lower=0, upper=10>    r;        // recovery rate
3  6      real <lower=0, upper=500>    y0[3];    // initial conditions
4  7
5  8      }

```

7 Finally we have the model block. This crucial part of the code specifies the inter-
 8 action between the parameters and the data. The core component of the model
 9 indicates we are fitting an approximation of an ODE model using Euler integration
 10 steps (one per day), with the initial conditions and SIR parameters unknown. Fur-
 11 ther, we can also specify the prior distributions to draw new parameter values from.
 12 The initial conditions are taken to be close to the initial data point, with adjustment
 13 for observation error, while the other parameters are assumed to be coming from
 14 log-normal distributions with relatively small means. Together, we have

```

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

```

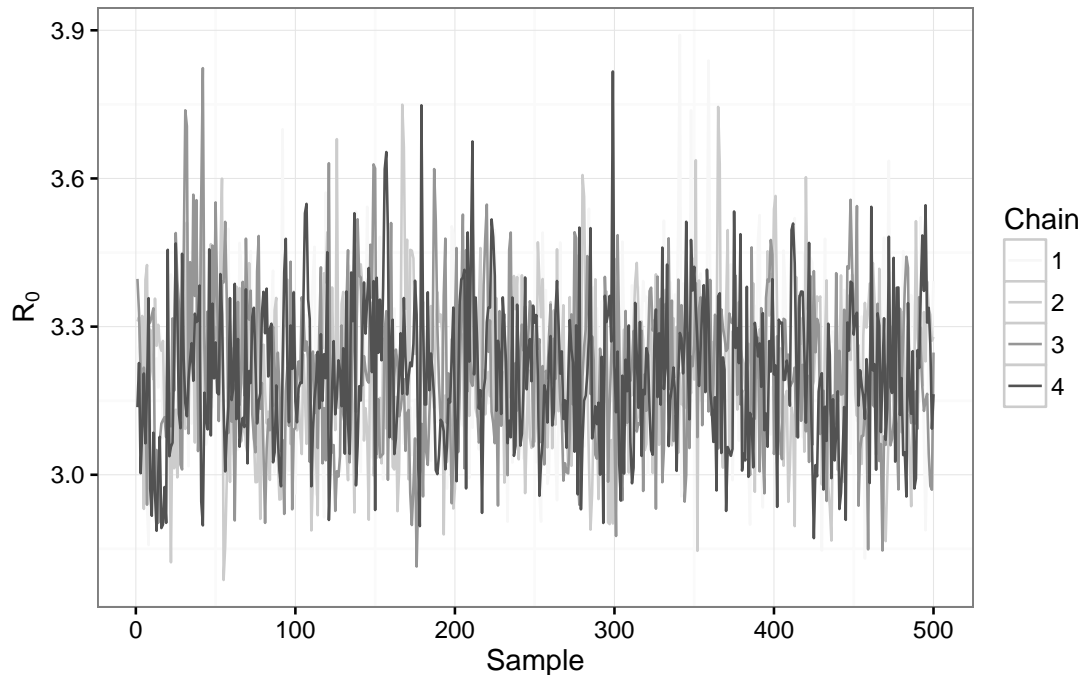


Figure 2.3: Traceplot of samples drawn for parameter R_0 , excluding warmup

Examining 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. This is shown in Figure [2.3].

Further, if we look at the chain data including the warmup samples in Figure [2.4], we can see why is is wise to discard these samples (note the scale).

Now if we look at the kernel density estimates for each of the model parameters and the initial number of cases, we see that while the estimates are not perfect, they are fairly decent. This is shown in Figure [2.5].

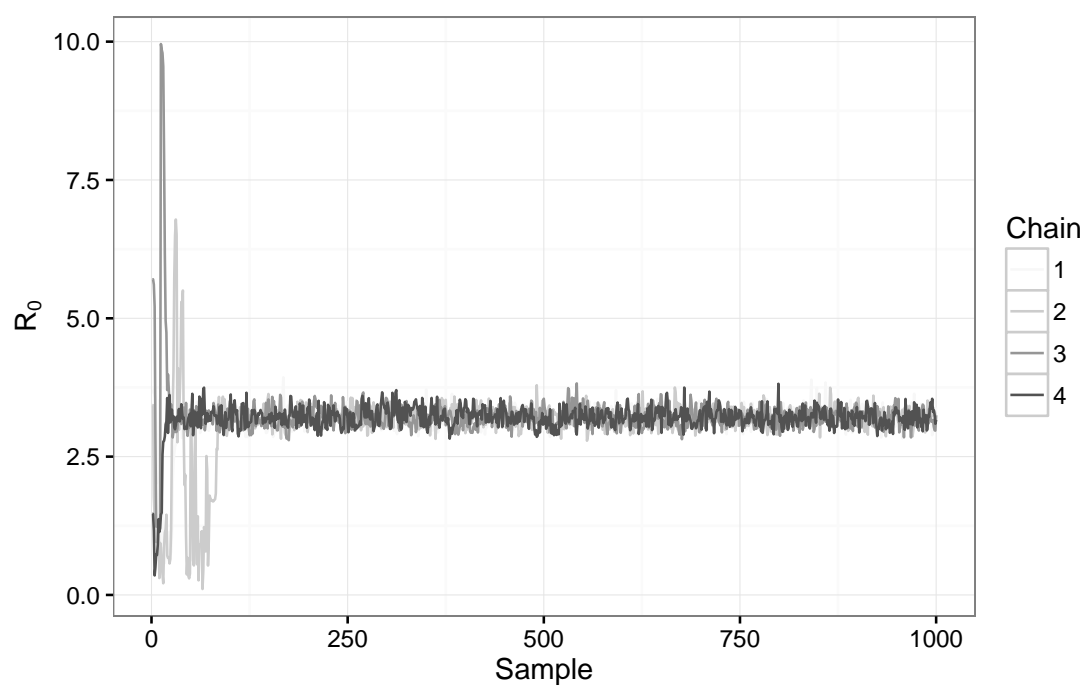


Figure 2.4: Traceplot of samples drawn for parameter R_0 , including warmup.

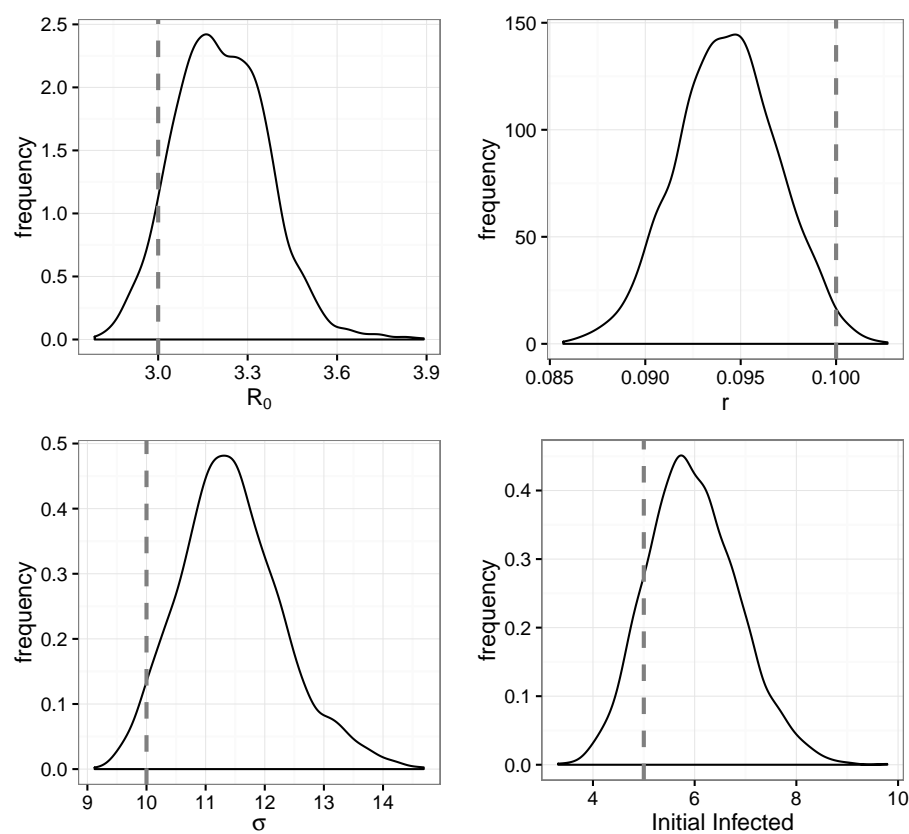


Figure 2.5: Kernel density estimates produced by Stan

Chapter 3

Iterated Filtering

3.1 Intro

Particle filters are similar to MCMC-based methods in that they attempt to draw samples from an approximation of the posterior distribution of model parameters θ given observed data D . 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 not dissimilar from Darwinian selection, then this will be a Sequential Importance Resampling (SIR) particle filter.

3.2 Formulation

Particle filters, also called Sequential Monte-Carlo (SMC) or bootstrap 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

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

And the observation process by 2

$$Y_t \sim f_2(X_t, \theta). \quad (3.2) \quad 3$$

Note that components of θ can contribute to both functions, but a typical formula- 4
tion is to have some components contribute to $f_1(\cdot, \theta)$ and others to $f_2(\cdot, \theta)$. 5

The prediction part of the cycle utilises $f_1(\cdot, \theta)$ 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

3.3 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 Algo- 19
rithm [3]. 20

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

```

3.4 Particle Collapse

Not uncommonly, 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

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

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

3.5 Iterated Filtering and Data Cloning

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

1 Rigorous proofs have been developed (references to Ionides et. al. work) that show
2 that by treating the parameters as stochastic processes instead of fixed values, the
3 multiple passes through the data will indeed force convergence of the process mean
4 toward maximum likelihood, and the process variance toward 0.

5 **3.6 IF2**

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

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

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

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

28

Algorithm 4: IF2

```

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

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

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

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

```

3.7 Fitting

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

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

$$\begin{aligned}\frac{dS}{dt} &= -\beta IS \\ \frac{dI}{dt} &= \beta IS - rI \\ \frac{dR}{dt} &= rI\end{aligned}\tag{3.4}$$

where S is the number of individuals susceptible to infection, I is the number of infectious individuals, R is the number of recovered individuals, $\beta = R_0 r / N$ is the force of infection, R_0 is the number of secondary cases per infected individual, r is the recovery rate, and N is the population size.

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

```

1  SIR <- function(Time, State, Pars) {
2
3      with(as.list(c(State, Pars)), {
4
5          B    <- R0*r/N    # calculate Beta
6          BSI  <- B*S*I    # save product
7          rI   <- r*I      # save product
8
9          dS   = -BSI      # change in Susceptible people
10         dI   = BSI - rI   # change in Infected people
11         dR   = rI        # change in Removed (recovered people)
12
13         return(list(c(dS, dI, dR)))
14
15     })
16
17 }
```

The true parameter values were set to $R_0 = 3.0$, $r = 0.1$, $N = 500$ by

```

1  pars <- c(R0 = 3.0, # new infected people per infected person
```

```

2      r    = 0.1, # recovery rate
3      N    = 500) # population size

```

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. These were set using

```

1 true_init_cond <- c(S = N - i_infec,
2                    I = i_infec,
3                    R = 0)

```

The `ode()` function is called as

```

1 odeout <- ode(y = true_init_cond, times = 0:(T-1), func = SIR,
2              parms = true_pars)

```

where `odeout` is a $T \times 4$ matrix where the rows correspond to solutions at the given times (the first row is the initial condition), and the columns correspond to the solution times and S-I-R counts at those times.

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

These “true” values were perturbed to mimic observation error by

```

1 set.seed(1001) # set RNG seed for reproducibility
2 sigma <- 10    # observation error standard deviation
3 infec_counts_raw <- odeout[,3] + rnorm(101, 0, sigma)
4 infec_counts <- ifelse(infec_counts_raw < 0, 0, infec_counts)

```

where the last two lines simply set negative observations (impossible) to 0.

Plotting the data using the `ggplot2` package by

```

1 plotdata <- data.frame(times=1:T, true=trueTraj, data=infec_counts)
2
3 g <- ggplot(plotdata, aes(times)) +
4   geom_line(aes(y = true, colour = "True")) +
5   geom_point(aes(y = data, color = "Data")) +
6   labs(x = "Time", y = "Infection count", color = "") +
7   scale_color_brewer(palette="Paired") +
8   theme(panel.background = element_rect(fill = "#F0F0F0"))

```

we obtain Figure [3.1].

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

```

1 sourceCpp(paste(getwd(), "if2.cpp", sep="/"))

```

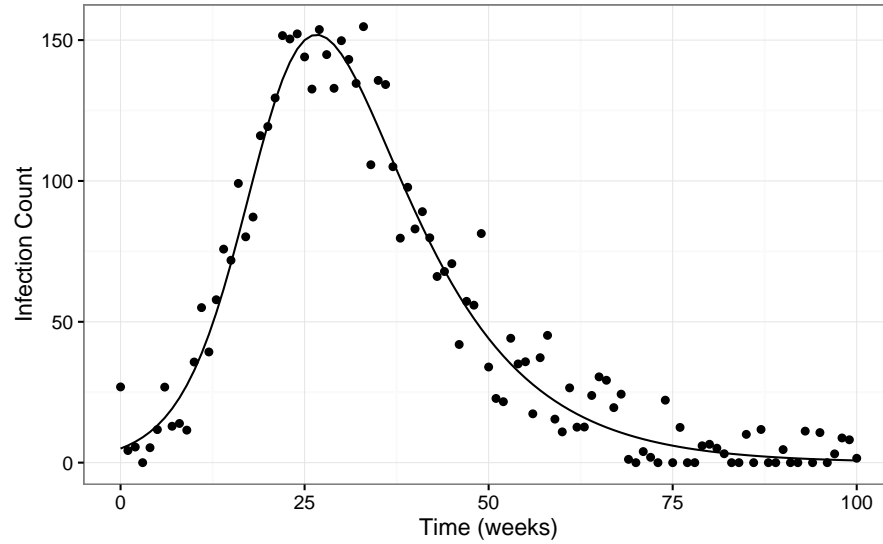


Figure 3.1: True SIR ODE solution infected counts, and with added observation noise.

1 Then run and packed into a data frame using

```
2 1 paramdata <- data.frame(if2(infec_counts[1:Tlim], Tlim, N))
3 2 colnames(paramdata) <- c("R0", "r", "sigma", "Sinit", "Iinit", "
4 Rinit")
5
```

7 The final kernel estimates for four of the key parameters are shown in Figure
8 [3.2].

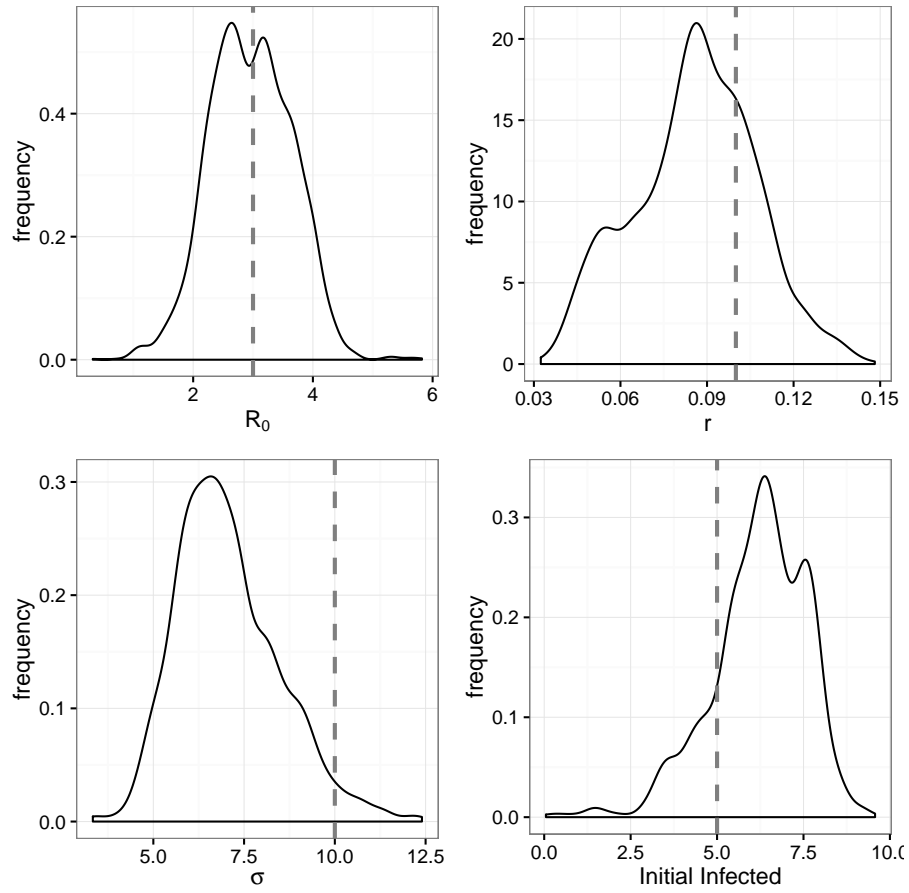


Figure 3.2: Kernel estimates for four essential system parameters. True values are indicated by solid vertical lines, sample means by dashed lines.

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 given by

$$\begin{aligned}\frac{dS}{dt} &= -\beta SI \\ \frac{dI}{dt} &= \beta SI - \gamma I \\ \frac{dR}{dt} &= \gamma I,\end{aligned}\tag{4.1}$$

and add process noise by allowing β to embark on a geometric random walk given by

$$\beta_{t+1} = \exp \left(\log(\beta_t) + \eta(\log(\bar{\beta}) - \log(\beta_t)) + \epsilon_t \right).\tag{4.2}$$

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).

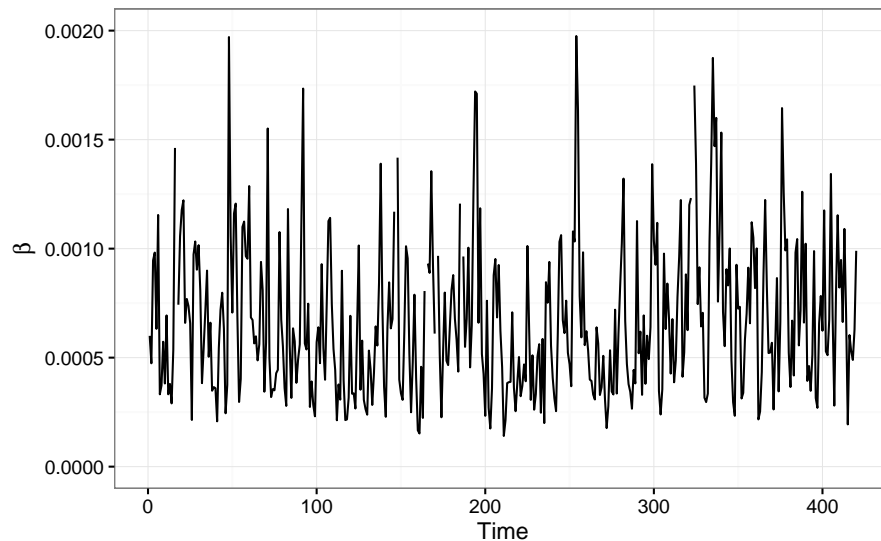


Figure 4.1: Simulated geometric autoregressive process show in Equation [4.2].

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

$$X_{t+1} = c + \rho X_t + \epsilon_t, \quad (4.3)$$

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 $\sigma = \sigma_E^2/(1 - \rho^2)$. If we choose $\eta = 0.5$, the resulting log-normal distribution has a mean of 6.80×10^{-4} and standard deviation of 4.46×10^{-4} .

Simulating the process in Equation [4.2] with $\eta = 0.5$ gives us the plot in Figure [4.1].

We can obtain the corresponding density plot of the values in Figure [4.1], shown in Figure [4.2].

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

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

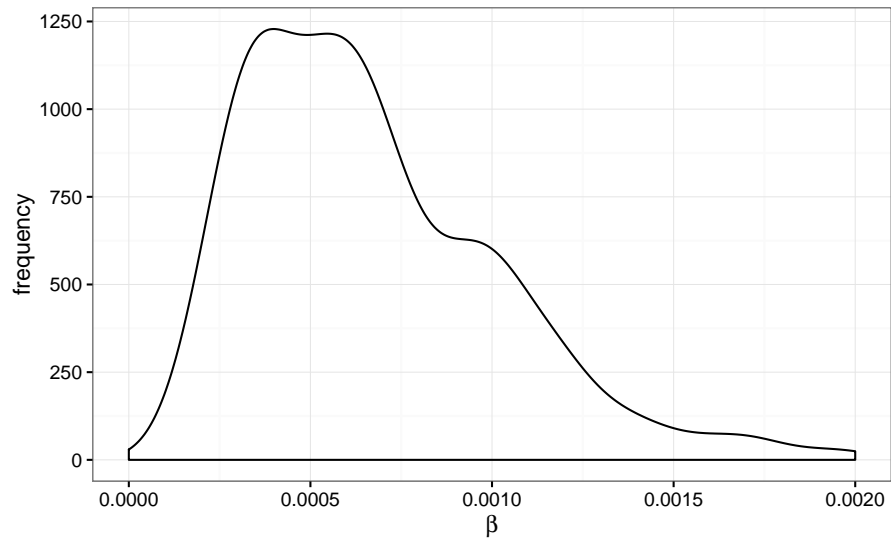


Figure 4.2: Density plot of values shown in Figure 4.1.

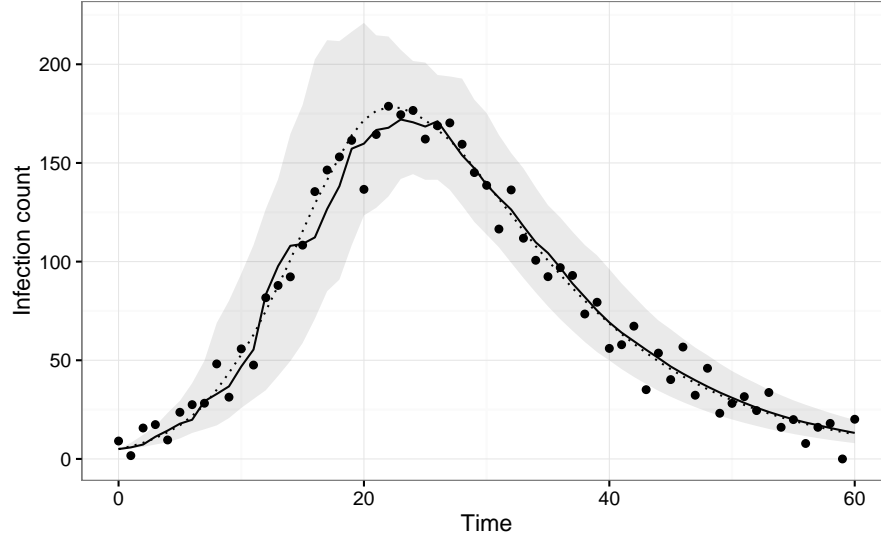


Figure 4.3: Stochastic SIR model simulated using an explicit Euler stepping scheme. The solid line is a single random trajectory, the dots show the data points obtained by adding in observation error defined as $\epsilon_{obs} = \mathcal{N}(0, 10)$, and the grey ribbon is centre 95th quantile from 100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

4.2 Calibrating Samples

In order to compare HMCMC and IF2 we need to set up a fair and theoretically justified way to select the number of samples to draw for the HMCMC iterations and the number of particles to use for IF2. We assume that we are working with a problem that has an unknown real solution, so we use the Monte Carlo Standard Error (MCSE).

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

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

$$\hat{\sigma}_n = \sqrt{\frac{1}{n} \sum_{i=1}^n (X_i - \hat{\mu})^2}, \quad (4.4)$$

which is known as the Monte Carlo Standard Error.

We can modify this formula to account for multiple variables by replacing the single variance measure sum by

$$\Theta^* V (\Theta^*)^T \quad (4.5)$$

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

$$\hat{\sigma}_n = \sqrt{\frac{1}{n} \frac{1}{P} \Theta^* V (\Theta^*)^T} \quad (4.6)$$

where P is the number of particles.

The goal here is to then pick the number of HMCMC samples and IF2 particles to yield similar MCSE values. To do this we picked a combination of parameters for RStan that yielded decent results when applied to the stochastic SIR model specified above, calculated the resulting mean MCSE across several model fits, and isolated

1 the expected number of IF2 particles needed to obtain the same value. This was
2 used as a starting value to “titrate” the IF2 iterations to the same point.

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

6 4.3 IF2 Fitting

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

13 The code used here is a mix of R and C++ implemented using RCpp. The fitting
14 was undertaken using 2500 particles with 50 IF2 passes and a cooling schedule given
15 by a reduction in particle spread determined by 0.975^p , where p is the pass number
16 starting with 0.

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

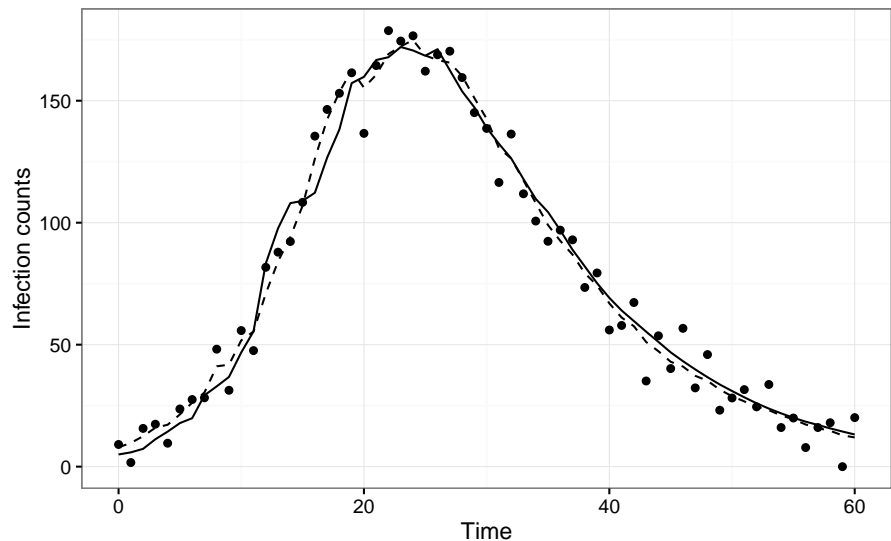


Figure 4.5: True system trajectory (solid line), observed data (dots), and IF2 estimated real state (dashed line).

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.

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

1
2
3

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
4 estimates as a function of the pass number. Plots showing evolution of the mean
5 estimates are shown in Figure [4.6] for the six most critical parameters.

6 Similarly, we can look at the evolution of the standard deviations of the parameter
7 estimates from the particle swarm as a function of the pass number, shown in Figure
8 [4.7].

9 As expected there is a downward trend in all plots, with a very strong trend in all
10 but two of them.

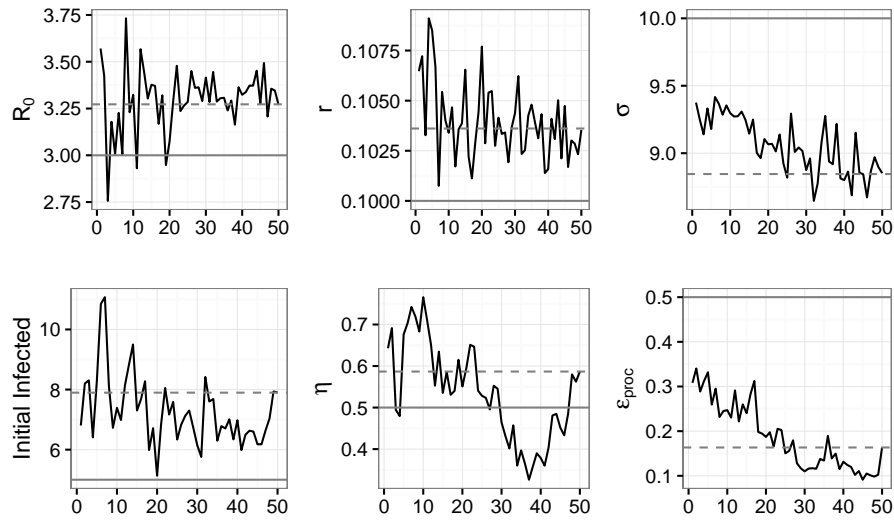


Figure 4.6: The horizontal axis shows the IF2 pass number. The solid black lines show the evolution of the ML estimates, the solid grey lines show the true value, and the dashed grey lines show the mean parameter estimates from the particle swarm after the final pass.

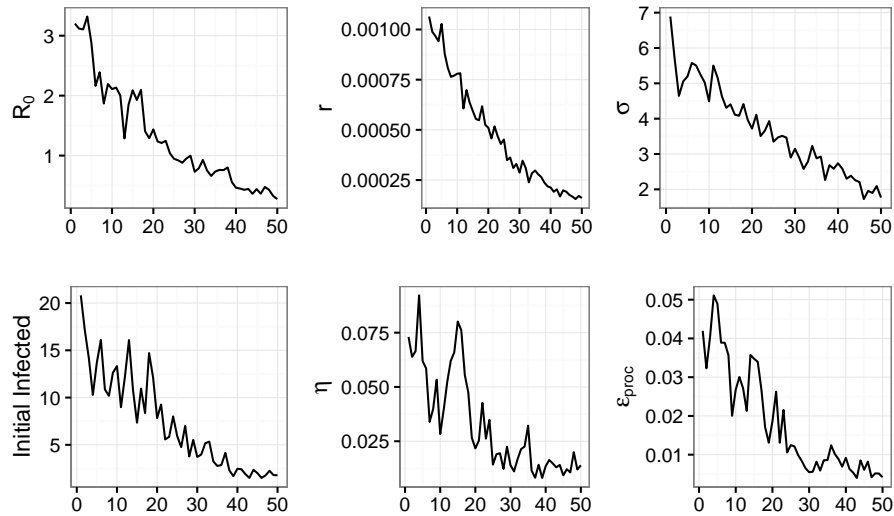


Figure 4.7: The horizontal axis shows the IF2 pass number and the solid black lines show the evolution of the standard deviations of the particle swarm values.

1 4.5 IF2 Densities

2 Of diagnostic importance are the densities of the parameter estimates given by the
3 final parameter swarm. These are shown in Figure [4.8].

4 It is worth noting that the IF2 parameters chosen were in part chosen so as to
5 not artificially narrow these densities; a more aggressive cooling schedule and/or
6 an increased number of passes would have resulted in much narrower densities, and
7 indeed have the potential to collapse them to point estimates.

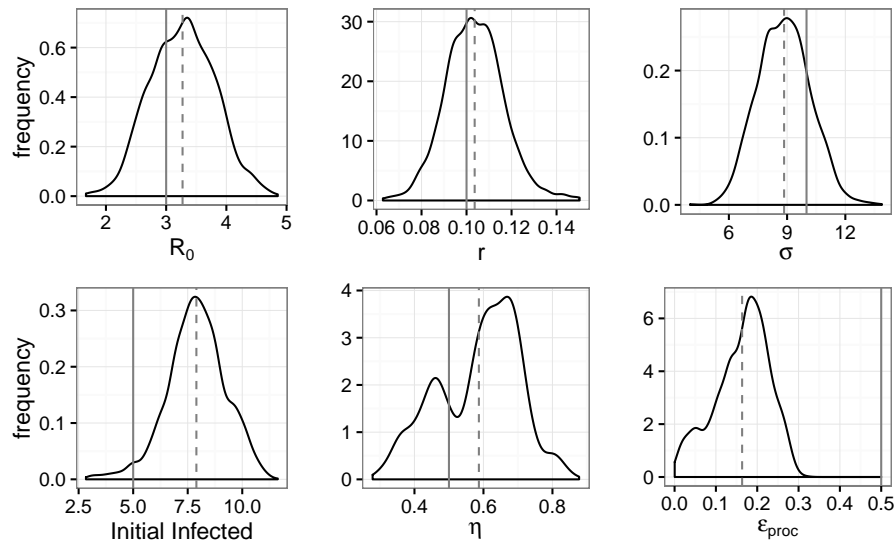


Figure 4.8: As before, the solid grey lines show the true parameter values and the dashed grey lines show the density means.

4.6 HMC MC Fitting

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

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

4.7 HMC MC Densities

The parameter estimation densities from the Stan HMC MC fitting are shown in Figure [4.9].

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.

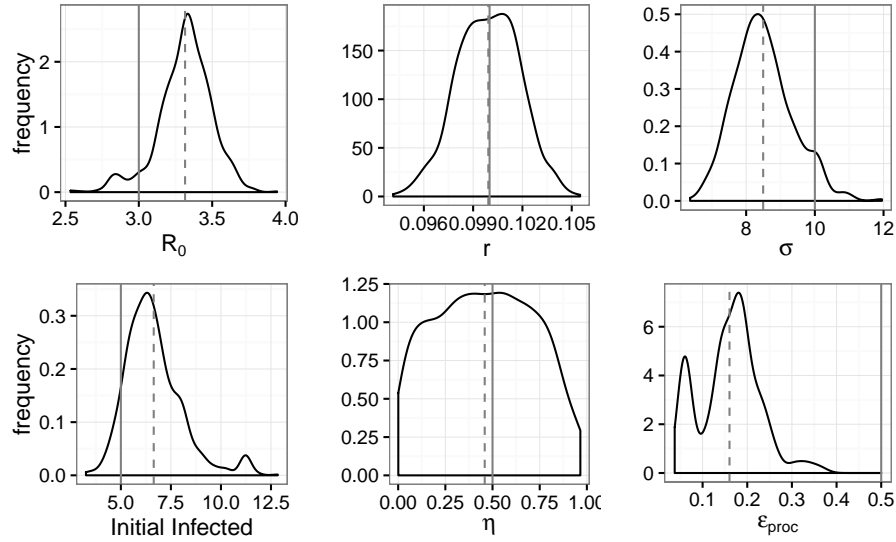


Figure 4.9: As before, the solid grey lines show the true parameter values and the dashed grey lines show the density means.

1 4.8 HMC and Bootstrapping

2 Unlike particle particle-filtering-based approaches, HMC does not produce state es-
 3 timates as a by-product of parameter fitting, but we can use information about
 4 the stochastic nodes related to the noise in the β geometric random walk to recon-
 5 struct state estimates. The results of 100 bootstrap trajectories is shown in Figure
 6 [4.10].

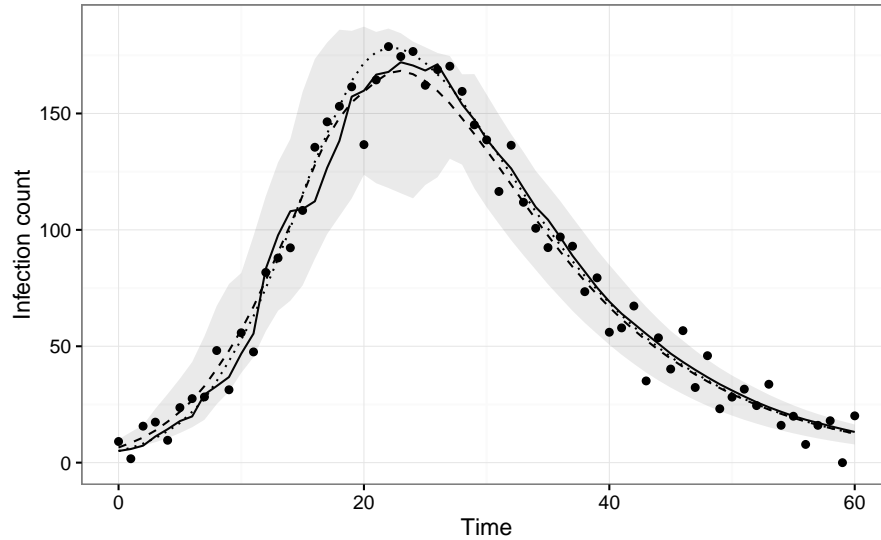


Figure 4.10: Result from 100 HMCBC bootstrap trajectories. The solid line shows the true states, the dots show the data, the dotted line shows the average system behaviour, the dashed line shows the bootstrap mean, and the grey ribbon shows the centre 95th quantile of the bootstrap trajectories.

4.9 Multi-trajectory Parameter Estimation

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

The densities by and large display similar coverage, with the IF2 densities for r and ε_{proc} showing slightly wider coverage than the HMCBC densities for the same parameters.

The running times for each algorithm are summarized in Figure [4.12].

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

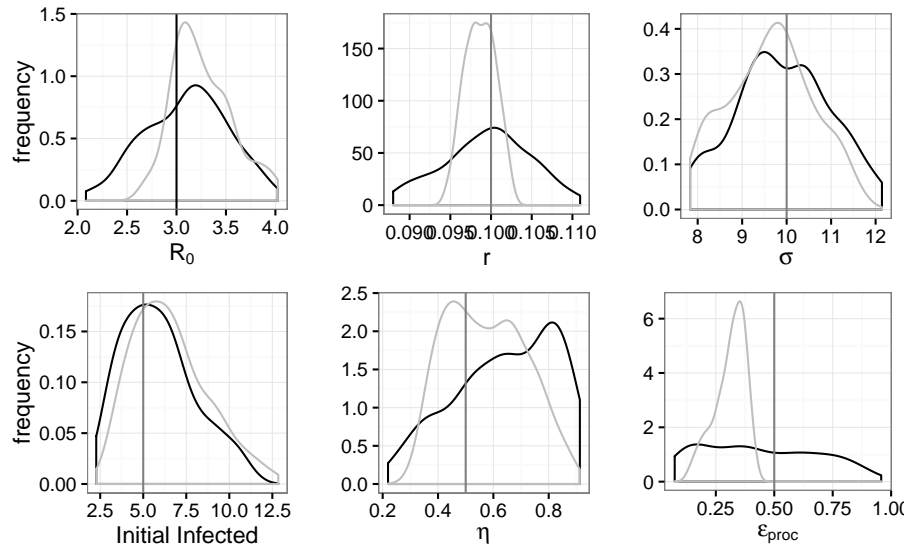


Figure 4.11: IF2 point estimate densities are shown in black and HMCMC point estimate densities are shown in grey. The vertical black lines show the true parameter values.

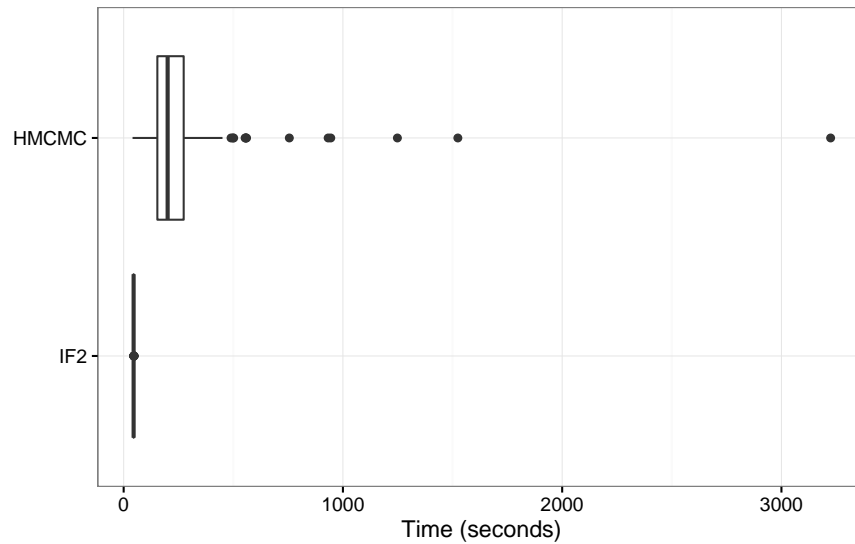


Figure 4.12: Fitting times for IF2 and HMCMC, in seconds. The centre box in each plot shows the centre 50th quantile, with the bold centre line showing the median.

Chapter 5 1

Forecasting Frameworks 2

5.1 Data Setup 3

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. 4
5
6

An example of a simulated system with truncated data can be seen in Figure [5.1]. 7
8

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. 9
10

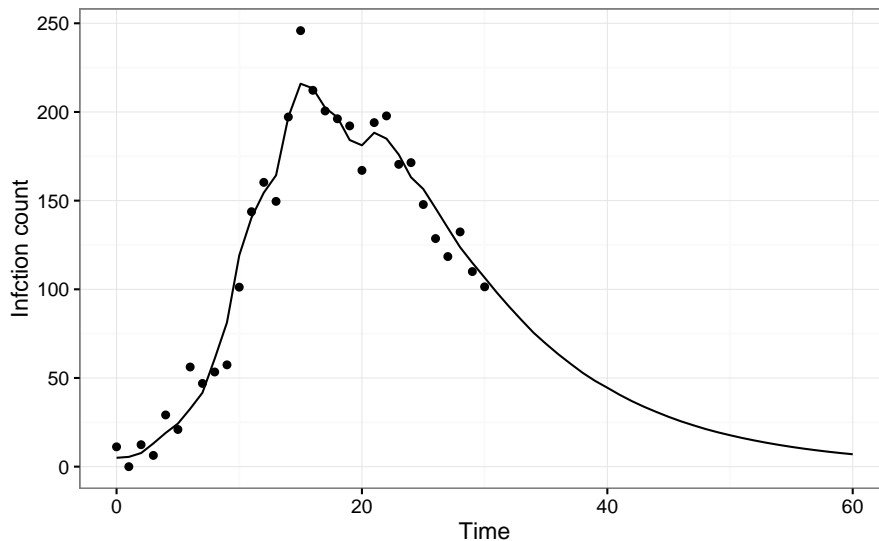


Figure 5.1: Infection count data truncated at $T = 30$. The solid line shows the true underlying system states, and the dots show those states with added observation noise. Parameters used were $R_0 = 3.0$, $r = 0.1$, $\eta = .05$, $\sigma_{proc} = 0.5$, and additive observation noise was drawn from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

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 param-
4 eter maximum likelihood point estimates. Both of these sources of information will
5 be used to produce forecasts by parametric bootstrapping using the final parameter
6 estimates from the particle swarm after the last IF2 pass, then using the newly
7 generated parameter sets along with the system state point estimates from the first
8 fitting to 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 First, we can see the state estimates for each time point produced by the last IF2
12 pass in Figure [5.2].

13 Recall that IF2 is not trying to generate parameter estimation densities, but rather
14 produce a point estimate. Since we wish to determine the approximate distribution
15 of each of the parameters in addition to the point estimate, we must turn to another
16 method, parametric bootstrapping.

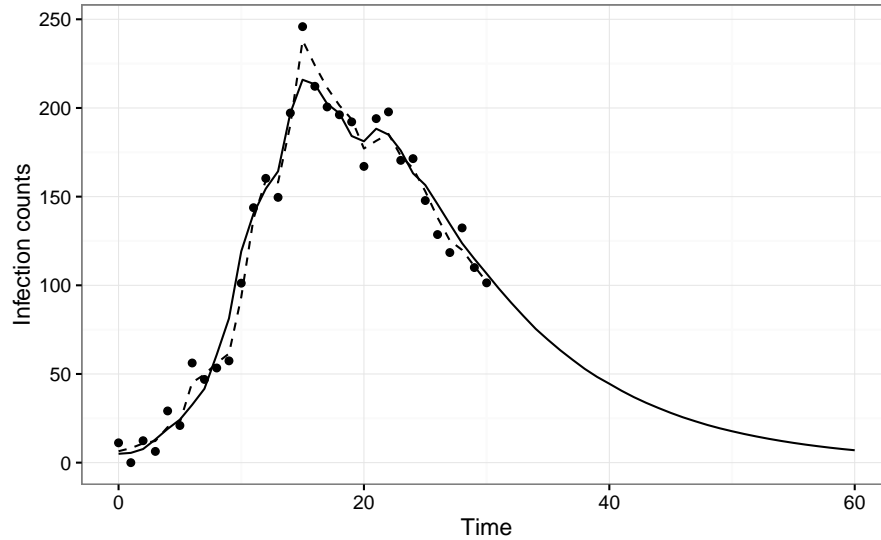


Figure 5.2: Infection count data truncated at $T = 30$ from Figure [5.1]. The dashed line shows IF2's attempt to reconstruct the true underlying state from the observed data points.

5.2.1 Parametric Bootstrapping

The goal of the parametric bootstrap is use an initial density sample θ^* to generate further samples $\theta_1, \theta_2, \dots, \theta_M$. It works by using θ to generate artificial data sets D_1, D_2, \dots, D_M to which we can refit our model of interest and generate new parameter sets.

[I'm still trying to dig up a good paper that talks about applicability to dynamical systems, there will be a paragraph here about it.]

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

5.2.2 IF2 Forecasts

Using the parameter sets $\theta_1, \theta_2, \dots, \theta_M$ and the point estimate of the state provided by the initial IF2 fit, we can use a normal bootstrap to produce estimates of the future state. A plot showing a projection of the data from the previous plots can be seen in Figure [5.3].

We can define a metric to gauge forecast effectiveness by calculating the SSE and dividing that value by the number of values predicted to get the average squared error per point. For the data in Figure [5.3] the value was $\overline{SSE} = 1.67$.

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

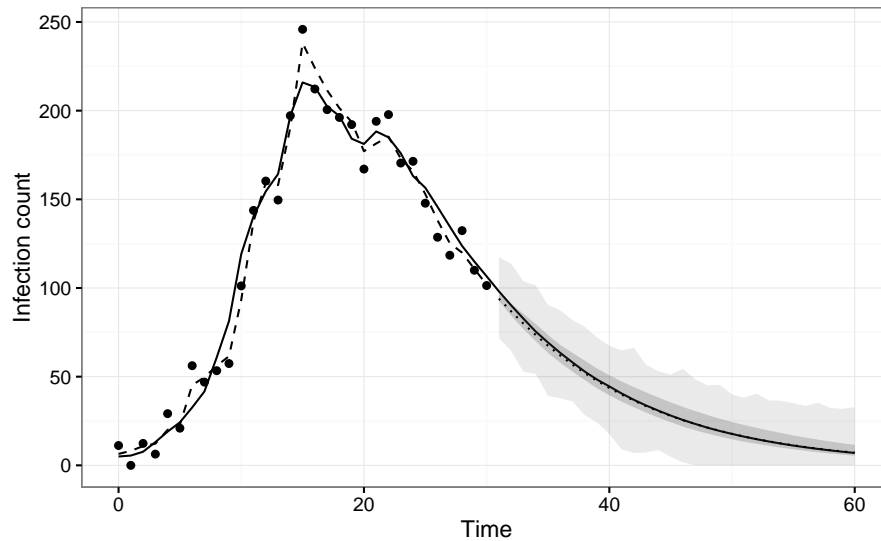


Figure 5.3: Forecast produced by the IF2 / parametric bootstrapping framework. The dotted line shows the mean estimate of the forecasts, the dark grey ribbon shows the centre 95th quantile of the true state estimates, and the lighter grey ribbon shows the centre 95th quantile of the true state estimates with added observation noise drawn from $\mathcal{N}(0, \sigma)$.

5.3 HMCMC

1

For HMCMC we can use a simpler bootstrapping approach. We do not get state estimates directly from the RStan fitting due to the way we implemented the model, but we can construct them using the process noise latent variables. Once we've done this we can forward simulate the system from the state estimate into the future.

2

3

4

5

As before we fit the stochastic SIR model to the partial data, but now perform bootstrapping as described above, and obtain the plot in Figure [5.4].

6

7

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

8

9

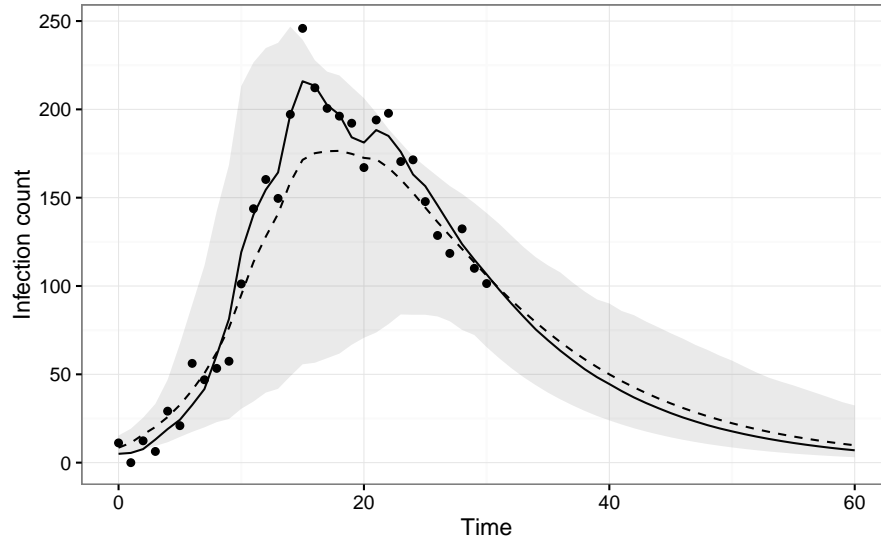


Figure 5.4: Forecast produced by the HMC MC / bootstrapping framework with $M = 200$ trajectories. The dotted line shows the mean estimate of the forecasts, and the grey ribbon shows the centre 95th quantile.

5.4 Truncation vs. Error

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

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

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

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

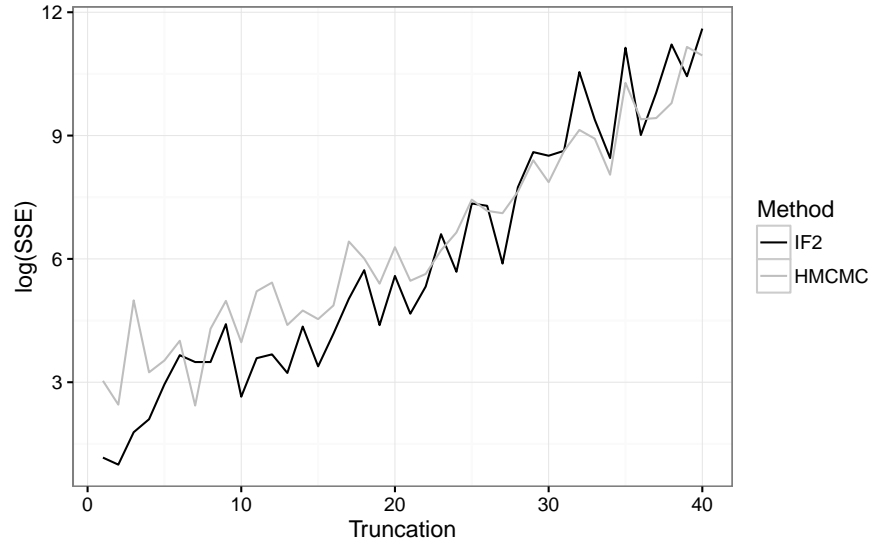


Figure 5.5: Error growth as a function of data truncation amount. Both methods used 200 bootstrap trajectories. Note that the y-axis shows the natural log of the averaged SSE, not the total SSE.

additional effort. The code used here has this capability, but it was not utilised in the comparison so as to accurately represent total computational cost, not potential running time.

1
2
3

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 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 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:

First construct a matrix A and vector b defined as

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

where i ranges over 1 to the length of the library, and j ranges over $[0, E]$. It should be noted that in the above equations and the ones that follow, $x_t(0) = 1$ to account for the linear term in the map.

The weighting function w is defined as

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

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

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

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

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

6.2 S-map Algorithm

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

Algorithm 6: S-map

```

/* Select a starting point */
Input : Time series  $x_1, x_2, \dots, x_T$ , 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.

With that in mind, the only fair way to compare the efficacy of s-mapping to IF2 or Hamiltonian MCMC 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. 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 re-susceptibility rate η through which people become able to be reinfected, and a seasonality factor Γ defined as

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

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

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

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

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

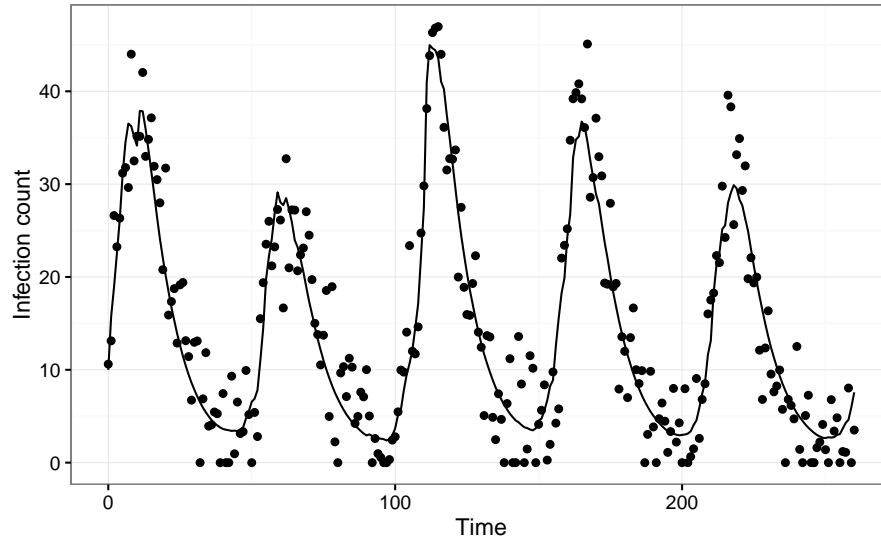


Figure 6.1: Five cycles generated by the SIRS function. The solid line the the true number of cases, dots show case counts with added observation noise. The Parameter values were $R_0 = 3.0$, $\gamma = 0.1$, $\eta = 1$, $\sigma = 5$, and 10 initial cases.

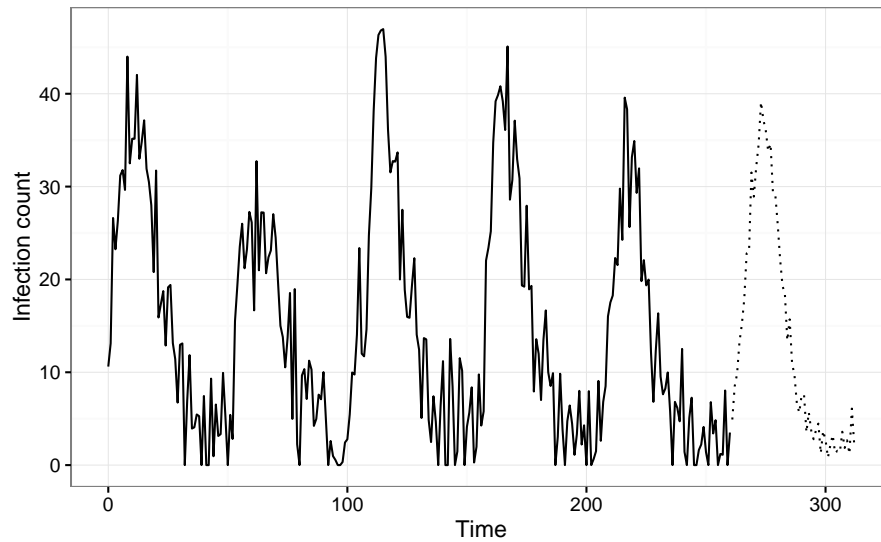


Figure 6.2: S-map applied to the data from the previous figure. The solid line shows the infection counts with observation noise from the previous plot, and the dotted line is the S-map forecast. Parameters chosen were $E = 14$ and $\theta = 3$.

The parameters used in the S-map algorithm to obtain the forecast used in Figure [6.2] were obtained using a grid search of potential parameters outlined in (Sugihara ref). The script is included in the appendices.

6.4 SIRS Model Forecasting

Naturally we wish to compare the efficacy of this comparatively simple technique against the more complex and more computationally taxing frameworks we have established to perform forecasting using IF2 and HMCMC.

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

The results of the simulation are shown in Figure [6.3].

Interestingly, all methods produce roughly the same result, which is to say the spike in each outbreak cycle are difficult to accurately predict. IF2 produces better results than either HMCMC and the S-map for the majority of forecast lengths, with the S-map producing the poorest results with the exception of the second rise in infection rates in which it outperforms the other methods.

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

It is clear from Figure [4.12] that the S-map running times are minute compared to the other methods, but to emphasize the degree: The average running time for the S-map is about 1.49×10^{-1} seconds, for IF2 it is about 4.70×10^4 , and for HMCMC it is about 9.20×10^3 . This is a speed-up of over 316,000x compared to IF2 and over 61,800x compared to HMCMC.

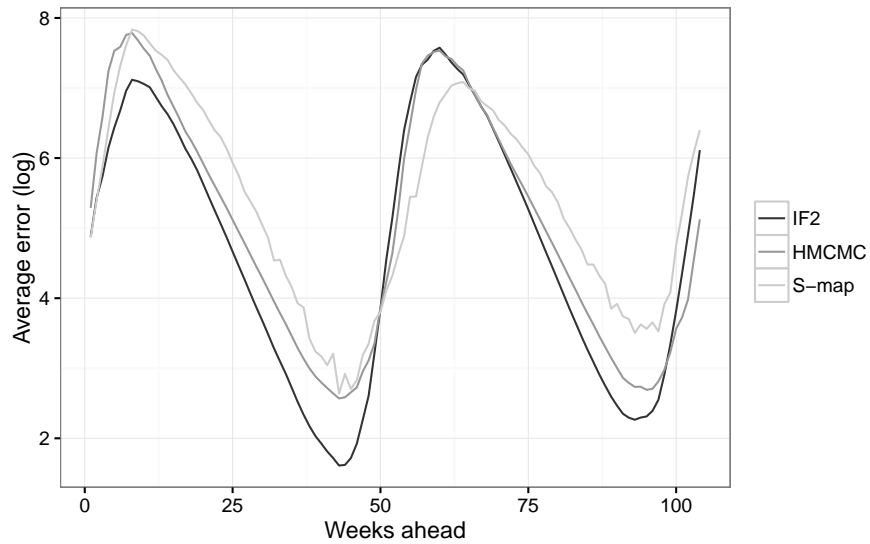


Figure 6.3: Error as a function of forecast length.

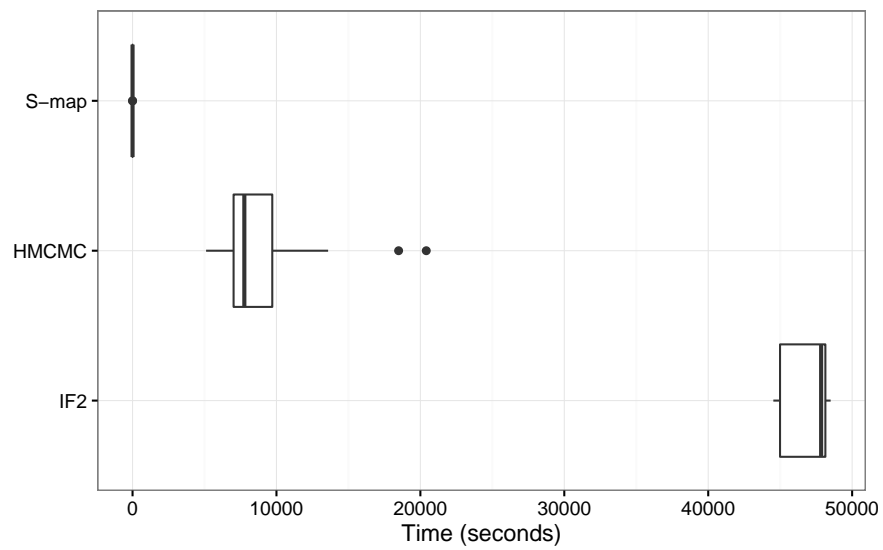


Figure 6.4: Runtimes for producing SIRS forecasts. The box shows the middle 50th quantile, the bold line is the median, and the dots are outliers.

Chapter 7

1

Spatial Epidemics

2

7.1 Spatial SIR

3

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.

4

5

6

7

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

8

9

10

11

12

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

13

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.

14

15

16

17

18

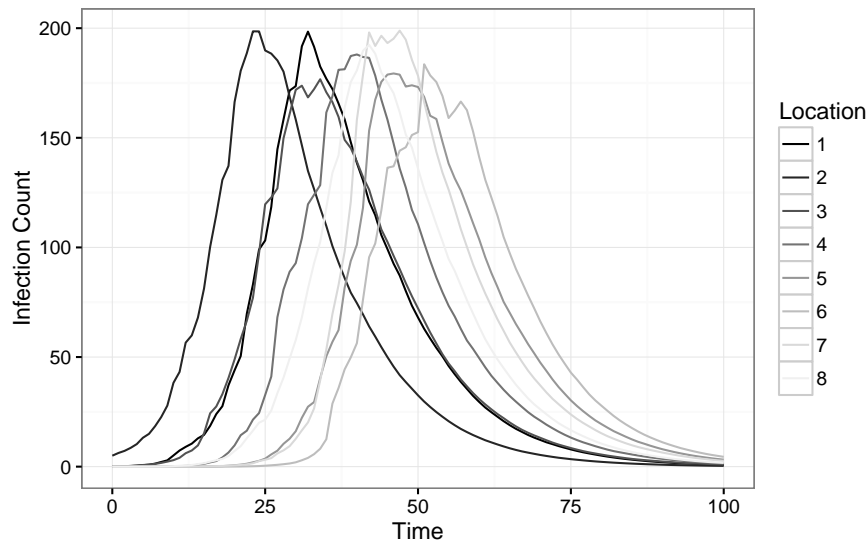


Figure 7.1: Evolution of a spatial epidemic in a ring topology. The outbreak was started with 5 cases in Location 2. Parameters were $R_0 = 3.0$, $\gamma = 0.1$, $\eta = 0.5$, $\sigma_{err} = 0.5$, and $\phi = 0.5$.

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

$$2 \quad \beta_{i,t+1} = \exp \left(\log(\beta_{i,t}) + \eta(\log(\bar{\beta}) - \log(\beta_{i,t})) + \epsilon_t \right). \quad (7.2)$$

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

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

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

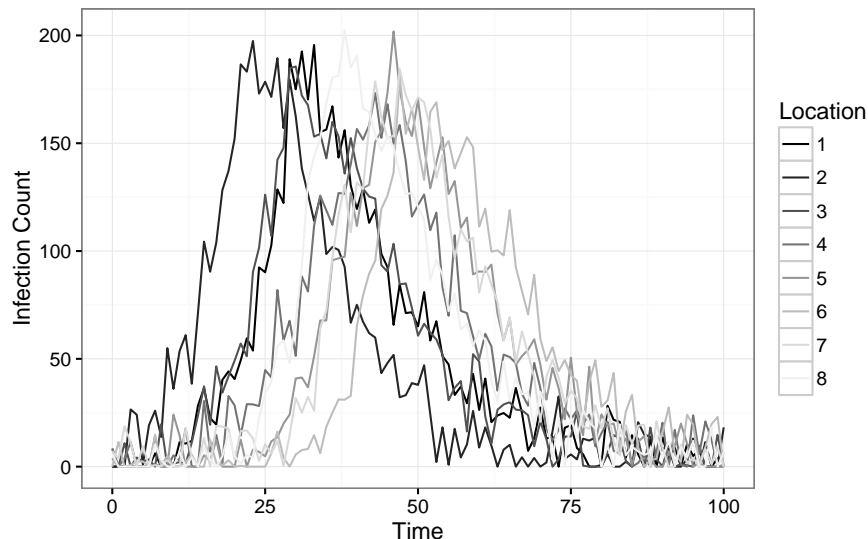


Figure 7.2: Evolution of a spatial epidemic as in Figure [7.1], with added observation noise drawn from $\mathcal{N}(0, 10)$.

7.2 Dewdrop Regression

1

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.

2
3
4
5
6
7
8
9

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

10
11
12
13
14
15
16
17
18
19

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

20
21

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
 3 location is on the smaller side, meaning the S-map will have a high-quality library
 4 from which to build forecasts.

5 **7.3 Spatial Model Forecasting**

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

13 The results are shown in Figure [7.3].

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

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

20 As before, the S-map with Dewdrop Regression runs faster than the other two
 21 methods with a huge margin. It is again hard to see exactly how large the margin
 22 is from the figure due to the scale, but we can examine the average values: the
 23 average running time for S-mapping with Dewdrop Regression was about 249 sec-
 24 onds, whereas the average times for IF2 and HMCMC were about 2.90×10^4 and
 25 3.88×10^4 , respectively. This is a speed-up of just over 116x over IF2 and 156x over
 26 HMCMC.

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

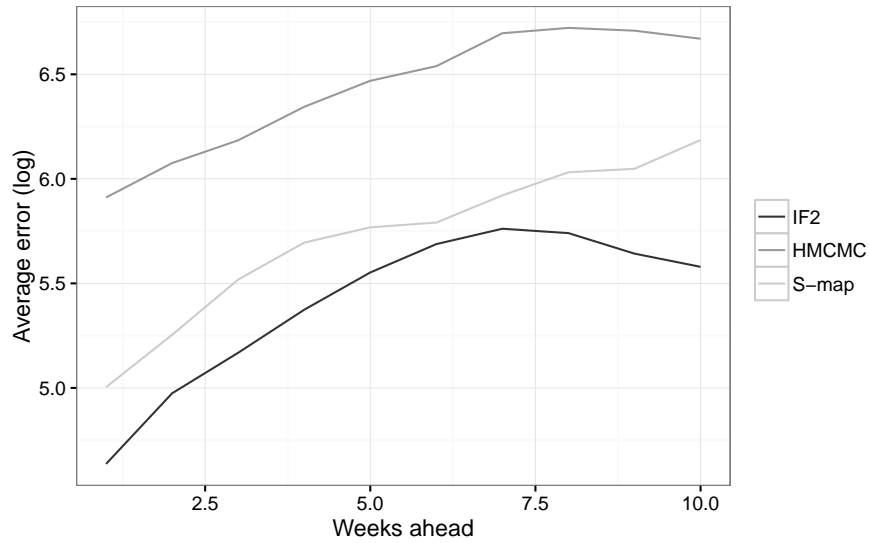


Figure 7.3: Average SSE (log scale) across each location and all trials as a function of the number of weeks ahead in the forecast.

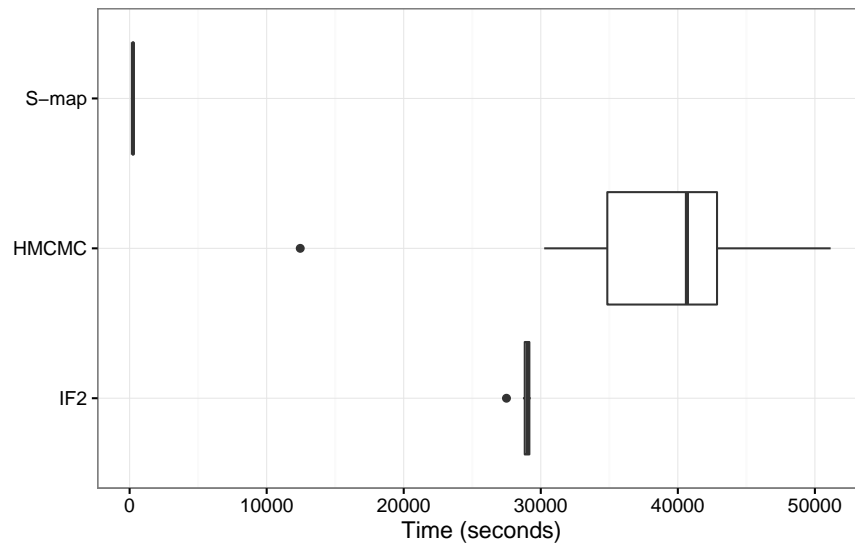


Figure 7.4: Runtimes for producing spatial SIR forecasts. The box shows the middle 50th quantile, the bold line is the median, and the dots are outliers.

Chapter 8

Discussion and Future Directions

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.).

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

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

Of course this raises many problems on the software and algorithm side of computing.

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

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

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

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

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

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

- 3 • Forward evolution of the particles' internal system state using their parameter
 4 state
- 5 • Weighting those state estimates against the data point using the observation
 6 function
- 7 • Particle weight normalizations
- 8 • Resampling from the particle weight distribution
- 9 • Particle parameter perturbations

10 Luckily, 4 of the 5 steps can be individually parallelized and run on a per-particle ba-
 11 sis. The particle weight normalizations, however, cannot. Summation “reductions”
 12 are a well-known problem for parallel algorithms; they can be parallelized to a degree
 13 using binary reduction, but that only reduces the approximate running time from
 14 $\mathcal{O}(n)$ to $\mathcal{O}(\log(n))$. The normalization process requires the particles' weight sum
 15 to be determined, hence the unavoidable obstacle of summation reductions rears its
 16 head. However this is in practice a less-taxing step, and its more demanding siblings
 17 are more amenable to parallelization.

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

23 A future offshoot of this project would be a good parallel implementation of both
 24 the IF2 fitting process and the parametric bootstrapping framework. An ideal
 25 platform for this work would be NVIDIA's Compute Unified Device Architecture
 26 (CUDA) Graphics Processing Unit (GPU) computing framework. While a CUDA
 27 implementation of a spatial epidemic IF2 parameter fitting algorithm was imple-
 28 mented, it lacked a good front-end implementation, R integration, and a parametric
 29 bootstrapping framework and so was not included in the main results of this pa-
 30 per. The code, however, as well as some preliminary results, are included in the
 31 appendices.

32 S-mapping, like the other two methods, is parallelizable to a degree. However, the
 33 S-map is already a great deal faster than the other two methods, and in the worst
 34 case (paired with Dewdrop Regression and applied to a spatiotemporal data set)
 35 still only takes a few minutes to run. Setting this observation aside, if one were
 36 investing in developing a faster S-map implementation, this is certainly possible.
 37 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.

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

```

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)", 18
      ylab = "Infection Count") + 19
46 |   geom_point(aes(y = infec_counts)) + 20
47 |   theme_bw() 21
48 | 22
49 | print(g) 23
50 | ggsave(g, filename="dataplot.pdf", height=4, width=6.5) 24
51 | 25
52 | sPw ← 7 26
53 | datlen ← (T-1)*7 + 1 27
54 | 28
55 | data ← matrix(data = -1, nrow = T+1, ncol = sPw) 29
56 | data[,1] ← infec_counts 30
57 | standata ← as.vector(t(data))[1:datlen] 31
58 | 32
59 | sir_data ← list( T = datlen,      # simulation time 33
      y = standata, # infection count data 34
      N = 500,      # population size 35
      h = 1/sPw )   # step size per day 36
60 | 37
61 | 38
62 | rstan_options(auto_write = TRUE) 39
63 | options(mc.cores = parallel::detectCores()) 40
64 | stan_options ← list( chains = 4,      # number of chains 41
      iter = 2000, # iterations per chain 42
      warmup = 1000, # warmup iterations 43
      thin = 2) # thinning number 44
65 | fit ← stan(file = "d_sirode_euler.stan", 45
      data = sir_data, 46
      chains = stan_options$chains, 47
      iter = stan_options$iter, 48
      warmup = stan_options$warmup, 49
      thin = stan_options$thin ) 50
66 |

```

```

1 77 exfit ← extract(fit, permuted = TRUE, inc_warmup = FALSE)
2 78
3 79 R0points ← exfit$R0
4 80 R0kernel ← qplot(R0points, geom = "density", xlab = expression(R[0])
5      , ylab = "frequency") +
6 81     geom_vline(aes(xintercept=R0), linetype="dashed", size=1,
7      color="grey50") +
8 82     theme_bw()
9 83
10 84 print(R0kernel)
11 85 ggsave(R0kernel, filename="kernelR0.pdf", height=3, width=3.25)
12 86
13 87 rpoints ← exfit$r
14 88 rkernell ← qplot(rpoints, geom = "density", xlab = "r", ylab = "
15     frequency") +
16 89     geom_vline(aes(xintercept=r), linetype="dashed", size=1,
17     color="grey50") +
18 90     theme_bw()
19 91
20 92 print(rkernell)
21 93 ggsave(rkernell, filename="kernelr.pdf", height=3, width=3.25)
22 94
23 95 sigmapoints ← exfit$sigma
24 96 sigmakernel ← qplot(sigmapoints, geom = "density", xlab = expression
25     (sigma), ylab = "frequency") +
26 97     geom_vline(aes(xintercept=sigma), linetype="dashed", size=1,
27     color="grey50") +
28 98     theme_bw()
29 99
30 100 print(sigmakernel)
31 101 ggsave(sigmakernel, filename="kernelsigma.pdf", height=3, width
32     =3.25)
33 102
34 103 infecpoints ← exfit$y0[,2]
35 104 infeckernel ← qplot(infecpoints, geom = "density", xlab = "Initial
36     Infected", ylab = "frequency") +
37 105     geom_vline(aes(xintercept=y_ini[['I']]), linetype="dashed",
38     size=1, color="grey50") +
39 106     theme_bw()
40 107
41 108 print(infeckernel)
42 109 ggsave(infeckernel, filename="kernelinfec.pdf", height=3, width
43     =3.25)
44 110
45 111 exfit ← extract(fit, permuted = FALSE, inc_warmup = FALSE)
46 112 plotdata ← melt(exfit[,,"R0"])
47 113 tracefitR0 ← ggplot() +
48 114     geom_line(data = plotdata,
49 115     aes(x = iterations,
50 116     y = value,

```

```

117         color = factor(chains, labels = 1:stan_      1
                        options$chains))) +          2
118     labs(x = "Sample", y = expression(R[0]), color = "  3
           Chain") +                                4
119     scale_color_brewer(palette="Greys") +          5
120     theme_bw()                                     6
121                                                    7
122 print(tracefitR0)                                  8
123 ggsave(tracefitR0, filename="traceplotR0.pdf", height=4, width=6.5)  9
124                                                    10
125 exfit ← extract(fit, permuted = FALSE, inc_warmup = TRUE) 11
126 plotdata ← melt(exfit[, "R0"])                    12
127 tracefitR0 ← ggplot() +                           13
128     geom_line(data = plotdata,                     14
129               aes(x = iterations,                  15
130                   y = value,                       16
131                   color = factor(chains, labels = 1:stan_ 17
                                   options$chains))) + 18
132     labs(x = "Sample", y = expression(R[0]), color = " 19
           Chain") +                                20
133     scale_color_brewer(palette="Greys") +          21
134     theme_bw()                                     22
135                                                    23
136 print(tracefitR0)                                  24
137 ggsave(tracefitR0, filename="traceplotR0_inc.pdf", height=4, width 25
           =6.5)                                     26
138                                                    27
139 sso ← as.shinystan(fit)                            28
140 sso ← launch_shinystan(sso)                        29

```

A.2 Full Stan code

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

```

1  ## Dexter Barrows                                32
2  ## McMaster University                          33
3  ## 2016                                          34
4                                                    35
5  data {                                          36
6                                                    37
7      int      <lower=1>    T;      // total integration steps  38
8      real     y[T];        // observed number of cases        39
9      int      <lower=1>    N;      // population size         40
10     real     h;           // step size                        41
11                                                    42
12 }                                              43
13                                                    44

```

```

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

```


Appendix B

1

Iterated Filtering

2

B.1 Full R code

3

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

4

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

5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33

```

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

```

```

75 print(R0kernel) 1
76 ggsave(R0kernel, filename="kernelR0.pdf", height=3, width=3.25) 2
77 3
78 rpoints ← paramdata$r 4
79 rkern ← qplot(rpoints, geom = "density", xlab = "r", ylab = " 5
    frequency") + 6
80     geom_vline(aes(xintercept=true_pars[["r"]]), linetype=" 7
        dashed", size=1, color="grey50") + 8
81     theme_bw() 9
82 10
83 print(rkernel) 11
84 ggsave(rkernel, filename="kernelr.pdf", height=3, width=3.25) 12
85 13
86 sigmapoints ← paramdata$sigma 14
87 sigmakern ← qplot(sigmapoints, geom = "density", xlab = expression 15
    (sigma), ylab = "frequency") + 16
88     geom_vline(aes(xintercept=sigma), linetype="dashed", size=1, 17
        color="grey50") + 18
89     theme_bw() 19
90 20
91 print(sigmakern) 21
92 ggsave(sigmakern, filename="kernelsigma.pdf", height=3, width 22
    =3.25) 23
93 24
94 infecpoints ← paramdata$Iinit 25
95 infeckern ← qplot(infecpoints, geom = "density", xlab = "Initial 26
    Infected", ylab = "frequency") + 27
96     geom_vline(aes(xintercept=true_init_cond[["I"]]), linetype=" 28
        dashed", size=1, color="grey50") + 29
97     theme_bw() 30
98 31
99 print(infeckern) 32
100 ggsave(infeckern, filename="kernelinfec.pdf", height=3, width 33
    =3.25) 34
101 35
102 # show grid 36
103 grid.arrange(R0kernel, rkern, sigmakern, infeckern, ncol = 2, 37
    nrow = 2) 38
104 39
105 pdf("if2kernels.pdf", height = 6.5, width = 6.5) 40
106 grid.arrange(R0kernel, rkern, sigmakern, infeckern, ncol = 2, 41
    nrow = 2) 42
107 dev.off() 43
108 #ggsave(filename="if2kernels.pdf", g2, height=6.5, width=6.5) 44

```

1 B.2 Full C++ code

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

```

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

```

```

47     double Iinitmean;    double Iinitstd;      1
48     double Rinitmean;    double Rinitstd;      2
49 };                                             3
50                                             4
51                                             5
52 int timeval_subtract (double *result, struct timeval *x, struct  6
    timeval *y);                                7
53 int check_double(double x,double y);           8
54 void exp_euler_SIR(double h, double t0, double tn, int N, Particle *  9
    particle);                                  10
55 void copyParticle(Particle * dst, Particle * src);      11
56 void perturbParticles(Particle * particles, int N, int NP, int  12
    passnum, double coolrate);                  13
57 bool isCollapsed(Particle * particles, int NP);         14
58 void particleDiagnostics(ParticleInfo * partInfo, Particle *  15
    particles, int NP);                          16
59 NumericMatrix if2(NumericVector * data, int T, int N);   17
60 double randu();                                         18
61 double randn();                                         19
62                                                     20
63 // [[Rcpp::export]]                                    21
64 NumericMatrix if2(NumericVector data, int T, int N) {    22
65                                                     23
66     int      NP          = 2500;                      24
67     int      nPasses     = 50;                         25
68     double   coolrate    = 0.975;                     26
69                                                     27
70     int      i_infec     = I0;                         28
71                                                     29
72     NumericMatrix paramdata(NP, 6);                    30
73                                                     31
74     srand(time(NULL)); // Seed PRNG with system time    32
75                                                     33
76     double w[NP]; // particle weights                  34
77                                                     35
78     Particle particles[NP]; // particle estimates for current  36
        step                                           37
79     Particle particles_old[NP]; // intermediate particle states for  38
        resampling                                     39
80                                                     40
81     printf("Initializing particle states\n");           41
82                                                     42
83     // initialize particle parameter states (seeding)    43
84     for (int n = 0; n < NP; n++) {                     44
85                                                     45
86         double R0can, rcan, sigmacan, Iinitcan;        46
87                                                     47
88         do {                                            48
89             R0can = R0true + R0true*randn();           49
90         } while (R0can < 0);                            50

```

```

1  91         particles[n].R0 = R0can;
2  92
3  93         do {
4  94             rcan = rtrue + rtrue*randn();
5  95         } while (rcan < 0);
6  96         particles[n].r = rcan;
7  97
8  98         do {
9  99             sigmaican = merr + merr*randn();
10 100        } while (sigmaican < 0);
11 101        particles[n].sigma = sigmaican;
12 102
13 103        do {
14 104            Iinitcan = i_infec + i_infec*randn();
15 105        } while (Iinitcan < 0 || N < Iinitcan);
16 106        particles[n].Sinit = N - Iinitcan;
17 107        particles[n].Iinit = Iinitcan;
18 108        particles[n].Rinit = 0.0;
19 109
20 110    }
21 111
22 112    // START PASSES THROUGH DATA
23 113
24 114    printf("Starting filter\n");
25 115    printf("-----\n");
26 116    printf("Pass\n");
27 117
28 118
29 119    for (int pass = 0; pass < nPasses; pass++) {
30 120
31 121        printf("...%d / %d\n", pass, nPasses);
32 122
33 123        perturbParticles(particles, N, NP, pass, coolrate);
34 124
35 125        // initialize particle system states
36 126        for (int n = 0; n < NP; n++) {
37 127
38 128            particles[n].S = particles[n].Sinit;
39 129            particles[n].I = particles[n].Iinit;
40 130            particles[n].R = particles[n].Rinit;
41 131
42 132        }
43 133
44 134        // between-pass perturbations
45 135
46 136        for (int t = 1; t < T; t++) {
47 137
48 138            // between-iteration perturbations
49 139            perturbParticles(particles, N, NP, pass, coolrate);
50 140

```

```

141 // generate individual predictions and weight      1
142 for (int n = 0; n < NP; n++) {                    2
143                                                     3
144     exp_euler_SIR(1.0/10.0, 0.0, 1.0, N, &particles[n]); 4
145                                                     5
146     double merr_par = particles[n].sigma;          6
147     double y_diff   = data[t] - particles[n].I;    7
148                                                     8
149     w[n] = 1.0/(merr_par*sqrt(2.0*M_PI)) * exp( - y_diff  9
        *y_diff / (2.0*merr_par*merr_par) );        10
150                                                     11
151 }                                                  12
152                                                  13
153 // cumulative sum                                14
154 for (int n = 1; n < NP; n++) {                    15
155     w[n] += w[n-1];                                16
156 }                                                  17
157                                                  18
158 // save particle states to resample from          19
159 for (int n = 0; n < NP; n++){                     20
160     copyParticle(&particles_old[n], &particles[n]); 21
161 }                                                  22
162                                                  23
163 // resampling                                    24
164 for (int n = 0; n < NP; n++) {                     25
165                                                     26
166     double w_r = randu() * w[NP-1];                27
167     int i = 0;                                     28
168     while (w_r > w[i]) {                           29
169         i++;                                       30
170     }                                             31
171                                                     32
172     // i is now the index to copy state from       33
173     copyParticle(&particles[n], &particles_old[i]); 34
174                                                     35
175 }                                                  36
176                                                  37
177 }                                                  38
178                                                  39
179 }                                                  40
180                                                  41
181 ParticleInfo pInfo;                               42
182 particleDiagnostics(&pInfo, particles, NP);        43
183                                                  44
184 printf("Parameter results (mean | sd)\n");         45
185 printf("-----\n");                               46
186 printf("R0      %f %f\n", pInfo.R0mean, pInfo.R0sd); 47
187 printf("r       %f %f\n", pInfo.rmean, pInfo.rsd);  48
188 printf("sigma   %f %f\n", pInfo.sigamean, pInfo.sigmasd); 49
189 printf("S_init  %f %f\n", pInfo.Sinitmean, pInfo.Sinit  50

```

```

1 190     printf("I_init      %f %f\n", pInfo.Iinitmean, pInfo.Iinitstd);
2 191     printf("R_init      %f %f\n", pInfo.Rinitmean, pInfo.Rinitstd);
3 192
4 193     printf("\n");
5 194
6 195
7 196
8 197     // Get particle results to pass back to R
9 198
10 199     for (int n = 0; n < NP; n++) {
11 200
12 201         paramdata(n, 0) = particles[n].R0;
13 202         paramdata(n, 1) = particles[n].r;
14 203         paramdata(n, 2) = particles[n].sigma;
15 204         paramdata(n, 3) = particles[n].Sinit;
16 205         paramdata(n, 4) = particles[n].Iinit;
17 206         paramdata(n, 5) = particles[n].Rinit;
18 207
19 208     }
20 209
21 210     return paramdata;
22 211
23 212 }
24 213
25 214
26 215 /* Use the Explicit Euler integration scheme to integrate SIR model
27     forward in time
28 216     double h      - time step size
29 217     double t0     - start time
30 218     double tn     - stop time
31 219     double * y    - current system state; a three-component vector
32                     representing [S I R], susceptible-infected-recovered
33 220
34 221     */
35 222 void exp_euler_SIR(double h, double t0, double tn, int N, Particle *
36     particle) {
37 223
38 224     int num_steps = floor( (tn-t0) / h );
39 225
40 226     double S = particle->S;
41 227     double I = particle->I;
42 228     double R = particle->R;
43 229
44 230     double R0    = particle->R0;
45 231     double r      = particle->r;
46 232     double B      = R0 * r / N;
47 233
48 234     for(int i = 0; i < num_steps; i++) {
49 235         // get derivatives
50 236         double dS = - B*S*I;

```



```

237     double dI = B*S*I - r*I;      1
238     double dR = r*I;              2
239     // step forward by h          3
240     S += h*dS;                    4
241     I += h*dI;                    5
242     R += h*dR;                    6
243 }                                  7
244                                  8
245     particle->S = S;               9
246     particle->I = I;              10
247     particle->R = R;              11
248                                  12
249 }                                  13
250                                  14
251                                  15
252 /* Particle pertubation function to be run between iterations and 16
253    passes                          17
254    */                              18
255 void perturbParticles(Particle * particles, int N, int NP, int 19
256    passnum, double coolrate) {     20
257     double coolcoef = pow(coolrate, passnum); 21
258                                     22
259     double spreadR0    = coolcoef * R0true / 10.0; 23
260     double spreadr     = coolcoef * rtrue  / 10.0; 24
261     double spreadsigma = coolcoef * merr   / 10.0; 25
262     double spreadIinit = coolcoef * I0     / 10.0; 26
263                                     27
264     double R0can, rcan, sigmacan, Iinitcan; 28
265                                     29
266     for (int n = 0; n < NP; n++) { 30
267                                     31
268         do {                          32
269             R0can = particles[n].R0 + spreadR0*randn(); 33
270         } while (R0can < 0);           34
271         particles[n].R0 = R0can;       35
272                                     36
273         do {                          37
274             rcan = particles[n].r + spreadr*randn(); 38
275         } while (rcan < 0);            39
276         particles[n].r = rcan;         40
277                                     41
278         do {                          42
279             sigmacan = particles[n].sigma + spreadsigma*randn(); 43
280         } while (sigmacan < 0);        44
281         particles[n].sigma = sigmacan; 45
282                                     46
283         do {                          47
284             Iinitcan = particles[n].Iinit + spreadIinit*randn(); 48
285                                     49
286                                     50

```

```

1 285         } while (Iinitcan < 0 || Iinitcan > 500);
2 286         particles[n].Iinit = Iinitcan;
3 287         particles[n].Sinit = N - Iinitcan;
4 288
5 289     }
6 290
7 291 }
8 292
9 293
10 294 /* Convenience function for particle resampling process
11 295
12 296 */
13 297 void copyParticle(Particle * dst, Particle * src) {
14 298
15 299     dst->R0      = src->R0;
16 300     dst->r        = src->r;
17 301     dst->sigma    = src->sigma;
18 302     dst->S        = src->S;
19 303     dst->I        = src->I;
20 304     dst->R        = src->R;
21 305     dst->Sinit    = src->Sinit;
22 306     dst->Iinit    = src->Iinit;
23 307     dst->Rinit    = src->Rinit;
24 308
25 309 }
26 310
27 311
28 312 /* Checks to see if particles are collapsed
29 313    This is done by checking if the standard deviations between the
30    particles' parameter
31 314    values are significantly close to one another. Spread threshold
32    may need to be tuned.
33 315
34 316 */
35 317 bool isCollapsed(Particle * particles, int NP) {
36 318
37 319     bool retVal;
38 320
39 321     double R0mean = 0, rmean = 0, sigmamean = 0, Sinitmean = 0,
40         Iinitmean = 0, Rinitmean = 0;
41 322
42 323     // means
43 324
44 325     for (int n = 0; n < NP; n++) {
45 326
46 327         R0mean      += particles[n].R0;
47 328         rmean       += particles[n].r;
48 329         sigmamean   += particles[n].sigma;
49 330         Sinitmean   += particles[n].Sinit;
50 331         Iinitmean   += particles[n].Iinit;

```

```

332         Rinitmean    += particles[n].Rinit;
333
334     }
335
336     R0mean    /= NP;
337     rmean     /= NP;
338     sigmamean /= NP;
339     Sinitmean  /= NP;
340     Iinitmean  /= NP;
341     Rinitmean  /= NP;
342
343     double R0sd = 0, rsd = 0, sigmasd = 0, Sinitsd = 0, Iinitsd =
344         0, Rinitdsd = 0;
345
346     for (int n = 0; n < NP; n++) {
347         R0sd    += ( particles[n].R0 - R0mean ) * ( particles[n].R0
348             - R0mean );
349         rsd     += ( particles[n].r - rmean ) * ( particles[n].r -
350             rmean );
351         sigmasd += ( particles[n].sigma - sigmamean ) * ( particles[
352             n].sigma - sigmamean );
353         Sinitsd += ( particles[n].Sinit - Sinitmean ) * ( particles[
354             n].Sinit - Sinitmean );
355         Iinitsd += ( particles[n].Iinit - Iinitmean ) * ( particles[
356             n].Iinit - Iinitmean );
357         Rinitdsd += ( particles[n].Rinit - Rinitmean ) * ( particles[
358             n].Rinit - Rinitmean );
359
360     }
361
362     R0sd    /= NP;
363     rsd     /= NP;
364     sigmasd /= NP;
365     Sinitsd /= NP;
366     Iinitsd /= NP;
367     Rinitdsd /= NP;
368
369     if ( (R0sd + rsd + sigmasd) < 1e-5)
370         retVal = true;
371     else
372         retVal = false;
373
374     return retVal;
375 }
376
377 void particleDiagnostics(ParticleInfo * partInfo, Particle *
378     particles, int NP) {
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500

```

```

1 374     double  R0mean      = 0.0,
2 375         rmean        = 0.0,
3 376         sigmamean     = 0.0,
4 377         Sinitmean      = 0.0,
5 378         Iinitmean      = 0.0,
6 379         Rinitmean      = 0.0;
7 380
8 381     // means
9 382
10 383     for (int n = 0; n < NP; n++) {
11 384
12 385         R0mean      += particles[n].R0;
13 386         rmean       += particles[n].r;
14 387         sigmamean   += particles[n].sigma;
15 388         Sinitmean   += particles[n].Sinit;
16 389         Iinitmean   += particles[n].Iinit;
17 390         Rinitmean   += particles[n].Rinit;
18 391
19 392     }
20 393
21 394     R0mean      /= NP;
22 395     rmean       /= NP;
23 396     sigmamean   /= NP;
24 397     Sinitmean   /= NP;
25 398     Iinitmean   /= NP;
26 399     Rinitmean   /= NP;
27 400
28 401     // standard deviations
29 402
30 403     double  R0sd      = 0.0,
31 404         rsd         = 0.0,
32 405         sigmasd     = 0.0,
33 406         Sinitsd     = 0.0,
34 407         Iinitsd     = 0.0,
35 408         Rinitsd     = 0.0;
36 409
37 410     for (int n = 0; n < NP; n++) {
38 411
39 412         R0sd      += ( particles[n].R0 - R0mean ) * ( particles[n].R0
40         - R0mean );
41 413         rsd       += ( particles[n].r - rmean ) * ( particles[n].r -
42         rmean );
43 414         sigmasd += ( particles[n].sigma - sigmamean ) * ( particles[
44         n].sigma - sigmamean );
45 415         Sinitsd += ( particles[n].Sinit - Sinitmean ) * ( particles[
46         n].Sinit - Sinitmean );
47 416         Iinitsd += ( particles[n].Iinit - Iinitmean ) * ( particles[
48         n].Iinit - Iinitmean );
49 417         Rinitsd += ( particles[n].Rinit - Rinitmean ) * ( particles[
50         n].Rinit - Rinitmean );

```

418		1
419	}	2
420		3
421	R0sd /= NP;	4
422	rsd /= NP;	5
423	sigmasd /= NP;	6
424	Sinitisd /= NP;	7
425	Iinitisd /= NP;	8
426	Rinitisd /= NP;	9
427		10
428	partInfo->R0mean = R0mean;	11
429	partInfo->R0sd = R0sd;	12
430	partInfo->sigmamean = sigmamean;	13
431	partInfo->sigmasd = sigmasd;	14
432	partInfo->rmean = rmean;	15
433	partInfo->rsd = rsd;	16
434	partInfo->Sinitmean = Sinitmean;	17
435	partInfo->Sinitisd = Sinitisd;	18
436	partInfo->Iinitmean = Iinitmean;	19
437	partInfo->Iinitisd = Iinitisd;	20
438	partInfo->Rinitmean = Rinitmean;	21
439	partInfo->Rinitisd = Rinitisd;	22
440		23
441	}	24
442		25
443	double randu() {	26
444		27
445	return (double) rand() / (double) RAND_MAX;	28
446		29
447	}	30
448		31
449		32
450	/* Return a normally distributed random number with mean 0 and	33
	standard deviation 1	34
451	Uses the polar form of the Box-Muller transformation	35
452	From http://www.design.caltech.edu/erik/Misc/Gaussian.html	36
453	*/	37
454	double randn() {	38
455		39
456	double x1, x2, w, y1;	40
457		41
458	do {	42
459	x1 = 2.0 * randu() - 1.0;	43
460	x2 = 2.0 * randu() - 1.0;	44
461	w = x1 * x1 + x2 * x2;	45
462	} while (w >= 1.0);	46
463		47
464	w = sqrt((-2.0 * log(w)) / w);	48
465	y1 = x1 * w;	49
466		50

```
1 467     return y1;  
2 468  
3 469 }
```

Appendix C

1

Parameter Fitting

2

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   source(stoc_sir_file)
17 13
18 14   if (nTrials < 2)
19 15     ntrials ← 2
20 16
21 17   # unpack if2 first fit data
22 18   # ...parameters
23 19   paramdata_parent ← data.frame( if2data_parent$paramdata )
24 20   names(paramdata_parent) ← c("R0", "r", "sigma", "eta", "berr", "
25 21     Sinit", "Iinit", "Rinit")
26 22   parmeans_parent ← colMeans(paramdata_parent)
27 23   names(parmeans_parent) ← c("R0", "r", "sigma", "eta", "berr", "
28 24     Sinit", "Iinit", "Rinit")
29 25   # ...states
30 26   statedata_parent ← data.frame( if2data_parent$statedata )
31 27   names(statedata_parent) ← c("S", "I", "R", "B")
```



```

26 statemeans_parent ← colMeans(statedata_parent)      1
27 names(statemeans_parent) ← c("S", "I", "R", "B")    2
28                                                     3
29                                                     4
30 ## use parametric bootstrapping to generate forecasts 5
31 ##                                                  6
32 trajectories ← foreach( i = 1:nTrials, .combine = rbind, .packages 7
    = "Rcpp") %dopar% {                                8
33                                                     9
34     source(stoc_sir_file)                          10
35                                                     11
36     ## draw new data                                12
37     ##                                              13
38                                                     14
39     pars ← with( as.list(parmmeans_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(parmmeans_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

```

```

1 71  ## generate the rest of the trajectory
2 72  ##
3 73
4 74  # pack new parameter estimates
5 75  pars ← with( as.list(parmmeans),
6 76               c(R0 = R0,
7 77                 r = r,
8 78                 N = N,
9 79                 eta = eta,
10 80                berr = berr) )
11 81  init_cond ← c(S = statemeans_parent[['S']],
12 82               I = statemeans_parent[['I']],
13 83               R = statemeans_parent[['R']])
14 84
15 85  # generate remaining trajectory part
16 86  sdeout_future ← StocSIR(init_cond, pars, T-Tlim, steps)
17 87  colnames(sdeout_future) ← c('S', 'I', 'R', 'B')
18 88
19 89  return ( c( counts = unname(sdeout_future[, 'I']),
20 90             parmeans,
21 91             time = if2time[['user.self']]) )
22 92
23 93
24 94  }
25 95
26 96  return(trajectories)
27 97
28 98 }

```

30 D.2 RStan Forward Simulator

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

```

33 1 StocSIRstan ← function(y, pars, T, steps, berrvec, bveclim) {
34 2
35 3   out ← matrix(NA, nrow = (T+1), ncol = 4)
36 4
37 5   R0 ← pars[['R0']]
38 6   r ← pars[['r']]
39 7   N ← pars[['N']]
40 8   eta ← pars[['eta']]
41 9   berr ← pars[['berr']]
42 10
43 11  S ← y[['S']]
44 12  I ← y[['I']]
45 13  R ← y[['R']]

```

14		1
15	$B_0 \leftarrow R_0 * r / N$	2
16	$B \leftarrow B_0$	3
17		4
18	$out[1,] \leftarrow c(S, I, R, B)$	5
19		6
20	$h \leftarrow 1 / steps$	7
21		8
22	for (i in 1:(T*steps)) {	9
23		10
24	if (i <= bveclim) {	11
25	$B \leftarrow \exp(\log(B) + \eta * (\log(B_0) - \log(B)) + berrvec[i])$	12
26	} else {	13
27	$B \leftarrow \exp(\log(B) + \eta * (\log(B_0) - \log(B)) + rnorm(1, 0, berr))$	14
28	}	15
29		16
30	$BSI \leftarrow B * S * I$	17
31	$rI \leftarrow r * I$	18
32		19
33	$dS \leftarrow -BSI$	20
34	$dI \leftarrow BSI - rI$	21
35	$dR \leftarrow rI$	22
36		23
37	$S \leftarrow S + h * dS$ #newInf	24
38	$I \leftarrow I + h * dI$ #newInf - h*dR	25
39	$R \leftarrow R + h * dR$ #h*dR	26
40		27
41	if (i %% steps == 0)	28
42	$out[i/steps+1,] \leftarrow c(S, I, R, B)$	29
43		30
44	}	31
45		32
46	return(out)	33
47		34
48	}	35
		36
		37

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

29		1
30	BSI \leftarrow Bfac*B*S*I	2
31	rI \leftarrow r*I	3
32	reR \leftarrow re*R	4
33		5
34	dS \leftarrow -BSI + reR	6
35	dI \leftarrow BSI - rI	7
36	dR \leftarrow rI - reR	8
37		9
38	S \leftarrow S + h*dS #newInf	10
39	I \leftarrow I + h*dI #newInf - h*dR	11
40	R \leftarrow R + h*dR #h*dR	12
41		13
42	if (i %% steps == 0)	14
43	out[i/steps+1,] \leftarrow c(S,I,R,B)	15
44		16
45	}	17
46		18
47	colnames(out) \leftarrow c("S","I","R","B")	19
48	return(out)	20
49		21
50	}	22
51		23
52	### suggested parameters	24
53	#	25
54	# T \leftarrow 200	26
55	# i_infec \leftarrow 10	27
56	# steps \leftarrow 7	28
57	# N \leftarrow 500	29
58	# sigma \leftarrow 5	30
59	#	31
60	# pars \leftarrow 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	37

E.2 SMAP Code

39

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

40

1	library(pracma)	41
2		42
3	smap \leftarrow function(data, E, theta, stepsAhead) {	43
4		44
5	# construct library	45
		46

```

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

```

55	<code>#allPredicttees[i,] ← predictee</code>	1
56		2
57	<code>}</code>	3
58		4
59	<code>return(predictions)</code>	5
60		6
61	<code>}</code>	7

E.3 SMAP Parameter Optimization Code

9

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

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

```

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

```



```

81 quartz()
82 filled.contour(~ssemat)
83
84 #print(ssemat)
85 #cms ← colMeans(ssemat)
86 #rms ← rowMeans(ssemat)
87
88 #Emin ← Elist[which.min(rms)]
89 #thetamin ← thetalist[which.min(cms)]
90 #print(Emin)
91 #print(thetamin)
92
93 mininds ← which(ssemat==min(ssemat),arr.ind=TRUE)
94
95 Emin ← Elist[mininds[, 'row']]
96 thetamin ← thetalist[mininds[, 'col']]
97
98 print(Emin)
99 print(thetamin)

```

E.4 RStan SIRS Code

This code implements a periodic SIRS model in Rstan.

```

1 data {
2
3   int      <lower=1>    T;      // total integration steps
4   real      y[T];      // observed number of cases
5   int      <lower=1>    N;      // population size
6   real      h;          // step size
7
8 }
9
10 parameters {
11
12   real <lower=0, upper=10>    R0;      // R0
13   real <lower=0, upper=10>    r;       // recovery rate
14   real <lower=0, upper=10>    re;      // resusceptibility rate
15   real <lower=0, upper=20>    sigma;   // observation error
16   real <lower=0, upper=30>    Iinit;   // initial infected
17   real <lower=0, upper=1>     eta;     // geometric walk
18   attraction strength
19   real <lower=0, upper=1>     berr;    // beta walk noise
20   real <lower=-1.5, upper=1.5> Bnoise[T]; // Beta vector
21 }
22

```

```

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

```

E.5 IF2 SIRS Code

1

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

2

```

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

```

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

```

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

```

91	<code>srand(time(NULL));</code>	<code>// Seed PRNG with system time</code>	1
92			2
93	<code>double w[NP];</code>	<code>// particle weights</code>	3
94			4
95	<code>Particle particles[NP];</code>	<code>// particle estimates for current</code>	5
	<code>step</code>		6
96	<code>Particle particles_old[NP];</code>	<code>// intermediate particle states for</code>	7
	<code>resampling</code>		8
97			9
98	<code>printf("Initializing particle states\n");</code>		10
99			11
100	<code>// initialize particle parameter states (seeding)</code>		12
101	<code>for (int n = 0; n < NP; n++) {</code>		13
102			14
103	<code>double R0can, rcan, recan, sigmacan, Iinitcan, etacan,</code>		15
	<code>berrcan;</code>		16
104			17
105	<code>do {</code>		18
106	<code> R0can = R0true + R0true*randn();</code>		19
107	<code>} while (R0can < 0);</code>		20
108	<code>particles[n].R0 = R0can;</code>		21
109			22
110	<code>do {</code>		23
111	<code> rcan = rtrue + rtrue*randn();</code>		24
112	<code>} while (rcan < 0);</code>		25
113	<code>particles[n].r = rcan;</code>		26
114			27
115	<code>do {</code>		28
116	<code> recan = retrue + retrue*randn();</code>		29
117	<code>} while (recan < 0);</code>		30
118	<code>particles[n].re = recan;</code>		31
119			32
120	<code>particles[n].B = (double) R0can * rcan / N;</code>		33
121			34
122	<code>do {</code>		35
123	<code> sigmacan = merr + merr*randn();</code>		36
124	<code>} while (sigmacan < 0);</code>		37
125	<code>particles[n].sigma = sigmacan;</code>		38
126			39
127	<code>do {</code>		40
128	<code> etacan = etatrue + PSC*etatrue*randn();</code>		41
129	<code>} while (etacan < 0 etacan > 1);</code>		42
130	<code>particles[n].eta = etacan;</code>		43
131			44
132	<code>do {</code>		45
133	<code> berrcan = berrtrue + PSC*berrtrue*randn();</code>		46
134	<code>} while (berrcan < 0);</code>		47
135	<code>particles[n].berr = berrcan;</code>		48
136			49
137	<code>do {</code>		50

```

1 138         Iinitcan = I0 + I0*randn();
2 139     } while (Iinitcan < 0 || N < Iinitcan);
3 140     particles[n].Sinit = N - Iinitcan;
4 141     particles[n].Iinit = Iinitcan;
5 142     particles[n].Rinit = 0.0;
6 143
7 144 }
8 145
9 146 // START PASSES THROUGH DATA
10 147
11 148 printf("Starting filter\n");
12 149 printf("-----\n");
13 150 printf("Pass\n");
14 151
15 152
16 153 for (int pass = 0; pass < nPasses; pass++) {
17 154
18 155     printf("...%d / %d\n", pass, nPasses);
19 156
20 157     // reset particle system evolution states
21 158     for (int n = 0; n < NP; n++) {
22 159
23 160         particles[n].S = particles[n].Sinit;
24 161         particles[n].I = particles[n].Iinit;
25 162         particles[n].R = particles[n].Rinit;
26 163         particles[n].B = (double) particles[n].R0 * particles[n
27         ].r / N;
28 164
29 165     }
30 166
31 167     if (pass == (nPasses-1)) {
32 168         State sMeans;
33 169         getStateMeans(&sMeans, particles, NP);
34 170         statemeans(0,0) = sMeans.S;
35 171         statemeans(0,1) = sMeans.I;
36 172         statemeans(0,2) = sMeans.R;
37 173     }
38 174
39 175     for (int t = 1; t < T; t++) {
40 176
41 177         // generate individual predictions and weight
42 178         for (int n = 0; n < NP; n++) {
43 179
44 180             exp_euler_SIRS(1.0/7.0, (double) t-1, (double) t, N,
45             &particles[n]);
46 181
47 182             double merr_par = particles[n].sigma;
48 183             double y_diff = data[t] - particles[n].I;
49 184
50 185             w[n] = 1.0/(merr_par*sqrt(2.0*M_PI)) * exp( - y_diff

```

	<code>*y_diff / (2.0*merr_par*merr_par));</code>	1
186		2
187	<code>}</code>	3
188		4
189	<code>// cumulative sum</code>	5
190	<code>for (int n = 1; n < NP; n++) {</code>	6
191	<code> w[n] += w[n-1];</code>	7
192	<code>}</code>	8
193		9
194	<code>// save particle states to resample from</code>	10
195	<code>for (int n = 0; n < NP; n++){</code>	11
196	<code> copyParticle(&particles_old[n], &particles[n]);</code>	12
197	<code>}</code>	13
198		14
199	<code>// resampling</code>	15
200	<code>for (int n = 0; n < NP; n++) {</code>	16
201		17
202	<code> double w_r = randu() * w[NP-1];</code>	18
203	<code> int i = 0;</code>	19
204	<code> while (w_r > w[i]) {</code>	20
205	<code> i++;</code>	21
206	<code> }</code>	22
207		23
208	<code> // i is now the index to copy state from</code>	24
209	<code> copyParticle(&particles[n], &particles_old[i]);</code>	25
210		26
211	<code>}</code>	27
212		28
213	<code>// between-iteration perturbations, not after last time</code>	29
	<code>step</code>	30
214	<code>if (t < (T-1))</code>	31
215	<code> perturbParticles(particles, N, NP, pass, coolrate);</code>	32
216		33
217	<code>if (pass == (nPasses-1)) {</code>	34
218	<code> State sMeans;</code>	35
219	<code> getStateMeans(&sMeans, particles, NP);</code>	36
220	<code> statemeans(t,0) = sMeans.S;</code>	37
221	<code> statemeans(t,1) = sMeans.I;</code>	38
222	<code> statemeans(t,2) = sMeans.R;</code>	39
223	<code>}</code>	40
224		41
225	<code>}</code>	42
226		43
227	<code>ParticleInfo pInfo;</code>	44
228	<code>particleDiagnostics(&pInfo, particles, NP);</code>	45
229		46
230	<code>means(pass, 0) = pInfo.R0mean;</code>	47
231	<code>means(pass, 1) = pInfo.rmean;</code>	48
232	<code>means(pass, 2) = pInfo.remean;</code>	49
233	<code>means(pass, 3) = pInfo.sigamean;</code>	50

```

1 234     means(pass, 4) = pInfo.etamean;
2 235     means(pass, 5) = pInfo.berrmean;
3 236     means(pass, 6) = pInfo.Sinitmean;
4 237     means(pass, 7) = pInfo.Iinitmean;
5 238     means(pass, 8) = pInfo.Rinitmean;
6 239
7 240     sds(pass, 0) = pInfo.R0sd;
8 241     sds(pass, 1) = pInfo.rsd;
9 242     sds(pass, 2) = pInfo.resd;
10 243     sds(pass, 3) = pInfo.sigmasd;
11 244     sds(pass, 4) = pInfo.etasd;
12 245     sds(pass, 5) = pInfo.berrsd;
13 246     sds(pass, 6) = pInfo.Sinitds;
14 247     sds(pass, 7) = pInfo.Iinitds;
15 248     sds(pass, 8) = pInfo.Rinitds;
16 249
17 250     // between-pass perturbations, not after last pass
18 251     if (pass < (nPasses + 1))
19 252         perturbParticles(particles, N, NP, pass, coolrate);
20 253
21 254 }
22 255
23 256 ParticleInfo pInfo;
24 257 particleDiagnostics(&pInfo, particles, NP);
25 258
26 259 printf("Parameter results (mean | sd)\n");
27 260 printf("-----\n");
28 261 printf("R0      %f %f\n", pInfo.R0mean, pInfo.R0sd);
29 262 printf("r      %f %f\n", pInfo.rmean, pInfo.rsd);
30 263 printf("re     %f %f\n", pInfo.remean, pInfo.resd);
31 264 printf("sigma   %f %f\n", pInfo.sigamean, pInfo.sigmasd);
32 265 printf("eta     %f %f\n", pInfo.etamean, pInfo.etasd);
33 266 printf("berr    %f %f\n", pInfo.berrmean, pInfo.berrsd);
34 267 printf("S_init  %f %f\n", pInfo.Sinitmean, pInfo.Sinitds);
35 268 printf("I_init   %f %f\n", pInfo.Iinitmean, pInfo.Iinitds);
36 269 printf("R_init   %f %f\n", pInfo.Rinitmean, pInfo.Rinitds);
37 270
38 271 printf("\n");
39 272
40 273
41 274
42 275 // Get particle results to pass back to R
43 276
44 277 for (int n = 0; n < NP; n++) {
45 278
46 279     paramdata(n, 0) = particles[n].R0;
47 280     paramdata(n, 1) = particles[n].r;
48 281     paramdata(n, 2) = particles[n].re;
49 282     paramdata(n, 3) = particles[n].sigma;
50 283     paramdata(n, 4) = particles[n].eta;

```



```

284     paramdata(n, 5) = particles[n].berr;
285     paramdata(n, 6) = particles[n].Sinit;
286     paramdata(n, 7) = particles[n].Iinit;
287     paramdata(n, 8) = particles[n].Rinit;
288
289 }
290
291 for (int n = 0; n < NP; n++) {
292
293     statedata(n, 0) = particles[n].S;
294     statedata(n, 1) = particles[n].I;
295     statedata(n, 2) = particles[n].R;
296     statedata(n, 3) = particles[n].B;
297
298 }
299
300
301
302     return Rcpp::List::create( Rcpp::Named("paramdata") = paramdata
303                               ,
304                               Rcpp::Named("means") = means,
305                               Rcpp::Named("statemeans") =
306                                   statemeans,
307                               Rcpp::Named("statedata") = statedata
308                               ,
309                               Rcpp::Named("sds") = sds);
310 }
311
312 /* Use the Explicit Euler integration scheme to integrate SIR model
313    forward in time
314    double h      - time step size
315    double t0     - start time
316    double tn     - stop time
317    double * y    - current system state; a three-component vector
318                   representing [S I R], susceptible-infected-recovered
319
320    */
321 void exp_euler_SIRS(double h, double t0, double tn, int N, Particle
322 * particle) {
323
324     int num_steps = floor( (tn-t0) / h );
325
326     double S = particle->S;
327     double I = particle->I;
328     double R = particle->R;
329
330     double R0 = particle->R0;
331     double r = particle->r;

```

```

1 328     double re    = particle->re;
2 329     double B0    = R0 * r / N;
3 330     double eta    = particle->eta;
4 331     double berr   = particle->berr;
5 332
6 333     double B = particle->B;
7 334
8 335     for(int i = 0; i < num_steps; i++) {
9 336
10 337         //double Bfac = 0.5 - 0.95*cos( (2.0*M_PI/365)*(t0*num_steps
11         +i) )/2.0;
12 338         double Bfac = exp(2*cos((2*M_PI/365)*(t0*num_steps+i)) - 2);
13 339         B = exp( log(B) + eta*(log(B0) - log(B)) + berr*randn() );
14 340
15 341         double BSI = Bfac*B*S*I;
16 342         double rI  = r*I;
17 343         double reR = re*R;
18 344
19 345         // get derivatives
20 346         double dS = - BSI + reR;
21 347         double dI = BSI - rI;
22 348         double dR = rI - reR;
23 349
24 350         // step forward by h
25 351         S += h*dS;
26 352         I += h*dI;
27 353         R += h*dR;
28 354
29 355     }
30 356
31 357     particle->S = S;
32 358     particle->I = I;
33 359     particle->R = R;
34 360     particle->B = B;
35 361
36 362 }
37 363
38 364
39 365 /* Particle pertubation function to be run between iterations and
40     passes
41 366
42 367 */
43 368 void perturbParticles(Particle * particles, int N, int NP, int
44     passnum, double coolrate) {
45 369
46 370     //double coolcoef = exp( - (double) passnum / coolrate );
47 371     double coolcoef = pow(coolrate, passnum);
48 372
49 373
50 374     double spreadR0      = coolcoef * R0true / 10.0;

```

```

375 double spreadr      = coolcoef * rtrue / 10.0;      1
376 double spreadre     = coolcoef * retrue / 10.0;     2
377 double spreadsigma  = coolcoef * merr / 10.0;       3
378 double spreadIinit  = coolcoef * I0 / 10.0;         4
379 double spreadeta    = coolcoef * etatrue / 10.0;    5
380 double spreadberr   = coolcoef * berrtrue / 10.0;   6
381                                                         7
382                                                         8
383 double R0can, rcan, recan, sigmacan, Iinitcan, etacan, berrcan; 9
384                                                         10
385 for (int n = 0; n < NP; n++) { 11
386                                                         12
387     do { 13
388         R0can = particles[n].R0 + spreadR0*randn(); 14
389     } while (R0can < 0); 15
390     particles[n].R0 = R0can; 16
391                                                         17
392     do { 18
393         rcan = particles[n].r + spreadr*randn(); 19
394     } while (rcan < 0); 20
395     particles[n].r = rcan; 21
396                                                         22
397     do { 23
398         recan = particles[n].re + spreadre*randn(); 24
399     } while (recan < 0); 25
400     particles[n].re = recan; 26
401                                                         27
402     do { 28
403         sigmacan = particles[n].sigma + spreadsigma*randn(); 29
404     } while (sigmacan < 0); 30
405     particles[n].sigma = sigmacan; 31
406                                                         32
407     do { 33
408         etacan = particles[n].eta + PSC*spreadeta*randn(); 34
409     } while (etacan < 0 || etacan > 1); 35
410     particles[n].eta = etacan; 36
411                                                         37
412     do { 38
413         berrcan = particles[n].berr + PSC*spreadberr*randn(); 39
414     } while (berrcan < 0); 40
415     particles[n].berr = berrcan; 41
416                                                         42
417     do { 43
418         Iinitcan = particles[n].Iinit + spreadIinit*randn(); 44
419     } while (Iinitcan < 0 || Iinitcan > 500); 45
420     particles[n].Iinit = Iinitcan; 46
421     particles[n].Sinit = N - Iinitcan; 47
422                                                         48
423 } 49
424                                                         50

```

```

1 425 }
2 426
3 427
4 428 /* Convenience function for particle resampling process
5 429
6 430 */
7 431 void copyParticle(Particle * dst, Particle * src) {
8 432
9 433     dst->R0      = src->R0;
10 434     dst->r       = src->r;
11 435     dst->re      = src->re;
12 436     dst->sigma   = src->sigma;
13 437     dst->eta     = src->eta;
14 438     dst->berr    = src->berr;
15 439     dst->B       = src->B;
16 440     dst->S       = src->S;
17 441     dst->I       = src->I;
18 442     dst->R       = src->R;
19 443     dst->Sinit   = src->Sinit;
20 444     dst->Iinit   = src->Iinit;
21 445     dst->Rinit   = src->Rinit;
22 446
23 447 }
24 448
25 449 void particleDiagnostics(ParticleInfo * partInfo, Particle *
26     particles, int NP) {
27 450
28 451     double   R0mean      = 0.0,
29 452             rmean       = 0.0,
30 453             remean      = 0.0,
31 454             sigmamean   = 0.0,
32 455             etamean     = 0.0,
33 456             berrmean    = 0.0,
34 457             Sinitmean   = 0.0,
35 458             Iinitmean   = 0.0,
36 459             Rinitmean   = 0.0;
37 460
38 461     // means
39 462
40 463     for (int n = 0; n < NP; n++) {
41 464
42 465         R0mean      += particles[n].R0;
43 466         rmean       += particles[n].r;
44 467         remean      += particles[n].re;
45 468         etamean     += particles[n].eta;
46 469         berrmean    += particles[n].berr;
47 470         sigmamean   += particles[n].sigma;
48 471         Sinitmean   += particles[n].Sinit;
49 472         Iinitmean   += particles[n].Iinit;
50 473         Rinitmean   += particles[n].Rinit;

```

474		1
475	}	2
476		3
477	R0mean /= NP;	4
478	rmean /= NP;	5
479	remean /= NP;	6
480	sigmamean /= NP;	7
481	etamean /= NP;	8
482	berrmean /= NP;	9
483	Sinitmean /= NP;	10
484	Iinitmean /= NP;	11
485	Rinitmean /= NP;	12
486		13
487	// standard deviations	14
488		15
489	double R0sd = 0.0,	16
490	rsd = 0.0,	17
491	resd = 0.0,	18
492	sigmasd = 0.0,	19
493	etasd = 0.0,	20
494	berrsd = 0.0,	21
495	Sinitsd = 0.0,	22
496	Iinitsd = 0.0,	23
497	Rinitsd = 0.0;	24
498		25
499	for (int n = 0; n < NP; n++) {	26
500		27
501	R0sd += (particles[n].R0 - R0mean) * (particles[n].R0	28
	- R0mean);	29
502	rsd += (particles[n].r - rmean) * (particles[n].r -	30
	rmean);	31
503	resd += (particles[n].re - rmean) * (particles[n].re -	32
	rmean);	33
504	sigmasd += (particles[n].sigma - sigmamean) * (particles[34
	n].sigma - sigmamean);	35
505	etasd += (particles[n].eta - etamean) * (particles[n].	36
	eta - etamean);	37
506	berrsd += (particles[n].berr - berrmean) * (particles[n]	38
].berr - berrmean);	39
507	Sinitsd += (particles[n].Sinit - Sinitmean) * (particles[40
	n].Sinit - Sinitmean);	41
508	Iinitsd += (particles[n].Iinit - Iinitmean) * (particles[42
	n].Iinit - Iinitmean);	43
509	Rinitsd += (particles[n].Rinit - Rinitmean) * (particles[44
	n].Rinit - Rinitmean);	45
510		46
511	}	47
512		48
513	R0sd /= NP;	49
514	rsd /= NP;	50

```

1 515     resd          /= NP;
2 516     sigmasd       /= NP;
3 517     etasd         /= NP;
4 518     berrsd         /= NP;
5 519     Sinitsd        /= NP;
6 520     Iinitsd        /= NP;
7 521     Rinitsd        /= NP;
8 522
9 523     partInfo->R0mean    = R0mean;
10 524    partInfo->R0sd      = R0sd;
11 525    partInfo->rmean     = rmean;
12 526    partInfo->rsd       = rsd;
13 527    partInfo->remean    = remean;
14 528    partInfo->resd      = resd;
15 529    partInfo->sigmamean = sigmamean;
16 530    partInfo->sigmasd   = sigmasd;
17 531    partInfo->etamean   = etamean;
18 532    partInfo->etasd     = etasd;
19 533    partInfo->berrmean   = berrmean;
20 534    partInfo->berrsd    = berrsd;
21 535    partInfo->Sinitmean = Sinitmean;
22 536    partInfo->Sinitsd   = Sinitsd;
23 537    partInfo->Iinitmean = Iinitmean;
24 538    partInfo->Iinitsd   = Iinitsd;
25 539    partInfo->Rinitmean = Rinitmean;
26 540    partInfo->Rinitsd   = Rinitsd;
27 541
28 542 }
29 543
30 544 double randu() {
31 545
32 546     return (double) rand() / (double) RAND_MAX;
33 547
34 548 }
35 549
36 550 void getStateMeans(State * state, Particle* particles, int NP) {
37 551
38 552     double Smean = 0, Imean = 0, Rmean = 0;
39 553
40 554     for (int n = 0; n < NP; n++) {
41 555         Smean += particles[n].S;
42 556         Imean += particles[n].I;
43 557         Rmean += particles[n].R;
44 558     }
45 559
46 560     state->S = (double) Smean / NP;
47 561     state->I = (double) Imean / NP;
48 562     state->R = (double) Rmean / NP;
49 563
50 564 }

```

565		1
566		2
567	/* Return a normally distributed random number with mean 0 and	3
	standard deviation 1	4
568	Uses the polar form of the Box-Muller transformation	5
569	From http://www.design.caltech.edu/erik/Misc/Gaussian.html	6
570	*/	7
571	double randn() {	8
572		9
573	double x1, x2, w, y1;	10
574		11
575	do {	12
576	x1 = 2.0 * randu() - 1.0;	13
577	x2 = 2.0 * randu() - 1.0;	14
578	w = x1 * x1 + x2 * x2;	15
579	} while (w >= 1.0);	16
580		17
581	w = sqrt((-2.0 * log(w)) / w);	18
582	y1 = x1 * w;	19
583		20
584	return y1;	21
585		22
586	}	23

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']]
```


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
48		23
49	for (loc in 1:nloc) {	24
50	n ← neinum[loc]	25
51	sphi ← 1 - phi*(n/(n+1))	26
52	ophi ← phi/(n+1)	27
53	nBIsu ← B[neibmat[loc,1:n]] %*% I[neibmat[loc,1:n]]	28
54	BSI[loc] ← S[loc]*(sphi*B[loc]*I[loc] + ophi*nBIsu)	29
55	}	30
56	#if(i == 1)	31
57	# print(BSI)	32
58		33
59	rI ← r*I	34
60		35
61	dS ← -BSI	36
62	dI ← BSI - rI	37
63	dR ← rI	38
64		39
65	S ← S + h*dS	40
66	I ← I + h*dI	41
67	R ← R + h*dR	42
68		43
69	if (i %% steps == 0) {	44
70	out[, ,i/steps+1] ← cbind(S,I,R,B)	45
71	}	46
72		47
73	}	48
74		49
		50

```

1 75     #out[, ,2] ← cbind(S,I,R,B)
2 76
3 77     return(out)
4 78
5 79 }
6 80
7 81 ### Suggested parameters
8 82 #
9 83 # T         ← 60
10 84 # i_infec ← 5
11 85 # steps    ← 7
12 86 # N        ← 500
13 87 # sigma    ← 10
14 88 #
15 89 # pars ← c(R0 = 3.0,      # new infected people per infected person
16 90 #         r = 0.1,      # recovery rate
17 91 #         N = 500,      # population size
18 92 #         eta = 0.5,    # geometric random walk
19 93 #         berr = 0.5)   # Beta geometric walk noise
20

```

21 F.2 RStan Spatial SIR Code

22 This code implements a Spatial SIR model in Rstan.

```

23
24 1 data {
25 2
26 3     int      <lower=1>    T;      // total integration steps
27 4     int      <lower=1>    nloc;   // number of locations
28 5     real     y[nloc, T];  // observed number of cases
29 6     int      <lower=1>    N;      // population size
30 7     real     h;          // step size
31 8     int      <lower=0>    neinum[nloc]; // number of neighbors
32     each location has
33 9     int      neibmat[nloc, nloc]; // neighbor list for
34     each location
35 10
36 11 }
37 12
38 13 parameters {
39 14
40 15     real <lower=0, upper=10>    R0;      // R0
41 16     real <lower=0, upper=10>    r;      // recovery rate
42 17     real <lower=0, upper=20>    sigma;  // observation error
43 18     real <lower=0, upper=30>    Iinit[nloc]; // initial
44     infected for each location
45 19     real <lower=0, upper=1>    eta;     // geometric walk
46     attraction strength

```

20	real <lower=0, upper=1>	berr;	// beta walk noise	1
21	real <lower=-1.5, upper=1.5>	Bnoise[nloc,T];	// Beta vector	2
22	real <lower=0, upper=1>	phi;	// interconnectivity	3
	strength			4
23				5
24	}			6
25				7
26	model {			8
27				9
28	real S[nloc, T];			10
29	real I[nloc, T];			11
30	real R[nloc, T];			12
31	real B[nloc, T];			13
32	real B0;			14
33				15
34	real BSI[nloc, T];			16
35	real rI[nloc, T];			17
36	int n;			18
37	real sph;			19
38	real ophi;			20
39	real nBIs;			21
40				22
41	B0 ← R0 * r / N;			23
42				24
43	for (loc in 1:nloc) {			25
44	S[loc, 1] ← N - Iinit[loc];			26
45	I[loc, 1] ← Iinit[loc];			27
46	R[loc, 1] ← 0.0;			28
47	B[loc, 1] ← B0;			29
48	}			30
49				31
50	for (t in 2:T) {			32
51	for (loc in 1:nloc) {			33
52	Bnoise[loc, t] ~ normal(0,berr);			34
53	B[loc, t] ← exp(log(B[loc, t-1]) + eta * (log(B0) -			35
54	log(B[loc, t-1])) + Bnoise[loc, t]);			36
55				37
56	n ← neinum[loc];			38
57	sph ← 1.0 - phi*(n/(n+1.0));			39
58	ophi ← phi/(n+1.0);			40
59				41
60	nBIs ← 0.0;			42
61	for (j in 1:n)			43
62	nBIs ← nBIs + B[neibmat[loc, j], t-1] * I[44
	neibmat[loc, j], t-1];			45
63				46
64	BSI[loc, t] ← S[loc, t-1]*(sph*B[loc, t-1]*I[loc, t-1]			47
	+ ophi*nBIs);			48
65	rI[loc, t] ← r*I[loc, t-1];			49
				50

```

1 66
2 67     S[loc, t] ← S[loc, t-1] + h*( - BSI[loc, t] );
3 68     I[loc, t] ← I[loc, t-1] + h*( BSI[loc, t] - rI[loc, t] );
4    ;
5 69     R[loc, t] ← R[loc, t-1] + h*( rI[loc, t] );
6 70
7 71     if (y[loc, t] > 0) {
8 72         y[loc, t] ~ normal( I[loc, t], sigma );
9 73     }
10 74
11 75 }
12 76 }
13 77
14 78 R0      ~ lognormal(1,1);
15 79 r       ~ lognormal(1,1);
16 80 sigma   ~ lognormal(1,1);
17 81 for (loc in 1:nloc) {
18 82     Iinit[loc] ~ normal(y[loc, 1], sigma);
19 83 }
20 84
21 85 }

```

23 F.3 IF2 Spatial SIR Code

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

```

25 1 /* Author: Dexter Barrows
26 2   Github: dbarrows.github.io
27 3
28 4   */
29 5
30 6 #include <stdio.h>
31 7 #include <math.h>
32 8 #include <sys/time.h>
33 9 #include <time.h>
34 10 #include <stdlib.h>
35 11 #include <string>
36 12 #include <cmath>
37 13 #include <cstdlib>
38 14 #include <fstream>
39 15
40 16 // #include "rand.h"
41 17 // #include "timer.h"
42 18
43 19 #define Treal      100          // time to simulate over
44 20 #define R0true     3.0          // infectiousness
45 21 #define rtrue      0.1          // recovery rate

```

```

22 #define Nreal      500.0      // population size      1
23 #define etatrue    0.5        // real drift attraction strength  2
24 #define berrtrue    0.5        // real beta drift noise          3
25 #define phitrue     0.5        // real connectivity strength     4
26 #define merr        10.0       // expected measurement error     5
27 #define I0          5.0        // Initial infected individuals    6
28                                     7
29 #define PSC         0.5        // perturbation scale factor for   8
    more sensitive parameters      9
30                                     10
31 #include <Rcpp.h>                11
32 using namespace Rcpp;           12
33                                 13
34 struct Particle {                14
35     double R0;                  15
36     double r;                   16
37     double sigma;               17
38     double eta;                 18
39     double berr;                19
40     double phi;                 20
41     double * S;                 21
42     double * I;                 22
43     double * R;                 23
44     double * B;                 24
45     double * Iinit;             25
46 };                              26
47                                 27
48                                 28
49 int timeval_subtract (double *result, struct timeval *x, struct  29
    timeval *y);                30
50 int check_double(double x, double y); 31
51 void initializeParticles(Particle ** particles, int NP, int nloc, 32
    int N);                    33
52 void exp_euler_SSIR(double h, double t0, double tn, int N, Particle 34
    * particle,                35
53     NumericVector neinum, NumericMatrix neibmat, int 36
    nloc) ;                    37
54 void copyParticle(Particle * dst, Particle * src, int nloc); 38
55 void perturbParticles(Particle * particles, int N, int NP, int nloc, 39
    int passnum, double coolrate); 40
56 double randu();                41
57 double randn();                42
58                                 43
59 // [[Rcpp::export]]            44
60 Rcpp::List if2_spa(NumericMatrix data, int T, int N, int NP, int 45
    nPasses, double coolrate, NumericVector neinum, NumericMatrix 46
    neibmat, int nloc) {        47
61                                 48
62     NumericMatrix paramdata(NP, 6); // for R0, r, sigma, eta, 49
    berr, phi                    50

```

```

1 63 NumericMatrix initInfec(nloc, NP); // for Iinit
2 64 NumericMatrix infecmeans(nloc, T); // mean infection counts for
3   each location
4 65 NumericMatrix finalstate(nloc, 4); // SIRB means for each
5   location
6 66
7 67 srand(time(NULL)); // Seed PRNG with system time
8 68
9 69 double w[NP]; // particle weights
10 70
11 71 // initialize particles
12 72 printf("Initializing particle states\n");
13 73 Particle * particles = NULL; // particle estimates for
14   current step
15 74 Particle * particles_old = NULL; // intermediate particle
16   states for resampling
17 75 initializeParticles(&particles, NP, nloc, N);
18 76 initializeParticles(&particles_old, NP, nloc, N);
19 77
20 78 /*
21 79 // copy particle test
22 80 copyParticle(&particles[0], &particles_old[0], nloc);
23 81
24 82 // perturb particle test
25 83 perturbParticles(particles, N, NP, nloc, 1, coolrate);
26 84
27 85 // evolution test
28 86 // reset particle system evolution states
29 87 for (int n = 0; n < NP; n++) {
30 88     for (int loc = 0; loc < nloc; loc++) {
31 89         particles[n].S[loc] = N - particles[n].Iinit[loc];
32 90         particles[n].I[loc] = particles[n].Iinit[loc];
33 91         particles[n].R[loc] = 0.0;
34 92         particles[n].B[loc] = (double) particles[n].R0 *
35   particles[n].r / N;
36 93     }
37 94 }
38 95 printf("Before S:%f | I:%f | R:%f\n", particles[0].S[0],
39   particles[0].I[0], particles[0].R[0]);
40 96 exp_euler_SSIR(1.0/7.0, 0.0, 1.0, N, &particles[0], neinum,
41   neibmat, nloc);
42 97 printf("After S:%f | I:%f | R:%f\n", particles[0].S[0],
43   particles[0].I[0], particles[0].R[0]);
44 98 */
45 99
46 100 // START PASSES THROUGH DATA
47 101
48 102 printf("Starting filter\n");
49 103 printf("-----\n");
50 104 printf("Pass\n");

```

105		1
106		2
107	for (int pass = 0; pass < nPasses; pass++) {	3
108		4
109	printf("...%d / %d\n", pass, nPasses);	5
110		6
111	// reset particle system evolution states	7
112	for (int n = 0; n < NP; n++) {	8
113	for (int loc = 0; loc < nloc; loc++) {	9
114	particles[n].S[loc] = N - particles[n].Iinit[loc];	10
115	particles[n].I[loc] = particles[n].Iinit[loc];	11
116	particles[n].R[loc] = 0.0;	12
117	particles[n].B[loc] = (double) particles[n].R0 * particles[n].r / N;	13 14
118	}	15
119	}	16
120		17
121	if (pass == (nPasses-1)) {	18
122	double means[nloc];	19
123	for (int loc = 0; loc < nloc; loc++) {	20
124	means[loc] = 0.0;	21
125	for (int n = 0; n < NP; n++) {	22
126	means[loc] += particles[n].I[loc] / NP;	23
127	}	24
128	infecmeans(loc, 0) = means[loc];	25
129	}	26
130	}	27
131		28
132	for (int t = 1; t < T; t++) {	29
133		30
134	// generate individual predictions and weight	31
135	for (int n = 0; n < NP; n++) {	32
136		33
137	exp_euler_SSIR(1.0/7.0, 0.0, 1.0, N, &particles[n], neinum, neibmat, nloc);	34 35
138		36
139	double merr_par = particles[n].sigma;	37
140		38
141	w[n] = 1.0;	39
142	for (int loc = 0; loc < nloc; loc++) {	40
143	double y_diff = data(loc, t) - particles[n].I[loc];	41 42
144	w[n] *= 1.0/(merr_par*sqrt(2.0*M_PI)) * exp(- y_diff*y_diff / (2.0*merr_par*merr_par));	43 44
145	}	45
146		46
147	}	47
148		48
149	// cumulative sum	49
150	for (int n = 1; n < NP; n++) {	50

```

1 151         w[n] += w[n-1];
2 152     }
3 153
4 154     // save particle states to resample from
5 155     for (int n = 0; n < NP; n++){
6 156         copyParticle(&particles_old[n], &particles[n], nloc)
7 157         ;
8 158     }
9 158
10 159     // resampling
11 160     for (int n = 0; n < NP; n++) {
12 161
13 162         double w_r = randu() * w[NP-1];
14 163         int i = 0;
15 164         while (w_r > w[i]) {
16 165             i++;
17 166         }
18 167
19 168         // i is now the index to copy state from
20 169         copyParticle(&particles[n], &particles_old[i], nloc)
21 170         ;
22 170
23 171     }
24 172
25 173     // between-iteration perturbations, not after last time
26 174     step
27 174     if (t < (T-1))
28 175         perturbParticles(particles, N, NP, nloc, pass,
29 176         coolrate);
30 176
31 177     if (pass == (nPasses-1)) {
32 178         double means[nloc];
33 179         for (int loc = 0; loc < nloc; loc++) {
34 180             means[loc] = 0.0;
35 181             for (int n = 0; n < NP; n++) {
36 182                 means[loc] += particles[n].I[loc] / NP;
37 183             }
38 184             infecmeans(loc, t) = means[loc];
39 185         }
40 186     }
41 187
42 188 }
43 189
44 190 // between-pass perturbations, not after last pass
45 191 if (pass < (nPasses + 1))
46 192     perturbParticles(particles, N, NP, nloc, pass, coolrate)
47 193     ;
48 193
49 194 }
50 195

```



```

196 // pack parameter data (minus initial conditions) | 1
197 for (int n = 0; n < NP; n++) { | 2
198     paramdata(n, 0) = particles[n].R0; | 3
199     paramdata(n, 1) = particles[n].r; | 4
200     paramdata(n, 2) = particles[n].sigma; | 5
201     paramdata(n, 3) = particles[n].eta; | 6
202     paramdata(n, 4) = particles[n].berr; | 7
203     paramdata(n, 5) = particles[n].phi; | 8
204 } | 9
205 | 10
206 // Pack initial condition data | 11
207 for (int n = 0; n < NP; n++) { | 12
208     for (int loc = 0; loc < nloc; loc++) { | 13
209         initInfec(loc, n) = particles[n].Iinit[loc]; | 14
210     } | 15
211 } | 16
212 | 17
213 // Pack final state means data | 18
214 double Smeans[nloc], Imeans[nloc], Rmeans[nloc], Bmeans[nloc]; | 19
215 for (int loc = 0; loc < nloc; loc++) { | 20
216     Smeans[loc] = 0.0; | 21
217     Imeans[loc] = 0.0; | 22
218     Rmeans[loc] = 0.0; | 23
219     Bmeans[loc] = 0.0; | 24
220     for (int n = 0; n < NP; n++) { | 25
221         Smeans[loc] += particles[n].S[loc] / NP; | 26
222         Imeans[loc] += particles[n].I[loc] / NP; | 27
223         Rmeans[loc] += particles[n].R[loc] / NP; | 28
224         Bmeans[loc] += particles[n].B[loc] / NP; | 29
225     } | 30
226     finalstate(loc, 0) = Smeans[loc]; | 31
227     finalstate(loc, 1) = Imeans[loc]; | 32
228     finalstate(loc, 2) = Rmeans[loc]; | 33
229     finalstate(loc, 3) = Bmeans[loc]; | 34
230 } | 35
231 | 36
232 | 37
233 return Rcpp::List::create( Rcpp::Named("paramdata") = paramdata | 38
234     , | 39
235     Rcpp::Named("initInfec") = initInfec | 40
236     , | 41
237     Rcpp::Named("infecmeans") = | 42
238     infecmeans, | 43
239     Rcpp::Named("finalstate") = | 44
240     finalstate); | 45
241 } | 46
| 47
| 48
| 49
| 50

```

```

1 242
2 243 /* Use the Explicit Euler integration scheme to integrate SIR model
3      forward in time
4 244     double h      - time step size
5 245     double t0     - start time
6 246     double tn     - stop time
7 247     double * y    - current system state; a three-component vector
8                      representing [S I R], susceptible-infected-recovered
9 248
10 249     */
11 250 void exp_euler_SSIR(double h, double t0, double tn, int N, Particle
12     * particle,
13     NumericVector neinum, NumericMatrix neibmat, int
14     nloc) {
15 252
16 253     int num_steps = floor( (tn-t0) / h );
17 254
18 255     double * S = particle->S;
19 256     double * I = particle->I;
20 257     double * R = particle->R;
21 258     double * B = particle->B;
22 259
23 260     // create last state vectors
24 261     double S_last[nloc];
25 262     double I_last[nloc];
26 263     double R_last[nloc];
27 264     double B_last[nloc];
28 265
29 266     double R0    = particle->R0;
30 267     double r     = particle->r;
31 268     double B0    = R0 * r / N;
32 269     double eta   = particle->eta;
33 270     double berr  = particle->berr;
34 271     double phi   = particle->phi;
35 272
36 273     //printf("sphi \t\t| ophi \t\t| BSI \t\t| rI \t\t| dS \t\t| dI \
37         \t\t| dR \t\t| S\t\t| I \t\t| R |\n");
38 274
39 275     for(int t = 0; t < num_steps; t++) {
40 276
41 277         for (int loc = 0; loc < nloc; loc++) {
42 278             S_last[loc] = S[loc];
43 279             I_last[loc] = I[loc];
44 280             R_last[loc] = R[loc];
45 281             B_last[loc] = B[loc];
46 282         }
47 283
48 284         for (int loc = 0; loc < nloc; loc++) {
49 285
50 286             B[loc] = exp( log(B_last[loc]) + eta*(log(B0) - log(

```

```

287         B_last[loc])) + berr*randn() );
288
289     int n = neinum[loc];
290     double sphi = 1.0 - phi*( (double) n/(n+1.0) );
291     double ophi = phi/(n+1.0);
292
293     double nBIsu = 0.0;
294     for (int j = 0; j < n; j++)
295         nBIsu += B_last[(int) neibmat(loc, j) - 1] * I_last
296             [(int) neibmat(loc, j) - 1];
297
298     double BSI = S_last[loc]*( sphi*B_last[loc]*I_last[loc]
299         + ophi*nBIsu );
300     double rI = r*I_last[loc];
301
302     // get derivatives
303     double dS = - BSI;
304     double dI = BSI - rI;
305     double dR = rI;
306
307     // step forward by h
308     S[loc] += h*dS;
309     I[loc] += h*dI;
310     R[loc] += h*dR;
311
312     //if (loc == 1)
313     // printf("%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t%f\t\n", sphi, ophi, BSI, rI, dS, dI, dR, S[1], I
314     // [1], R[1]);
315
316 }
317
318 }
319
320 /*particle->S = S;
321 particle->I = I;
322 particle->R = R;
323 particle->B = B;*/
324
325 }
326
327 /* Initializes particles
328 */
329 void initializeParticles(Particle ** particles, int NP, int nloc,
330 int N) {
331
332     // allocate space for doubles
333     *particles = (Particle*) malloc (NP*sizeof(Particle));
334
335     // allocate space for arrays inside particles

```

```

1 331     for (int n = 0; n < NP; n++) {
2 332         (*particles)[n].S = (double*) malloc(nloc*sizeof(double));
3 333         (*particles)[n].I = (double*) malloc(nloc*sizeof(double));
4 334         (*particles)[n].R = (double*) malloc(nloc*sizeof(double));
5 335         (*particles)[n].B = (double*) malloc(nloc*sizeof(double));
6 336         (*particles)[n].Iinit = (double*) malloc(nloc*sizeof(double)
7         );
8 337     }
9 338
10 339     // initialize all all parameters
11 340     for (int n = 0; n < NP; n++) {
12 341
13 342         double R0can, rcan, sigmacan, Iinitcan, etacan, berrcan,
14         phican;
15 343
16 344         do {
17 345             R0can = R0true + R0true*randn();
18 346         } while (R0can < 0);
19 347         (*particles)[n].R0 = R0can;
20 348
21 349         do {
22 350             rcan = rtrue + rtrue*randn();
23 351         } while (rcan < 0);
24 352         (*particles)[n].r = rcan;
25 353
26 354         for (int loc = 0; loc < nloc; loc++)
27 355             (*particles)[n].B[loc] = (double) R0can * rcan / N;
28 356
29 357         do {
30 358             sigmacan = merr + merr*randn();
31 359         } while (sigmacan < 0);
32 360         (*particles)[n].sigma = sigmacan;
33 361
34 362         do {
35 363             etacan = etatrue + PSC*etatrue*randn();
36 364         } while (etacan < 0 || etacan > 1);
37 365         (*particles)[n].eta = etacan;
38 366
39 367         do {
40 368             berrcan = berrtrue + PSC*berrtrue*randn();
41 369         } while (berrcan < 0);
42 370         (*particles)[n].berr = berrcan;
43 371
44 372         do {
45 373             phican = phitrue + PSC*phitrue*randn();
46 374         } while (phican <= 0 || phican >= 1);
47 375         (*particles)[n].phi = phican;
48 376
49 377         for (int loc = 0; loc < nloc; loc++) {
50 378             do {

```

```

379         Iinitcan = I0 + I0*randn();
380     } while (Iinitcan < 0 || N < Iinitcan);
381     (*particles)[n].Iinit[loc] = Iinitcan;
382 }
383
384 }
385
386 }
387
388 /* Particle pertubation function to be run between iterations and
389    passes
390    */
391 void perturbParticles(Particle * particles, int N, int NP, int nloc,
392     int passnum, double coolrate) {
393     //double coolcoef = exp( - (double) passnum / coolrate );
394     double coolcoef = pow(coolrate, passnum);
395
396     double spreadR0      = coolcoef * R0true / 10.0;
397     double spreadr       = coolcoef * rtrue / 10.0;
398     double spreadsigma   = coolcoef * merr / 10.0;
399     double spreadIinit   = coolcoef * I0 / 10.0;
400     double spreadeta     = coolcoef * etatrue / 10.0;
401     double spreadberr    = coolcoef * berrtrue / 10.0;
402     double spreadphi     = coolcoef * phitrue / 10.0;
403
404     double R0can, rcan, sigmacan, Iinitcan, etacan, berrcan, phican;
405
406     for (int n = 0; n < NP; n++) {
407
408         do {
409             R0can = particles[n].R0 + spreadR0*randn();
410         } while (R0can < 0);
411         particles[n].R0 = R0can;
412
413         do {
414             rcan = particles[n].r + spreadr*randn();
415         } while (rcan < 0);
416         particles[n].r = rcan;
417
418         do {
419             sigmacan = particles[n].sigma + spreadsigma*randn();
420         } while (sigmacan < 0);
421         particles[n].sigma = sigmacan;
422
423         do {
424             etacan = particles[n].eta + PSC*spreadeta*randn();
425         } while (etacan < 0 || etacan > 1);
426         particles[n].eta = etacan;

```

```

1 427
2 428     do {
3 429         berrcan = particles[n].berr + PSC*spreadberr*randn();
4 430     } while (berrcan < 0);
5 431     particles[n].berr = berrcan;
6 432
7 433     do {
8 434         phican = particles[n].phi + PSC*spreadphi*randn();
9 435     } while (phican <= 0 || phican >= 1);
10 436     particles[n].phi = phican;
11 437
12 438     for (int loc = 0; loc < nloc; loc++) {
13 439         do {
14 440             Iinitcan = particles[n].Iinit[loc] + spreadIinit*
15             randn();
16 441         } while (Iinitcan < 0 || Iinitcan > 500);
17 442         particles[n].Iinit[loc] = Iinitcan;
18 443     }
19 444 }
20 445
21 446 }
22 447
23 448 /* Convenience function for particle resampling process
24 449 */
25 450 void copyParticle(Particle * dst, Particle * src, int nloc) {
26 451
27 452     dst->R0      = src->R0;
28 453     dst->r        = src->r;
29 454     dst->sigma    = src->sigma;
30 455     dst->eta      = src->eta;
31 456     dst->berr     = src->berr;
32 457     dst->phi      = src->phi;
33 458
34 459     for (int n = 0; n < nloc; n++) {
35 460         dst->S[n]      = src->S[n];
36 461         dst->I[n]      = src->I[n];
37 462         dst->R[n]      = src->R[n];
38 463         dst->B[n]      = src->B[n];
39 464         dst->Iinit[n]  = src->Iinit[n];
40 465     }
41 466
42 467 }
43 468
44 469
45 470
46 471 double randu() {
47 472
48 473     return (double) rand() / (double) RAND_MAX;
49 474
50 475 }

```

```

476 |
477 | /*
478 | void getStateMeans(State * state, Particle* particles, int NP) {
479 |
480 |     double Smean = 0, Imean = 0, Rmean = 0;
481 |
482 |     for (int n = 0; n < NP; n++) {
483 |         Smean += particles[n].S;
484 |         Imean += particles[n].I;
485 |         Rmean += particles[n].R;
486 |     }
487 |
488 |     state->S = (double) Smean / NP;
489 |     state->I = (double) Imean / NP;
490 |     state->R = (double) Rmean / NP;
491 |
492 | }
493 | */
494 |
495 | /* Return a normally distributed random number with mean 0 and
496 |    standard deviation 1
497 |    Uses the polar form of the Box-Muller transformation
498 |    From http://www.design.caltech.edu/erik/Misc/Gaussian.html
499 |    */
500 | double randn() {
501 |     double x1, x2, w, y1;
502 |
503 |     do {
504 |         x1 = 2.0 * randu() - 1.0;
505 |         x2 = 2.0 * randu() - 1.0;
506 |         w = x1 * x1 + x2 * x2;
507 |     } while ( w >= 1.0 );
508 |
509 |     w = sqrt( (-2.0 * log( w ) ) / w );
510 |     y1 = x1 * w;
511 |
512 |     return y1;
513 |
514 | }

```

F.4 CUDA IF2 Spatial Fitting Code

Below is the nascent CUDA code that will be expanded upon in future work.

```

1 | /* Author: Dexter Barrows
2 |    Github: dbarrows.github.io

```

```

1  3      */
2  4
3  5  /*  Runs a particle filter on synthetic noisy data and attempts to
4  6      reconstruct underlying true state at each time step. Note that
5  7      this program uses gnuplot to plot the data, so an x11
6  8      environment must be present. Also the multiplier of 1024 in the
7  9      definition of NP below should be set to a multiple of the number
8  10     of multiprocessors of your GPU for optimal results.
9  11
10 12     Also, the accompanying "pf.plg" file contains the instructions
11 13     gnuplot will use. It must be present in the same directory as
12 14     the executable generated by compiling this file.
13 15
14 16     Compile with:
15 17
16 18     nvcc -arch=sm_20 -O2 pf_cuda.cu timer.cpp rand.cpp -o pf_cuda.x
17 19
18 20     */
19 21
20 22 #include <cuda.h>
21 23 #include <iostream>
22 24 #include <fstream>
23 25 #include <curand.h>
24 26 #include <curand_kernel.h>
25 27 #include <string>
26 28 #include <sstream>
27 29 #include <cmath>
28 30
29 31 #include "timer.h"
30 32 #include "rand.h"
31 33 #include "readdata.h"
32 34
33 35 #define NP          (2*2500)      // number of particles
34 36 #define N           500.0        // population size
35 37 #define R0true      3.0          // infectiousness
36 38 #define rtrue       0.1          // recovery rate
37 39 #define etatrue     0.5          // real drift attraction strength
38 40 #define berrtrue    0.5          // real beta drift noise
39 41 #define phitrue     0.5          // real connectivity strength
40 42 #define merr        10.0         // expected measurement error
41 43 #define I0          5.0          // Initial infected individuals
42 44 #define PSC         0.5          // sensitive parameter perturbation
43     scaling
44 45 #define NLOC        10
45 46
46 47 #define PI          3.141592654f
47 48
48 49 // Wrapper for CUDA calls, from CUDA API
49 50 // Modified to also print the error code and string
50 51 # define CUDA_CALL(x) do { if ((x) != cudaSuccess ) {

```


127

```

1  93 // initialize PRNG state
2  94 curandState state;
3  95 curand_init(id, 0, 0, &state);
4  96
5  97 // allocate space for arrays inside particle
6  98 //particles[id].S = (float*) malloc(nloc*sizeof(float));
7  99 //particles[id].I = (float*) malloc(nloc*sizeof(float));
8 100 //particles[id].R = (float*) malloc(nloc*sizeof(float));
9 101 //particles[id].B = (float*) malloc(nloc*sizeof(float));
10 102 //particles[id].Iinit = (float*) malloc(nloc*sizeof(float));
11 103
12 104 // initialize all parameters
13 105
14 106 float R0can, rcan, sigmacan, Iinitcan, etacan, berrcan,
15      phican;
16 107
17 108 do {
18 109     R0can = R0true + R0true*curand_normal(&state);
19 110 } while (R0can < 0);
20 111 particles[id].R0 = R0can;
21 112
22 113 do {
23 114     rcan = rtrue + rtrue*curand_normal(&state);
24 115 } while (rcan < 0);
25 116 particles[id].r = rcan;
26 117
27 118 for (int loc = 0; loc < nloc; loc++)
28 119     particles[id].B[loc] = (float) R0can * rcan / N;
29 120
30 121 do {
31 122     sigmacan = merr + merr*curand_normal(&state);
32 123 } while (sigmacan < 0);
33 124 particles[id].sigma = sigmacan;
34 125
35 126 do {
36 127     etacan = etatrue + PSC*etatrue*curand_normal(&state);
37 128 } while (etacan < 0 || etacan > 1);
38 129 particles[id].eta = etacan;
39 130
40 131 do {
41 132     berrcan = berrtrue + PSC*berrtrue*curand_normal(&state);
42 133 } while (berrcan < 0);
43 134 particles[id].berr = berrcan;
44 135
45 136 do {
46 137     phican = phitrue + PSC*phitrue*curand_normal(&state);
47 138 } while (phican <= 0 || phican >= 1);
48 139 particles[id].phi = phican;
49 140
50 141 for (int loc = 0; loc < nloc; loc++) {

```

```

142         do {
143             Iinitcan = I0 + I0*curand_normal(&state);
144         } while (Iinitcan < 0 || N < Iinitcan);
145         particles[id].Iinit[loc] = Iinitcan;
146     }
147
148     particles[id].randState = state;
149
150 }
151
152 }
153
154 __global__ void resetStates (Particle * particles, int nloc) {
155
156     int id = blockIdx.x*blockDim.x + threadIdx.x; // global thread
157     ID
158
159     for (int loc = 0; loc < nloc; loc++) {
160         particles[id].S[loc] = N - particles[id].Iinit[loc];
161         particles[id].I[loc] = particles[id].Iinit[loc];
162         particles[id].R[loc] = 0.0;
163     }
164 }
165
166 __global__ void clobberParams (Particle * particles, int nloc) {
167
168     int id = blockIdx.x*blockDim.x + threadIdx.x; // global thread
169     ID
170
171     particles[id].R0 = R0true;
172     particles[id].r = rtrue;
173     particles[id].sigma = merr;
174     particles[id].eta = etatrue;
175     particles[id].berr = berrtrue;
176     particles[id].phi = phitrue;
177
178     for (int loc = 0; loc < nloc; loc++) {
179         particles[id].Iinit[loc] = I0;
180     }
181 }
182
183
184
185 /* Project particles forward, perturb, and save weight based on
186    data
187    int t - time step number (1,...,T)
188    */
189 __global__ void project (Particle * particles, int * neinum, int *

```

```

1      neibmat, int nloc) {
2 189
3 190      int id = blockIdx.x*blockDim.x + threadIdx.x;    // global id
4 191
5 192      if (id < NP) {
6 193          // project forward
7 194          exp_euler_SSIR(1.0/7.0, 0.0, 1.0, &particles[id], neinum,
8          neibmat, nloc);
9 195      }
10 196
11 197 }
12 198
13 199 __global__ void weight(float * data, Particle * particles, double *
14      w, int t, int T, int nloc) {
15 200
16 201      int id = blockIdx.x*blockDim.x + threadIdx.x;    // global id
17 202
18 203      if (id < NP) {
19 204
20 205          float merr_par = particles[id].sigma;
21 206
22 207          // Get weight and save
23 208          double w_local = 1.0;
24 209          for (int loc = 0; loc < nloc; loc++) {
25 210              float y_diff = data[loc*T + t] - particles[id].I[loc];
26 211              w_local *= 1.0/(merr_par*sqrt(2.0*PI)) * exp( - y_diff*
27              y_diff / (2.0*merr_par*merr_par) );
28 212          }
29 213
30 214          w[id] = w_local;
31 215
32 216      }
33 217
34 218 }
35 219
36 220 __global__ void stashParticles (Particle * particles, Particle *
37      particles_old, int nloc) {
38 221
39 222      int id = blockIdx.x*blockDim.x + threadIdx.x;    // global id
40 223
41 224      if (id < NP) {
42 225          // COPY PARTICLE
43 226          copyParticle(&particles_old[id], &particles[id], nloc);
44 227      }
45 228
46 229 }
47 230
48 231
49 232 /* The 0th thread will perform cumulative sum on the weights.
50 233    There may be a faster way to do this, will investigate.

```

```

234     */
235 __global__ void cumsumWeights (double * w) {
236
237     int id  = blockIdx.x*blockDim.x + threadIdx.x;  // global thread
238     ID
239
240     // compute cumulative weights
241     if (id == 0) {
242         for (int i = 1; i < NP; i++)
243             w[i] += w[i-1];
244     }
245 }
246
247
248 /* Resample from all particle states within cell
249     */
250 __global__ void resample (Particle * particles, Particle *
251     particles_old, double * w, int nloc) {
252
253     int id  = blockIdx.x*blockDim.x + threadIdx.x;
254
255     if (id < NP) {
256
257         // resampling proportional to weights
258         double w_r = curand_uniform(&particles[id].randState) * w[NP
259             -1];
260         int i = 0;
261         while (w_r > w[i]) {
262             i++;
263         }
264         // i is now the index of the particle to copy from
265         copyParticle(&particles[id], &particles_old[i], nloc);
266     }
267 }
268 }
269
270 // launch this with probably just nloc threads... block structure/
271 // size probably not important
272 __global__ void reduceStates (Particle * particles, float *
273     countmeans, int t, int T, int nloc) {
274
275     int id  = blockIdx.x*blockDim.x + threadIdx.x;
276
277     if (id < nloc) {
278         int loc = id;

```

```

1 279         double countmean_local = 0.0;
2 280         for (int n = 0; n < NP; n++) {
3 281             countmean_local += particles[n].I[loc] / NP;
4 282         }
5 283
6 284         countmeans[loc*T + t] = (float) countmean_local;
7 285
8 286     }
9 287
10 288 }
11 289
12 290 __global__ void perturbParticles(Particle * particles, int nloc, int
13     passnum, double coolrate) {
14 291
15 292     //double coolcoef = exp( - (double) passnum / coolrate );
16 293     double coolcoef = pow(coolrate, passnum);
17 294
18 295     double spreadR0      = coolcoef * R0true / 10.0;
19 296     double spreadr       = coolcoef * rtrue / 10.0;
20 297     double spreadsigma   = coolcoef * merr / 10.0;
21 298     double spreadIinit   = coolcoef * I0 / 10.0;
22 299     double spreadeta     = coolcoef * etatrue / 10.0;
23 300     double spreadberr    = coolcoef * berrtrue / 10.0;
24 301     double spreadphi     = coolcoef * phitrue / 10.0;
25 302
26 303     double R0can, rcan, sigmacan, Iinitcan, etacan, berrcan, phican;
27 304
28 305     int id = blockIdx.x*blockDim.x + threadIdx.x;
29 306
30 307     if (id < NP) {
31 308
32 309         do {
33 310             R0can = particles[id].R0 + spreadR0*curand_normal(&
34             particles[id].randState);
35 311         } while (R0can < 0);
36 312         particles[id].R0 = R0can;
37 313
38 314         do {
39 315             rcan = particles[id].r + spreadr*curand_normal(&
40             particles[id].randState);
41 316         } while (rcan < 0);
42 317         particles[id].r = rcan;
43 318
44 319         do {
45 320             sigmacan = particles[id].sigma + spreadsigma*
46             curand_normal(&particles[id].randState);
47 321         } while (sigmacan < 0);
48 322         particles[id].sigma = sigmacan;
49 323
50 324         do {

```

```

325         etacan = particles[id].eta + PSC*spreadeta*curand_normal 1
326             (&particles[id].randState); 2
327     } while (etacan < 0 || etacan > 1); 3
328     particles[id].eta = etacan; 4
329 328 5
329     do { 6
330         berrcan = particles[id].berr + PSC*spreadberr* 7
331             curand_normal(&particles[id].randState); 8
332     } while (berrcan < 0); 9
333     particles[id].berr = berrcan; 10
334 333 11
334     do { 12
335         phican = particles[id].phi + PSC*spreadphi*curand_normal 13
336             (&particles[id].randState); 14
337     } while (phican <= 0 || phican >= 1); 15
338     particles[id].phi = phican; 16
339 338 17
339     for (int loc = 0; loc < nloc; loc++) { 18
340         do { 19
341             Iinitcan = particles[id].Iinit[loc] + spreadIinit* 20
342                 curand_normal(&particles[id].randState); 21
343         } while (Iinitcan < 0 || Iinitcan > 500); 22
344         particles[id].Iinit[loc] = Iinitcan; 23
345     } 24
346 345 25
346     } 26
347 346 27
348 } 28
349 347 29
350 348 30
351 int main (int argc, char *argv[]) { 31
352 350 32
353 351 33
354     int T, nloc; 34
355 352 35
356     double restime; 36
357     struct timeval tdr0, tdr1, tdrMaster; 37
358 353 38
359     gettimeofday (&tdr0, NULL); 39
360 354 40
361 355 41
362     // Parse arguments 42
363     ***** 43
364 356 44
365     if (argc < 4) { 45
366         std::cout << "Not enough arguments" << std::endl; 46
367         return 0; 47
368     } 48
369 357 49
370     std::string arg1(argv[1]); // infection counts 50

```

```

1 370     std::string arg2(argv[2]); // neighbour counts
2 371     std::string arg3(argv[3]); // neighbour indices
3 372
4 373     std::cout << "Arguments:" << std::endl;
5 374     std::cout << "Infection data:      " << arg1 << std::endl;
6 375     std::cout << "Neighbour counts:    " << arg2 << std::endl;
7 376     std::cout << "Neighbour indices:  " << arg3 << std::endl;
8 377
9 378     //
10      *****
11
12 379
13 380
14 381     // Read count data
15      *****
16 382
17 383     std::cout << "Getting count data" << std::endl;
18 384     float * data = getDataFloat(arg1, &T, &nloc);
19 385     size_t datasize = nloc*T*sizeof(float);
20 386
21 387     //
22      *****
23
24 388
25 389     // Read neinum matrix data
26      *****
27 390
28 391     std::cout << "Getting neighbour count data" << std::endl;
29 392     int * neinum = getDataInt(arg2, NULL, NULL);
30 393     size_t neinumsize = nloc * sizeof(int);
31 394
32 395     //
33      *****
34
35 396
36 397     // Read neibmat matrix data
37      *****
38 398
39 399     std::cout << "Getting neighbour count data" << std::endl;
40 400     int * neibmat = getDataInt(arg3, NULL, NULL);
41 401     size_t neibmatsize = nloc * nloc * sizeof(int);
42 402
43 403     //
44      *****
45
46 404
47 405
48 406     gettimeofday (&tdr1, NULL);
49 407     timeval_subtract (&restime, &tdr1, &tdr0);
50 408

```



```

409     std::cout << "\t" << getHRtime(restime) << std::endl;
410
411     //
412     *****
413     // CUDA data
414     *****
415     std::cout << "Allocating device storage" << std::endl;
416
417     gettimeofday (&tdr0, NULL);
418
419     float      * d_data;           // device copy of data
420     Particle    * particles;        // particles
421     Particle    * particles_old;    // intermediate particle states
422     double      * w;               // weights
423     int          * d_neinum;        // device copy of adjacency
424     matrix      * d_neibmat;        // device copy of neighbour
425     counts matrix
426     float      * countmeans;        // host copy of reduced
427     infection count means from last pass
428     float      * d_countmeans;      // device copy of reduced
429     infection count means from last pass
430
431     CUDA_CALL( cudaMalloc( (void**) &d_data           , datasize )
432                );
433     CUDA_CALL( cudaMalloc( (void**) &particles        , NP*sizeof(
434         Particle)) );
435     CUDA_CALL( cudaMalloc( (void**) &particles_old    , NP*sizeof(
436         Particle)) );
437     CUDA_CALL( cudaMalloc( (void**) &w               , NP*sizeof(
438         double)) );
439     CUDA_CALL( cudaMalloc( (void**) &d_neinum         , neinumsize)
440                );
441     CUDA_CALL( cudaMalloc( (void**) &d_neibmat        , neibmatsize)
442                );
443     CUDA_CALL( cudaMalloc( (void**) &d_countmeans     , nloc*T*sizeof(
444         float)) );
445
446     gettimeofday (&tdr1, NULL);
447     timeval_subtract (&restime, &tdr1, &tdr0);
448
449     std::cout << "\t" << getHRtime(restime) << std::endl;
450
451     size_t avail, total;
452     cudaMemGetInfo( &avail, &total );
453     size_t used = total - avail;
454 
```

```

1 445
2 446     std::cout << "\t[" << getHRmemsize(used) << "]" used of [" <<
3         getHRmemsize(total) << "]" <<std::endl;
4 447
5 448     std::cout << "Copying data to device" << std::endl;
6 449
7 450     gettimeofday (&tdr0, NULL);
8 451
9 452     CUDA_CALL( cudaMemcpy(d_data      , data      , datasize      ,
10         cudaMemcpyHostToDevice) );
11 453     CUDA_CALL( cudaMemcpy(d_neinum    , neinum    , neinumsize    ,
12         cudaMemcpyHostToDevice) );
13 454     CUDA_CALL( cudaMemcpy(d_neibmat   , neibmat   , neibmatsize   ,
14         cudaMemcpyHostToDevice) );
15 455
16 456     gettimeofday (&tdr1, NULL);
17 457     timeval_subtract (&restime, &tdr1, &tdr0);
18 458
19 459     std::cout << "\t" << getHRtime(restime) << std::endl;
20 460
21 461     //
22         *****
23
24 462
25 463
26 464
27 465     // Initialize particles
28         *****
29 466
30 467     std::cout << "Initializing particles" << std::endl;
31 468
32 469     gettimeofday (&tdr0, NULL);
33 470
34 471     int nThreads      = 32;
35 472     int nBlocks       = ceil( (float) NP / nThreads);
36 473
37 474     initializeParticles <<< nBlocks, nThreads >>> (particles, nloc);
38 475     CUDA_CALL( cudaGetLastError() );
39 476     CUDA_CALL( cudaDeviceSynchronize() );
40 477
41 478     initializeParticles <<< nBlocks, nThreads >>> (particles_old,
42         nloc);
43 479     CUDA_CALL( cudaGetLastError() );
44 480     CUDA_CALL( cudaDeviceSynchronize() );
45 481
46 482     gettimeofday (&tdr1, NULL);
47 483     timeval_subtract (&restime, &tdr1, &tdr0);
48 484
49 485     std::cout << "\t" << getHRtime(restime) << std::endl;
50 486

```

```

487     cudaMemGetInfo( &avail, &total );
488     used = total - avail;
489     std::cout << "\t[" << getHRmemsize(used) << "]" used of [" <<
        getHRmemsize(total) << "]" <<std::endl;
490
491     //
        *****
492
493     // Starting filtering
        *****
494
495     for (int pass = 0; pass < 50; pass++) {
496
497         std::cout << "pass = " << pass << std::endl;
498
499         // ** TEMP **
500         //clobberParams <<< nBlocks, nThreads >>> (particles, nloc);
501         // ** TEMP **
502
503         nThreads      = 32;
504         nBlocks       = ceil( (float) NP / nThreads);
505
506         resetStates <<< nBlocks, nThreads >>> (particles, nloc);
507         CUDA_CALL( cudaGetLastError() );
508         CUDA_CALL( cudaDeviceSynchronize() );
509
510         std::cout << "Filtering over [1," << Tlim << "]"<< std::endl
            ;
511
512         gettimeofday ( &tdrMaster, NULL);
513
514         gettimeofday ( &tdr0, NULL);
515
516         nThreads = 1;
517         nBlocks  = 10;
518
519         if (pass == 49) {
520             reduceStates <<< nBlocks, nThreads >>> (particles,
                d_countmeans, 0, T, nloc);
521             CUDA_CALL( cudaGetLastError() );
522             CUDA_CALL( cudaDeviceSynchronize() );
523         }
524
525         gettimeofday ( &tdr1, NULL);
526         timeval_subtract (&restime, &tdr1, &tdr0);
527         std::cout << "Reduction          " << getHRtime(restime) <<
            std::endl;
528
529         int Tlim = T;

```

```

1 530
2 531     for (int t = 1; t < Tlim; t++) {
3 532
4 533         // Projection
5         *****
6 534
7 535         nThreads      = 32;
8 536         nBlocks       = ceil( (float) NP / nThreads);
9 537
10 538         //if (t == 1)
11 539         //  gettimeofday (&tdr0, NULL);
12 540
13 541         project <<< nBlocks, nThreads >>> (particles, d_neinum,
14         d_neibmat, nloc);
15 542         CUDA_CALL( cudaGetLastError() );
16 543         CUDA_CALL( cudaDeviceSynchronize() );
17 544
18 545         //if (t == 1) {
19 546         //  gettimeofday (&tdr1, NULL);
20 547         //  timeval_subtract (&restime, &tdr1, &tdr0);
21 548         //  std::cout << "\tProjection " << getHRtime(restime)
22         << std::endl;
23 549         //}
24 550
25 551         // Weighting
26         *****
27 552
28 553         nThreads      = 32;
29 554         nBlocks       = ceil( (float) NP / nThreads);
30 555
31 556         weight <<< nBlocks, nThreads >>>(d_data, particles, w, t
32         , T, nloc);
33 557         CUDA_CALL( cudaGetLastError() );
34 558         CUDA_CALL( cudaDeviceSynchronize() );
35 559
36 560         // Cumulative sum
37         *****
38 561
39 562         nThreads      = 1;
40 563         nBlocks       = 1;
41 564
42 565         if (t == 1)
43 566             gettimeofday (&tdr0, NULL);
44 567
45 568         cumsumWeights <<< nBlocks, nThreads >>> (w);
46 569         CUDA_CALL( cudaGetLastError() );
47 570         CUDA_CALL( cudaDeviceSynchronize() );
48 571
49 572         if (t == 1) {
50 573             gettimeofday (&tdr1, NULL);

```

```

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

```

```

1 617
2 618         reduceStates <<< nBlocks, nThreads >>> (particles,
3         d_countmeans, t, T, nloc);
4 619     CUDA_CALL( cudaGetLastError() );
5 620     CUDA_CALL( cudaDeviceSynchronize() );
6 621
7 622     if (t == 1) {
8 623         gettimeofday (&tdr1, NULL);
9 624         timeval_subtract (&restime, &tdr1, &tdr0);
10 625         std::cout << "Reduction          " << getHRtime(
11         restime) << std::endl;
12 626     }
13 627
14 628 }
15 629
16 630 // Perturb particles
17         *****
18 631
19 632     nThreads      = 32;
20 633     nBlocks       = ceil( (float) NP/ nThreads);
21 634
22 635     perturbParticles <<< nBlocks, nThreads >>> (particles,
23     nloc, pass, 0.975);
24 636     CUDA_CALL( cudaGetLastError() );
25 637     CUDA_CALL( cudaDeviceSynchronize() );
26 638
27 639     //}
28 640     /*
29 641     nThreads      = RB_DIM;
30 642     nBlocks       = nCells;
31 643
32 644
33 645
34 646     reduce <<< nBlocks, nThreads >>> (d_E, t, particles,
35     Beta_last, nCells);
36 647     CUDA_CALL( cudaGetLastError() );
37 648     CUDA_CALL( cudaDeviceSynchronize() );
38 649
39 650     if (t == 1) {
40 651         gettimeofday (&tdr1, NULL);
41 652         timeval_subtract (&restime, &tdr1, &tdr0);
42 653         std::cout << "Reduction          " << getHRtime(
43         restime) << std::endl;
44 654     }
45 655     */
46 656
47 657 } // end time
48 658
49 659 } // end pass
50 660

```

661		1
662	std::cout.precision(10);	2
663		3
664	countmeans = (float*) malloc (nloc*T*sizeof(float));	4
665	cudaMemcpy(countmeans, d_countmeans, nloc*T*sizeof(float),	5
	cudaMemcpyDeviceToHost);	6
666		7
667	std::string filename = "cuIF2states.dat";	8
668		9
669	std::cout << "Writing results to file '" << filename << "' ..."	10
	<< std::endl;	11
670		12
671	std::ofstream outfile;	13
672	outfile.open(filename.c_str());	14
673		15
674	for(int loc = 0; loc < nloc; loc++) {	16
675	for (int t = 0; t < T; t++) {	17
676	outfile << countmeans[loc*T + t] << " ";	18
677	}	19
678	outfile << std::endl;	20
679	}	21
680		22
681	/*	23
682	double * h_w = (double*) malloc (NP*sizeof(double));	24
683	cudaMemcpy(h_w, w, NP*sizeof(double), cudaMemcpyDeviceToHost);	25
684		26
685	for (int n = 0; n < NP; n++) {	27
686	std::cout << h_w[n] << " ";	28
687	}	29
688	*/	30
689		31
690	/*	32
691	for (int i = 0; i < nCells; i++) {	33
692	outfile << trueCounts[t*nCells + i];	34
693	if (i % dim == 0)	35
694	outfile << std::endl;	36
695	else	37
696	outfile << " ";	38
697	}	39
698	*/	40
699		41
700	outfile.close();	42
701		43
702	gettimeofday (&tdr1, NULL);	44
703	timeval_subtract (&restime, &tdr1, &tdrMaster);	45
704	std::cout << "Total PF time (excluding setup) " << getHRtime(46
	restime) << std::endl;	47
705		48
706	cudaFree(d_data);	49
707	cudaFree(particles);	50

```

1 708     cudaFree(particles_old);
2 709     cudaFree(w);
3 710     cudaFree(d_neinum);
4 711     cudaFree(d_neibmat);
5 712     cudaFree(d_countmeans);
6 713
7 714     exit (EXIT_SUCCESS);
8 715
9 716 }
10 717
11 718
12 719 /* Use the Explicit Euler integration scheme to integrate SIR model
13     forward in time
14 720     float h      - time step size
15 721     float t0     - start time
16 722     float tn     - stop time
17 723     float * y    - current system state; a three-component vector
18                     representing [S I R], susceptible-infected-recovered
19 724 */
20 725 __device__ void exp_euler_SSIR(float h, float t0, float tn, Particle
21     * particle, int * neinum, int * neibmat, int nloc) {
22 726
23 727     int num_steps = floor( (tn-t0) / h );
24 728
25 729     float * S = particle->S;
26 730     float * I = particle->I;
27 731     float * R = particle->R;
28 732     float * B = particle->B;
29 733
30 734     // create last state vectors
31 735     float * S_last = (float*) malloc (nloc*sizeof(float));
32 736     float * I_last = (float*) malloc (nloc*sizeof(float));
33 737     float * R_last = (float*) malloc (nloc*sizeof(float));
34 738     float * B_last = (float*) malloc (nloc*sizeof(float));
35 739
36 740     float R0      = particle->R0;
37 741     float r       = particle->r;
38 742     float B0      = R0 * r / N;
39 743     float eta     = particle->eta;
40 744     float berr    = particle->berr;
41 745     float phi     = particle->phi;
42 746
43 747     for(int t = 0; t < num_steps; t++) {
44 748
45 749         for (int loc = 0; loc < nloc; loc++) {
46 750             S_last[loc] = S[loc];
47 751             I_last[loc] = I[loc];
48 752             R_last[loc] = R[loc];
49 753             B_last[loc] = B[loc];
50 754         }

```



```

755 |                                     | 1
756 |     for (int loc = 0; loc < nloc; loc++) { | 2
757 |                                     | 3
758 |         B[loc] = exp( log(B_last[loc]) + eta*(log(B0) - log( | 4
759 |             B_last[loc])) + berr*curand_normal(&(particle-> | 5
760 |                 randState)) ); | 6
761 |                                     | 7
762 |         int n = neinum[loc]; | 8
763 |         float sphl = 1.0 - phi*( (float) n/(n+1.0) ); | 9
764 |         float ophi = phi/(n+1.0); | 10
765 |                                     | 11
766 |         float nBIsu = 0.0; | 12
767 |         for (int j = 0; j < n; j++) | 13
768 |             nBIsu += B_last[neibmat[nloc*loc + j]-1] * I_last[ | 14
769 |                 neibmat[nloc*loc + j]-1]; | 15
770 |                                     | 16
771 |         float BSI = S_last[loc]*( sphl*B_last[loc]*I_last[loc] + | 17
772 |             ophi*nBIsu ); | 18
773 |         float rI = r*I_last[loc]; | 19
774 |                                     | 20
775 |         // get derivatives | 21
776 |         float dS = - BSI; | 22
777 |         float dI = BSI - rI; | 23
778 |         float dR = rI; | 24
779 |                                     | 25
780 |         // step forward by h | 26
781 |         S[loc] += h*dS; | 27
782 |         I[loc] += h*dI; | 28
783 |         R[loc] += h*dR; | 29
784 |                                     | 30
785 |     } | 31
786 | } | 32
787 | | 33
788 | free(S_last); | 34
789 | free(I_last); | 35
790 | free(R_last); | 36
791 | free(B_last); | 37
792 | } | 38
793 | | 39
794 | } | 40
795 | | 41
796 | /* Convenience function for particle resampling process | 42
797 | */ | 43
798 | __device__ void copyParticle(Particle * dst, Particle * src, int | 44
799 |     nloc) { | 45
800 | | 46
801 |     dst->R0 = src->R0; | 47
802 |     dst->r = src->r; | 48
803 |     dst->sigma = src->sigma; | 49
804 |     dst->eta = src->eta; | 50

```

```

1 800     dst->berr    = src->berr;
2 801     dst->phi    = src->phi;
3 802
4 803     for (int n = 0; n < nloc; n++) {
5 804         dst->S[n]      = src->S[n];
6 805         dst->I[n]      = src->I[n];
7 806         dst->R[n]      = src->R[n];
8 807         dst->B[n]      = src->B[n];
9 808         dst->Iinit[n]   = src->Iinit[n];
10 809     }
11 810
12 811 }
13 812
14 813 /* Convert memory size in bytes to human-readable format
15 814 */
16 815 std::string getHRmemsize (size_t memsize) {
17 816
18 817     std::stringstream ss;
19 818     std::string valstring;
20 819
21 820     int kb = 1024;
22 821     int mb = kb*1024;
23 822     int gb = mb*1024;
24 823
25 824     if (memsize <= kb)
26 825         ss << memsize << " B";
27 826     else if (memsize > kb && memsize <= mb)
28 827         ss << (float) memsize/ kb << " KB";
29 828     else if (memsize > mb && memsize <= gb)
30 829         ss << (float) memsize/ mb << " MB";
31 830     else
32 831         ss << (float) memsize/ gb << " GB";
33 832
34 833     valstring = ss.str();
35 834
36 835     return valstring;
37 836
38 837 }
39 838
40 839
41 840 /* Convert time in seconds to human readable format
42 841 */
43 842 std::string getHRtime (float runtime) {
44 843
45 844     std::stringstream ss;
46 845     std::string valstring;
47 846
48 847     int mt = 60;
49 848     int ht = mt*60;
50 849     int dt = ht*24;

```

850		1
851	if (runtime <= mt)	2
852	ss << runtime << " s";	3
853	else if (runtime > mt && runtime <= ht)	4
854	ss << runtime/mt << " m";	5
855	else if (runtime > ht && runtime <= dt)	6
856	ss << runtime/dt << " h";	7
857	else	8
858	ss << runtime/ht << " d";	9
859		10
860	valstring = ss.str();	11
861		12
862	return valstring;	13
863		14
864	}	15

The parameter estimation means as compared to IF2 and HMCMC are shown in Figure []. 17
18

The running times for parameter fitting as compared to IF2 and HMCMC are shown in Figure[]. 19
20